Tru Reputation Token Documentation

Release 0.1.12

Tru Ltd

Nov 25, 2018

Project Documentation

Last Modified On	4th December 2017
Revision Version	0.1.12

The Tru Reputation Token is part of the Tru Reputation Protocol by Tru Ltd and forms the cornerstone of the crypto-economic model in the Tru Reputation Protocol as detailed in the Tru Reputation Protocol Whitepaper. This project and documentation contains all design specifications, source code, security audits and testing suites for the Tru Reputation Token including all Crowd Sale Smart Contracts, and supporting Smart Contracts and Libraries. The contents of this Project can be used under the Apache 2 License.

CHAPTER 1

Project Contents

The **Tru Reputation Token** project has the following directory structure and contents:

Path	Overview	Detail Link
`./.github/`	Templates for GitHub	N/A
`./audits/`	Security Audits for the Project	*Security and Code Auditing*
`./audits/mythril`	**'mythril'_** audits for the Project	*Security and Code Auditing*
`./audits/oyente`	**'oyente'_** audits for the Project	*Security and Code Auditing*
`./contracts`	Project Solidity Smart Contracts	*Smart Contract Reference*
`./contracts/supporting`	Supporting Solidity Contracts & Libraries	*Supporting Smart Contract Reference*
`./contracts/test-helpers`	Testing Helper Solidity Contracts	*2. Testing Helpers & Harnesses*
`./docs/`	Source files for this documentation	N/A
`./migrations/`	truffle migration contract	N/A
`./scripts/`	Supporting Project Scripts	*Supporting Scripts*
`./src/`	Flat Project Source files	*Security and Code Auditing*
`./test/`	Testing suite for Project	*3. Unit Tests*
`./test/helpers/`	Helpers for Testing Suite	*2. Testing Helpers & Harnesses*
`./tru-devnet/`	Folder containing configuration for Tru-DevNet Geth Network	N/A
`.babelrc`	Babel configuration for Project	N/A
`.gitignore`	git ignore file for Project	N/A
`.jshintrc`	JSHint configuration for Project	N/A
`.solcover.js`	solidity-coverage configuration for Project	N/A
`.solhint.json`	Solhint configuration for Project	N/A
`.soliumignore`	Solium ignore file for Project	N/A
`.soliumrc.json`	Solium configuration for Project	N/A
`.travis.yml`	Travis CI configuration for Project	N/A
`LICENSE`	Apache 2.0 License for Project	N/A
`package-lock.json`	Package Lock file for Project	N/A
`package.json`	Package file for Project	N/A
`README.md`	Readme file for Project	N/A
`truffle.js`	Configuration file for truffle	N/A

CHAPTER 2

Project Requirements

Author Ian Bray

Revision Date 26/11/2017

The following sections break down the requirements for the Tru Reputation Token and any associated Sale Smart Contracts, supporting libraries, security or testing requirements

2.1 Token Requirements

2.1.1 Tru Reputation Token Requirements

When designing the **Tru Reputation Token** the following requirements were specified:

Requirement	Requirement Description
TRTREQ 001	Token must be ERC-20 compliant
TRTREQ 002	Token must support up to 10^18 decimal places
TRTREQ 003	Token must be named Tru Reputation Token
TRTREQ 004	Token must use TRU as its symbol
TRTREQ 005	Token must only be minted during Sale events
TRTREQ 006	Token must have an address for the Executive Board
TRTREQ 007	Only the Executive Board should be able to change the Executive Board address
TRTREQ 008	Token must be able to be upgraded in the future
TRTREQ 009	Token upgrades should only occur through consensus
TRTREQ 010	Token upgrades should only be able to be set by Smart Contract owner
TRTREQ 011	Tokens should not be able to be transferred until all Sales are completed
TRTREQ 012	All Smart Contract code must be fully unit tested
TRTREQ 013	All Smart Contract code must be fully fuzz tested
TRTREQ 014	All Smart Contract code must be security audited

TRTREQ 001

Requirement: Token must be ERC-20 compliant

Description: To maintain optimal compatibility, security, functionality and in line with Best Practice the **Tru Reputation Token** must be ERC-20 Compliant.

Implementation Notes: Leveraged Zeppelin Solidity to ensure ERC-20 compliance by using a sub-class of the ERC20 Smart Contract.

Requirement Met? Yes

TRTREQ 002

Requirement: Token must support up to 10^18 decimal places

Description: To ensure future compatibility and in line with having an entirely fixed token supply, the **Tru Reputation Token** must support 10^18 decimal places to allow the highest level of granular fractionality for the Token. This is to ensure the precise reward & cost models of the **Tru Reputation Protocol** are met.

Implementation Notes: Leveraged Zeppelin Solidity to ensure ERC-20 compliance and set the *decimals* constant variable to **18**.

Requirement Met? Yes

TRTREQ 003

Requirement: Token must be named Tru Reputation Token

Description: To ensure the appropriate identification of the **Tru Reputation Token** it must be named as such in the Smart Contract and be publicly visible.

Implementation Notes: Leveraged Zeppelin Solidity to ensure ERC-20 compliance and set the *name* constant variable to **Tru Reputation Token**.

Requirement Met? Yes

TRTREQ 004

Requirement: Token must use TRU as its symbol

Description: To ensure the appropriate identification of the **Tru Reputation Token** it must be have its token symbol set to **TRU**.

Implementation Notes: Leveraged Zeppelin Solidity to ensure ERC-20 compliance and set the *symbol* constant variable to **TRU**.

Requirement Met? Yes

TRTREQ 005

Requirement: Token must only be minted during Sale events

Description: To prevent oversupply, the **Tru Reputation Token** should only be able to be minted during a Crowdsale event and once complete, no further tokens should be able to be minted.

Implementation Notes: Leveraged a customised version of the Zeppelin Solidity **MintableToken** the base token has the capability to both be minted and to be able to be finalise that minting process fully and finally. In conjunction with the TruSale Smart Contract and customizations made to the **MintableToken** to set these finalisation criteria.

Requirement Met? Yes

TRTREQ 006

Requirement: Token must have an address for the Executive Board

Description: As per the Tru Reputation Protocol Whitepaper the Tru Reputation Protocol will be governed, steered and maintained by an Advisory Board. Motions passed by the Advisory Board will need to be enacted by the Tru Ltd Executive Board and for that purpose a multi-signature wallet will be created to enact those changes. The address of this wallet needs to be set upon construction of the Smart Contract.

Implementation Notes: Implemented using the *execBoard* address variable, the *onlyExecBoard* modifier, *BoardAddressChanged* event, and the *changeBoardAddress* function that can only be executed by the existing Executive Board address. This implemenetation addresses this requirement and *TRTREQ 007*.

Requirement Met? Yes

TRTREQ 007

Requirement: Only the Executive Board should be able to change the Executive Board address

Description: In conjunection with *TRTREQ 006* only the current Executive Board Address should be able to change the Executive Board Address to a different value and there should be a full audit trail of any changes made.

Implementation Notes: Implemented along with *TRTREQ 006* by the *onlyExecBoard* modifier, *BoardAddressChanged* event, and the *changeBoardAddress* function.

Requirement Met? Yes

TRTREQ 008

Requirement: Token must be able to be upgraded in the future

Description: To allow the **Tru Reputation Protocol** to deliver new functionality and fix any potential issues, the **Tru Reputation Token** needs to have a mechanism to allow the Token to be upgraded over time.

Implementation Notes: By leveraging an updated version of the **UpgradeableToken** (**TruUpgradeableToken**) and **UpgradeAgent** Smart Contracts by Token Market, the **Tru Reputation Token** can be upgraded in the future.

Requirement Met? Yes

TRTREQ 009

Requirement: Token upgrades should only occur through consensus

Description: In line with the guiding principles of cryptocurrency, contract law & customs, any Token upgrade should require consensus of Token holders to adopt any upgrade to the Token.

Implementation Notes: By leveraging an updated version of the **UpgradeableToken** (**TruUpgradeableToken**) and **UpgradeAgent** Smart Contracts by Token Market, the **Tru Reputation Token** is upgraded by the Token holder when they can opt in to any potential upgrade.

Requirement Met? Yes

TRTREQ 010

Requirement: Token upgrades should only be able to be set by Smart Contract owner

Description: To protect **Tru Reputation Token** from malicious third-parties, only the owner of the Token Smart Contract should be able to provide any upgrade to the Smart Contract.

Implementation Notes: By leveraging an updated version of the **ReleaseableToken** Smart Contract by Token Market, the **Tru Reputation Token** has to be set to a released state explicitly before the Token can be exchanged or transferred between wallets beyond the initial address that purchased the tokens. By adding this event in the closing logic of the last Sale event, the capability to transfer **Tru Reputation Token** can be set at a time after that event.

Requirement Met? Yes

TRTREQ 011

Requirement: Tokens should not be able to be transferred until all Sales are completed

Description: To prevent Pre-Launch transfer of Tokens the **Tru Reputation Token** needs to be non-transferable until any and all Sale events have concluded.

Implementation Notes: By leveraging an updated version of the **ReleaseableToken** Smart Contract by Token Market, the **Tru Reputation Token** has to be set to a released state explicitly before the Token can be exchanged or transferred between wallets beyond the initial address that purchased the tokens. By adding this event in the closing logic of the last Sale event, the capability to transfer **Tru Reputation Token** can be set at a time after that event.

Requirement Met? Yes

TRTREQ 012

Requirement: All Smart Contract code must be fully unit tested

Description: All **Tru Reputation Token** Smart Contract functionality should be testable and verifable through unit tests.

Implementation Notes: Leveraging Truffle, Mocha Unit Tests have been created for the **Tru Reputation Token** Smart Contracts and supporting Smart Contracts. This Tesing Suite will be updated as code changes to ensure 100% coverage of lines, statements, functions & branches in the testing suite.

Requirement Met? Yes and ongoing

TRTREQ 013

Requirement: All Smart Contract code must be fully fuzz tested

Description: In keeping with good security practice, all **Tru Reputation Token** Smart Contract code must be fully fuzz tested where fuzzing would be applicable to prevent exploits in the Smart Contract.

Implementation Notes: Leveraging Truffle, and a Fuzzing Library for Javascript additional tests have been been created for the **Tru Reputation Token** Smart Contracts and supporting Smart Contracts. These tests stress each function and check for exploits and failures to ensure the security and robustness of the Smart Contracts. These tests are within the Mocha Test Suite and will be updated as code changes to ensure 100% coverage of lines, statements, functions & branches in the testing suite.

Requirement Met? Yes and ongoing

TRTREQ 014

Requirement: All Smart Contract code must be fully security audited

Description: Leveraging tools such as Oyente, all **Tru Reputation Token** Smart Contract code must be subjected to Static Analysis and security audit.

Implementation Notes: Oyente auditing has been implemented for all **Tru Reputation Token** Smart Contracts.

Requirement Met? Yes and ongoing

2.2 Sale Requirements

2.2.1 Common Sale Requirements

When designing the Pre-Sale and CrowdSale Smart Contracts for the **Tru Reputation Token** the following common requirements were specified:

Requirement	Requirement Description
SALREQ 001	Each sale must have a maximum cap of Tokens to be sold
SALREQ 002	Each sale should have a Start and End time
SALREQ 003	No purchases should be able to be made before Sale Start
SALREQ 004	No purchases should be able to be made after Sale End
SALREQ 005	Each sale must end if cap is hit
SALREQ 006	Each sale must end if end time has passed
SALREQ 007	Each sale must have a distinct Eth to Tru purchase rate
SALREQ 008	Each sale must track the amount of tokens sold
SALREQ 009	Each sale must track the amount of ETH raised
SALREQ 010	Each sale must track the number of purchasers
SALREQ 011	Each sale must pay all funds raised to a dedicated wallet
SALREQ 012	The end time of a Sale should be able to be changed
SALREQ 013	Each sale must have a AML/KYC Whitelist
SALREQ 014	Each sale must have maximum buy limit for non-WhiteListed accounts
SALREQ 015	Each sale must have a minimum buy limit for all buyers
SALREQ 016	Each sale must be able to be halted in an emergency
SALREQ 017	Each sale must mint tokens at the time of purchase
SALREQ 018	Each sale must mint appropriate amount of tokens for Tru Ltd when a purchase occurs
SALREQ 019	All buy activity on sales must be audited
SALREQ 020	All updates to the Whitelist must be audited
SALREQ 021	Must be able to remove an address from the WhiteList
SALREQ 022	All updates to the Sale End Time must be audited
SALREQ 023	Post Sale rate should be set to 1,000 TRU per ETH
SALREQ 024	No more than 125,000,000 TRU should be minted during the Sales

SALREQ 001

Requirement: Each sale must have a maximum cap of Tokens to be sold

Description: Each sale that occurs for **Tru Reputation Tokens** must have a maximum cap for that sale. In addition, there needs to be a global maximum cap for all Sales. If a previous Sale fails to raise to its cap, the remainder of the cap should carry forward to the next Sale.

Implementation Notes: Implemented using the *cap* variable and logic in the Constructor of child Smart Contracts.

Requirement Met? Yes

SALREQ 002

Requirement: Each sale should have a Start and End time

Description: Each sale that occurs for **Tru Reputation Tokens** must have a fixed Start Time and fixed End Time.

Implementation Notes: Implemented using *saleStartTime* and *saleEndTime* variables, the ref:*tru-sale-has-ended* constant function, and requiring the *saleStartTime* and *saleEndTime* variables in the constructor (*TruSale Constructor*).

Requirement Met? Yes

SALREQ 003

Requirement: No purchases should be able to be made before Sale Start

Description: No one should be able to purchase from a sale before a sale of **Tru Reputation Tokens** occurs.

Implementation Notes: Implemented using logic in the *buy* function to check that the Sale has started.

Requirement Met? Yes

SALREQ 004

Requirement: No purchases should be able to be made after Sale End

Description: Once the end time for the sale of **Tru Reputation Tokens** completes, no one should be able to purchase any further tokens from the sale.

Implementation Notes: Implemented using logic in the *buy* function and *hasEnded* constant function.

Requirement Met? Yes

SALREQ 005

Requirement: Each sale must end if cap is hit

Description: Once the cap for the sale of **Tru Reputation Tokens** is reached, the sale should be considered completed and no one should be able to purchase any further tokens from the sale.

Implementation Notes: Implemented using the *cap* variable, and logic in the *hasEnded* constant function.

Requirement Met? Yes

SALREQ 006

Requirement: Each sale must end if end time has passed

Description: Once the end time for the sale of **Tru Reputation Tokens** is reached, the sale should be considered completed and no one should be able to purchase any further tokens from the sale.

Implementation Notes: Implemented using logic in the *hasEnded* constant function.

Requirement Met? Yes

SALREQ 007

Requirement: Each sale must have a distinct Eth to Tru purchase rate

Description: Each sale of **Tru Reputation Tokens** must have its clear purchase rate of Tru per Ether to reflect the bonus applied for each Sale round. The post sale price should also be publicly visible within the sale Smart Contract.

Implementation Notes: Implemented using the *BASE_RATE*, *PRESALE_RATE*, *SALE_RATE*, *isPreSale* and *isCrowdSale* variables, and logic in the *buyTokens* function.

Requirement Met? Yes

SALREQ 008

Requirement: Each sale must track the amount of tokens sold

Description: Each sale of **Tru Reputation Tokens** must track the total number of **Tru Reputation Tokens** sold during that Sale through a publicly visible variable.

Implementation Notes: Implemented using the *tokenAmount* mapping and *soldTokens* variable.

Requirement Met? Yes

SALREQ 009

Requirement: Each sale must track the amount of ETH raised

Description: Each sale of **Tru Reputation Tokens** must track the total number of **ETH** raised during that Sale through a publicly visible variable.

Implementation Notes: Implemented using the *purchasedAmount* mapping.

Requirement Met? Yes

SALREQ 010

Requirement: Each sale must track the number of purchasers

Description: Each sale of **Tru Reputation Tokens** must track the total number of purchasers within that Sale. In addition, each purchaser and the amount purchased needs to be visible through a mapping to validate each purchase and provide an audit trail.

Implementation Notes: Implemented using the *purchaserCount* variable.

Requirement Met? Yes

SALREQ 011

Requirement: Each sale must pay all funds raised to a dedicated wallet

Description: Each sale of **Tru Reputation Tokens** must collect all raised funds in a dedicated wallet separate from the address that created the Smart Contract.

Implementation Notes: Implemented using the *multiSigWallet* address variable and requiring this be set on construction to act as the receiving wallet for all funds raised during the sale.

Requirement Met? Yes

SALREQ 012

Requirement: The end time of a Sale should be able to be changed

Description: The end time of each sale of **Tru Reputation Tokens** must be able to be changed in the event of an emergency by the Smart Contract owner (for example: closing a sale early, or extending the window due to an issue with the Ethereum network). This should only be able to be performed by the owner of the Sale Smart Contract.

Implementation Notes: Implemented using the *changeEndTime* function and leveraging the *onlyOwner* modifier.

Requirement Met? Yes

SALREQ 013

Requirement: Each sale must have a AML/KYC Whitelist

Description: Each sale of **Tru Reputation Tokens** must have a Whitelist of addresses connected to individuals and entities that have been validated off-chain in line with Anti-Money Laundering and Know Your Customer legislation & practice. Only the owner of the Sale Smart Contract should be able to amend this Whitelist.

Implementation Notes: Implemented using the *purchaserWhiteList* mapping, the *updateWhitelist* function and leveraging the *onlyOwner* modifier.

Requirement Met? Yes

SALREQ 014

Requirement: Each sale must have maximum buy limit for non-WhiteListed accounts

Description: Each sale of **Tru Reputation Tokens** must have a cumulative maximum amount of tokens a given address can purchase if they are not on the AML/KYC Whitelist. This limit should be set to 20 ETH.

Implementation Notes: Implemented using the *MAX_AMOUNT* variable and logic in the *buyTokens* function.

Requirement Met? Yes

SALREQ 015

Requirement: Each sale must have a minimum buy limit for all buyers

Description: Each sale of **Tru Reputation Tokens** must have a minimum amount of tokens a given address can purchase to participate in a sale. This minimum limit must be set to 1 ETH.

Implementation Notes: Implemented using the *MIN_AMOUNT* variable and logic in the *buyTokens* function.

Requirement Met? Yes

SALREQ 016

Requirement: Each sale must be able to be halted in an emergency

Description: Each sale of **Tru Reputation Tokens** must have the capability to be halted by the Sale Smart Contract owner in an emergency event that should stop the Sale. It should also have the capability to be unhalted. This should only be able to be performed by the owner of the Sale Smart Contract.

Implementation Notes: Leveraged a modified version of the the **Haltable** by Token Market.

Requirement Met? Yes

SALREQ 017

Requirement: Each sale must mint tokens at the time of purchase

Description: To prevent oversupply of tokens, each sale of **Tru Reputation Tokens** must mint tokens only at the time of purchase. This will remove the need to 'burn' tokens, and ensure stability of supply.

Implementation Notes: Implemented a modified version of **MintableToken** (**TruMintableToken**) by Zeppelin Solidity and implemented logic in the *buyTokens* function.

Requirement Met? Yes

SALREQ 018

Requirement: Each sale must mint appropriate amount of tokens for Tru Ltd when a purchase occurs

Description: As per *SALREQ 018*, to prevent oversupply of tokens each sale of **Tru Reputation Tokens** must mint an additional token for each token purchased and assign that to Tru Ltd's wallet to comply with the 50% sale of token supply as per the Tru Reputation Protocol Whitepaper.

Implementation Notes: Implemented a modified version of **MintableToken** (**TruMintableToken**) by Zeppelin Solidity and implemented logic in the *completion* function to mint the same number of tokens bought in a sale to match the number sold in that Sale rather than mint them at the moment of purchase.

Requirement Met? Yes

SALREQ 019

Requirement: All buy activity on sales must be audited

Description: Each sale of **Tru Reputation Tokens** must audit and track each time an address buys tokens, and include the purchaser address, the recipient address, the amount paid and the number of tokens purchased.

Implementation Notes: Implemented using the *TokenPurchased* event that is fired each time a purchase is successful. Event includes the address of the purchaser, the destination address (fixed to be the same in this implementation, but potentially could be different in another), the total amount spent and the total amount of tokens bought.

Requirement Met? Yes

SALREQ 020

Requirement: All updates to the Whitelist must be audited

Description: Each sale of **Tru Reputation Tokens** must audit and track each time the AML/KYC Whitelist is updated and include the Whitelisted address and its status on the Whitelist.

Implementation Notes: Implemented using the *WhiteListUpdated* event that is fired each time a Whitelist entry is added or updated. The event includes the address and their status on the Whitelist (true for enabled, false for disabled).

Requirement Met? Yes

SALREQ 021

Requirement: Must be able to remove an address from the WhiteList

Description: Each sale of **Tru Reputation Tokens** must offer the capability to remove or disable an address currently on the Whitelist. This should only be able to be performed by the owner of the Sale Smart Contract.

Implementation Notes: Implemented via the *purchaserWhiteList* mapping of a bool variable to an address. When that variable is set to *true* they are active and enabled on the Whitelist. When it is sent to *false* they are disabled and in effect 'removed' from the Whitelist. This status is checked by the *validatePurchase* function rather than purely checking they have an entry on the Whitelist.

Requirement Met? Yes

SALREQ 022

Requirement: All updates to the Sale End Time must be audited

Description: Each sale of **Tru Reputation Tokens** must audit and track each time the End Time for the sale is changed.

Implementation Notes: Implemented using the *EndChanged* event that is fired each time the *saleEndTime* variable is altered from its initial value. The event includes the both the old and the new end time.

Requirement Met? Yes

SALREQ 023

Requirement: Post Sale rate should be set to 1,000 TRU per ETH

Description: Each sale of **Tru Reputation Tokens** must have a publicly visible variable showing the Base Exchange Rate of 1,000 TRU per ETH

Implementation Notes: Implemented using the *BASE_RATE* variable.

Requirement Met? Yes

SALREQ 024

Requirement: No more than 125,000,000 TRU should be minted during the Sales

Description: The combined total of all Sales should not mint more than 125,000,000 **Tru Reputation Tokens**. Of this no more than 62,500,000 TRU should be sold with the remainder being minted for distribution by Tru Ltd as per the Tru Reputation Protocol Whitepaper.

Implementation Notes: Implemented using the ETH cap and buy rates ensuring that only 62,500,000 **Tru Reputation Tokens** can be sold, and that only a further 62,500,000 **Tru Reputation Tokens** can be minted to the sale wallet.

Requirement Met? Yes

2.2.2 Pre-Sale Requirements

When designing the Pre-Sale Smart Contract for the **Tru Reputation Token** the following common requirements were specified:

Requirement	Requirement Description
PSREQ 001	Cap for Pre-Sale must be fixed at 5,000 ETH
PSREQ 002	Sale Rate for Pre-Sale must be 1,250 TRU per ETH

PSREQ 001

Requirement: Cap for Pre-Sale must be fixed at 5,000 ETH

Description: The cap for the Pre-Sale of **Tru Reputation Token** must have a fixed sale cap of 8,000 ETH

Implementation Notes: Implemented by setting the *PRESALE_CAP* to 8000 x 10^18, and logic within the *validatePurchase* function.

Requirement Met? Yes

PSREQ 002

Requirement: Sale Rate for Pre-Sale must be 1,250 TRU per ETH

Description: The buy price for the Pre-Sale of **Tru Reputation Token** must be 1,250 TRU per ETH. This equals a 25% bonus/20% discount versus the Base Rate.

Implementation Notes: Implemented using logic within the *validatePurchase* function, and setting a constant variable for the *PRESALE_RATE* to 1250.

Requirement Met? Yes

2.2.3 CrowdSale Requirements

When designing the CrowdSale Smart Contract for the **Tru Reputation Token** the following common requirements were specified:

Requirement	Requirement Description
CSREQ 001	Cap for CrowdSale should be cumulative with any unsold Pre-Sale Cap
CSREQ 002	Cap for CrowdSale must be fixed to 50,000 ETH
CSREQ 003	Sale Rate for Pre-Sale should be 1,125 TRU per ETH

CSREQ 001

Requirement: Cap for CrowdSale should be cumulative with any unsold Pre-Sale Cap

Description: The cap for the CrowdSale of **Tru Reputation Token** must include any unsold tokens from the Pre-Sale (e.g. if only 4,000 ETH worth of Tru Tokens are sold during the Pre-Sale, this must be added to the CrowdSale cap).

Implementation Notes: Implemented using logic in the CrowdSale constructor to ensure that the result of the PreSale is passed into the constructor and the *TOTAL_CAP*, and then removing the PreSale raised amount from the *TOTAL_CAP*.

Requirement Met? Yes

CSREQ 002

Requirement: Cap for CrowdSale must be fixed to 80,000 ETH

Description: The cap for the CrowdSale of **Tru Reputation Token** must fixed at 80,000 ETH excluding any potential unsold cap from the Pre-Sale as per *CSREQ 001*. For example: If the Pre-Sale sells all 8,000 ETH worth of Tokens, then the CrowdSale will have a cap of 80,000 ETH. However, if the Pre-Sale only sells 7,000 ETH than the cap for the CrowdSale should be 81,000 ETH.

Implementation Notes: By setting the *TOTAL_CAP* to 88000 x 10^18, and logic within the constructor for the CrowdSale Smart Contract to remove total raised to date from the initial

Requirement Met? Yes

CSREQ 003

Requirement: Sale Rate for CrowdSale should be 1,125 TRU per ETH

Description: The buy price for the CrowdSale of **Tru Reputation Token** must be 1,125 TRU per ETH. This equals a 12.5% bonus/11.11...% discount versus the Base Rate.

Implementation Notes: Implemented using logic within the *validatePurchase* function, and setting a constant variable for the *SALE_RATE* to 1125, this requirement.

Requirement Met? Yes

CHAPTER 3

Project Testing

The following section covers the testing strategy and implementation for all Smart Contracts in the **Tru Reputation Token** project including supporting Libraries & Smart Contracts.

3.1 1. Strategy

The Testing Strategy for the **Tru Reputation Token** Project is as defined below:

- Due to the inherent financial risk of Cryptocurrency, and the nature of Solidity, all Contract code including any supporting Smart Contracts must be subjected to full coverage unit tests to cover all lines, statements, branches and functions.
- All testing is to be conducted on each commit to the Repository.
- Testing will include, as much as practicable, all contrary cases that could cause any failure.
- The **Tru Reputation Token** Project will not be released without the above items being met.

3.2 2. Testing Helpers & Harnesses

To facilitate full coverage, the following Testing Helpers and Harnesses have been created:

Name	Detail
MockFailUpgradeAgent.sol	Test harness of an *UpgradeAgent* used to test failure paths for upgrades on the *TruReputationToken*
MockMigrationTarget.sol	Test harness of an *TruReputationToken* to simulate an upgrade of the token
MockSale.sol	Test harness of a TruSale to provide full coverage of failure paths
MockSupportToken.sol	Test harness for full failure path testing of *StandardToken*
MockUpgradeableToken.sol	Test harness of an Upgradeable token for testing of the *TruUpgradeableToken* Smart Contract
MockUpgradeableAgent.sol	Test harness of an Upgradeable token for testing of the *UpgradeAgent* Smart Contract
EVMInvalidAddress.js	Javascript helper for catching *Invalid Address* errors from EVM
EVMRevert.js	Javascript helper for catching *Revert* errors from EVM
EVMThrow.js	Javascript helper for catching *Throw* errors from EVM
expectFuzzFail.js	Javascript helper for catching Fuzzing failure errors from EVM
expectNotDeployed.js	Javascript helper for catching *Not Deployed* errors from EVM
expectThrow.js	Javascript helper promise for catching *Throw* errors
increaseTime.js	Javascript helper to change current time on TestRPC
isEven.js	Javascript helper to detect in a number is odd or even
latestTime.js	Javascript helper to get current timestamp of block on TestRPC

3.2.1 MockFailUpgradeAgent.sol

Name:	MockFailUpgradeAgent.sol
Type:	Solidity Contract
Path:	`./contracts/test-helpers/MockFailUpgradeAgent.sol`
Detail:	Test harness of an *UpgradeAgent* used to test failure paths for upgrades on the *TruReputationToken*
Author:	Tru Ltd

3.2.2 MockMigrationTarget.sol

Name:	MockMigrationTarget.sol
Type:	Solidity Contract
Path:	`./contracts/test-helpers/MockMigrationTarget.sol`
Detail:	Test harness of an *TruReputationToken* to simulate an upgrade of the token
Author:	Tru Ltd

3.2.3 MockSale.sol

Name:	MockSale.sol
Type:	Solidity Contract
Path:	`./contracts/test-helpers/MockSale.sol`
Detail:	Test harness of a TruSale to provide full coverage of failure paths
Author:	Tru Ltd

3.2.4 MockSupportToken.sol

Name:	MockSupportToken.sol
Type:	Solidity Contract
Path:	`./contracts/test-helpers/MockSupportToken.sol`
Detail:	Test harness for full failure path testing of *StandardToken*
Author:	Tru Ltd

3.2.5 MockUpgradeableToken.sol

Name:	MockUpgradeableToken.sol
Type:	Solidity Contract
Path:	`./contracts/test-helpers/MockUpgradeableToken.sol`
Detail:	Test harness of an Upgradeable token for testing of the *TruUpgradeableToken* Smart Contract
Author:	Tru Ltd

3.2.6 MockUpgradeableAgent.sol

Name:	MockUpgradeableAgent.sol
Type:	Solidity Contract
Path:	`./contracts/test-helpers/MockUpgradeableAgent.sol`
Detail:	Test harness of an Upgradeable token for testing of the *UpgradeAgent* Smart Contract
Author:	Tru Ltd

3.2.7 EVMInvalidAddress.js

Name:	EVMInvalidAddress.js
Type:	Solidity Contract
Path:	`./test/helpers/EVMInvalidAddress.js`
Detail:	Javascript helper for catching *Invalid Address* errors from EVM
Author:	Tru Ltd

3.2.8 EVMRevert.js

Name:	EVMRevert.js
Type:	Solidity Contract
Path:	`./test/helpers/EVMRevert.js`
Detail:	Javascript helper for catching *Revert* errors from EVM
Author:	Tru Ltd

3.2.9 EVMThrow.js

Name:	EVMThrow.js
Type:	Solidity Contract
Path:	`./test/helpers/EVMThrow.js`
Detail:	Javascript helper for catching *Throw* errors from EVM
Author:	Zeppelin Solidity

3.2.10 expectFuzzFail.js

Name:	expectFuzzFail.js
Type:	Solidity Contract
Path:	`./test/helpers/expectFuzzFail.js`
Detail:	Javascript helper for catching Fuzzing failure errors from EVM
Author:	Tru Ltd

3.2.11 expectNotDeployed.js

Name:	expectNotDeployed.js
Type:	Solidity Contract
Path:	`./test/helpers/expectNotDeployed.js`
Detail:	Javascript helper for catching *Not Deployed* errors from EVM
Author:	Tru Ltd

3.2.12 expectThrow.js

Name:	expectThrow.js
Type:	Solidity Contract
Path:	`./test/helpers/expectThrow.js`
Detail:	Javascript helper promise for catching *Throw* errors
Author:	Zeppelin Solidity

3.2.13 increaseTime.js

Name:	increaseTime.js
Type:	Solidity Contract
Path:	`./test/helpers/increaseTime.js`
Detail:	Javascript helper to change current time on TestRPC
Author:	Zeppelin Solidity

3.2.14 isEven.js

Name:	isEven.js
Type:	Solidity Contract
Path:	`./test/helpers/isEven.js`
Detail:	Javascript helper to detect in a number is odd or even
Author:	Tru Ltd

3.2.15 latestTime.js

Name:	latestTime.js
Type:	Solidity Contract
Path:	`./test/helpers/latestTime.js`
Detail:	Javascript helper to get current timestamp of block on TestRPC
Author:	Zeppelin Solidity

3.3 3. Unit Tests

The following Unit Tests are defined for the **Tru Reputation Token** project:

3.3.1 3.1. TruReputationToken Unit Tests

Name:	TruReputationToken Unit Tests
Path:	`./test/Unit_Tests_TruReputationToken.js`
Detail:	Unit Tests covering the TruReputationToken.sol Smart Contract
No of Test Cases:	35

	Description
01	TruReputationToken should have correct name, symbol and description
02	Owner should be able to assign Executive Board Address once
03	No other account should be able to change Executive Board Address
04	Should be unable to assign an empty address as Exec Board
05	Should be unable to assign an self as Exec Board
06	Exec Board should be able to assign different Exec Board Account
07	TruReputationToken should have 0 total supply
08	Only TruReputationToken owner can set the Release Agent
09	Only TruReputationToken Owner can set transferAgent
10	mintingFinished should be false after construction
11	Should fail to deploy new Upgrade Token with no tokens
12	Should mint a token with 10^18 decimal places
13	Should mint 100 tokens to a supplied address
14	Should fail to mint after calling finishMinting
15	Token should have correct Upgrade Agent
16	Should deploy new Upgrade Token
17	Should fail to set empty UpgradeMaster

Continued on next page

Table 1 – continued from previous page

18	Should fail to set UpgradeMaster if not already master
19	Should set UpgradeMaster if already master
20	Token should be able to set the upgrade
21	Token should not upgrade without an upgrade agent set
22	Should not set an upgrade agent with empty address
23	Should not set an upgrade agent with a Token that is not allowed to upgrade
24	Should set an upgrade agent that is not an upgrade agent
25	Should set an upgrade agent
26	Only Token owner can set upgrade
27	Token should not upgrade with an empty upgrade amount
28	Token should not upgrade from an account without tokens
29	Token should not upgrade with an amount greater than the supply
30	Should upgrade the token
31	UpgradeAgent should not be changed after the upgrade has started
32	MockMigrationTarget should revert on attempt to transfer to it
33	Functions increaseApproval & decreaseApproval should increase & decrease approved allowance
34	Function transferFrom should fail with invalid values

3.3.2 3.2. TruPreSale Unit Tests

Name:	TruPreSale Unit Tests
Path:	`./test/Unit_Tests_TruPreSale.js`
Detail:	Unit Tests covering the TruPreSale.sol Smart Contract
No of Test Cases:	36

	Description
01	Cannot deploy TruPreSale with incorrect variables
02	TruPreSale and TruReputationToken are deployed
03	Fallback function should revert
04	Pre-Sale hard variables are as expected
05	Set Release Agent for TruReputationToken
06	Transfer TruReputationToken ownership to Pre-Sale
07	Can Add Purchaser to Purchaser Whitelist
08	Can Remove Purchaser from Purchaser Whitelist
09	Cannot purchase before start of Pre-Sale
10	Cannot purchase below minimum purchase amount
11	Cannot purchase above maximum purchase amount if not on Whitelist
12	Can purchase above maximum purchase amount if on Whitelist
13	Can halt Pre-Sale in an emergency
14	Tokens cannot be transferred before Pre-Sale is finalised
15	Only nominated Release Agent can make Tokens transferable
16	Only Token Owner can mint Tokens
17	Has correct Purchaser count
18	Cannot buy more than cap
19	Pre-Sale owner cannot finalise a Pre-Sale before it ends
20	Cannot buy with invalid address
21	Cannot buy 0 amount
22	Can buy repeatedly from the same address

Continued on next page

Table 2 – continued from previous page

23	Can buy up to the cap on the Pre-Sale
24	Cannot buy once the cap is reached on the Pre-Sale
25	Cannot buy once Pre-Sale has ended
26	Pre-Sale owner can finalise the Pre-Sale
27	Cannot finalise a finalised Pre-Sale
28	Minted TruReputationToken cannot be transferred yet
29	Can change Pre-Sale end time to further into the future
30	Cannot change Pre-Sale end time to less than start time
31	Can change Pre-Sale end time to less than current end time
32	Can change Pre-Sale end time to less than current time & end sale
33	Only Pre-Sale Owner can change Pre-Sale end time
34	Cannot create Pre-Sale with end time before start time
35	Cannot create Pre-Sale with invalid Token Address
36	Cannot create Pre-Sale with invalid Sale Wallet Address

3.3.3 3.3. TruCrowdSale Unit Tests

Name:	TruCrowdSale Unit Tests
Path:	`./test/Unit_Tests_TruCrowdSale.js`
Detail:	Unit Tests covering the TruCrowdSale.sol Smart Contract
No of Test Cases:	37

	Description
01	Cannot deploy TruCrowdSale with incorrect variables
02	TruPreSale and TruReputationToken are deployed
03	Simulate completed PreSale and transition to CrowdSale
04	Fallback function should revert
05	CrowdSale hard variables are as expected
06	Transfer TruReputationToken ownership to CrowdSale
07	Can Add Purchaser to CrowdSale Purchaser Whitelist
08	Can Remove Purchaser from CrowdSale Purchaser Whitelist
09	Cannot purchase before start of CrowdSale
10	Cannot purchase below minimum purchase amount
11	Cannot purchase above maximum purchase amount if not on CrowdSale Whitelist
12	Can purchase above maximum purchase amount if on CrowdSale Whitelist
13	Can halt CrowdSale in an emergency
14	Tokens cannot be transferred before CrowdSale is finalised
15	Only nominated Release Agent can make Tokens transferable
16	Only Token Owner can mint Tokens
17	CrowdSale has correct Purchaser count
18	Cannot buy more than CrowdSale cap
19	CrowdSale owner cannot finalise a CrowdSale before it ends
20	Cannot buy from CrowdSale with invalid address
21	Cannot buy 0 amount from CrowdSale
22	Can buy repeatedly from the same address
23	Can buy up to the cap on the CrowdSale
24	Cannot buy once the cap is reached on the CrowdSale
25	CrowdSale owner can finalise the CrowdSale

Continued on next page

Table 3 – continued from previous page

26	Cannot buy once CrowdSale has ended
27	Cannot finalise a finalised CrowdSale
28	Minted TruReputationToken can be transferred
29	CrowdSale has higher cap if PreSale did not hit cap
30	Can change CrowdSale end time to further into the future
31	Cannot change CrowdSale end time to less than start time
32	Can change CrowdSale end time to less than current end time
33	Can change CrowdSale end time to less than current time & end sale
34	Only Crowdsale Owner can change CrowdSale end time
35	Cannot create Crowdsale with end time before start time
36	Cannot create Crowdsale with invalid Token Address
37	Cannot create Crowdsale with invalid Sale Wallet Address

3.4 4. Fuzzing Tests

To ensure a robust testing strategy to ensure code quality and predictability, using fuzzing testing can expose non-obvious exploits through testing non-obvious code paths and reactions to large numbers of tests with large amount of data.

To ensure the security and stability of the **Tru Reputation Protocol** and the **Tru Reputation Token** project, Fuzzing is performed on all Smart Contracts to expose and remedy any potential vulnerabilities or exploits introduced in each release cycle.

Due to the nature of fuzzing and the defaults of Mocha and some characteristics of the TestRPC network these tests can take up to an hour to execute.

3.4.1 4.1. TruReputationToken Fuzzing Tests

Name:	TruReputationToken Fuzzing Tests
Path:	`./test/Fuzzing_Tests_TruReputationToken.js`
Detail:	Fuzzing Tests covering the TruReputationToken.sol Smart Contract
No of Test Cases:	27

	Description
01	Fuzz test of TruReputationToken Constructor with invalid executor address
02	Fuzz test of TruReputationToken changeBoardAddress with invalid arguments
03	Fuzz test of TruMintableToken mint with invalid arguments
04	Fuzz test of TruMintableToken finishMinting with invalid arguments
05	Fuzz test of ReleasableToken setTransferAgent with invalid arguments
06	Fuzz test of ReleasableToken setReleaseAgent with invalid arguments
07	Fuzz test of ReleasableToken releaseTokenTransfer with invalid arguments
08	Fuzz test of ReleasableToken transfer with invalid arguments
09	Fuzz test of ReleasableToken transferFrom with invalid arguments
10	Fuzz test of StandardToken approve with invalid arguments
11	Fuzz test of StandardToken allowance with invalid arguments
12	Fuzz test of StandardToken increaseApproval with invalid arguments
13	Fuzz test of StandardToken decreaseApproval with invalid arguments
14	Fuzz test of transferFrom of StandardToken with invalid arguments
15	Fuzz test of BasicToken balanceOf with invalid arguments
16	Fuzz test of transferOwnership of Ownable with invalid arguments
17	Fuzz test of UpgradeableToken setUpgradeAgent with invalid arguments
18	Fuzz test of UpgradeableToken setUpgradeMaster with invalid arguments
19	Fuzz test of UpgradeableToken upgrade with invalid arguments
20	Fuzz test of UpgradeableToken upgradeFrom with invalid arguments
21	Fuzz test of Ownable transferOwnership with invalid arguments
22	Fuzz test performing a large volume of transfer() transactions of 1 TRU between accounts
23	Fuzz test performing a large volume of transferFrom() transactions of 1 TRU between accounts
24	Fuzz test performing a large volume of transfer() transactions of 300,000,000 TRU between accounts
25	Fuzz test performing a large volume transferFrom() transactions of 300,000,000 TRU between accounts
26	Fuzz test of functions that receive no direct input
27	Fuzz test of structural send & sendTransaction functions

3.4.2 4.2. TruPreSale Fuzzing Tests

Name:	TruPreSale Fuzzing Tests
Path:	`./test/Fuzzing_Tests_TruPreSale.js`
Detail:	Fuzzing Tests covering the TruPreSale.sol Smart Contract
No of Test Cases:	13

	Description
01	Fuzz test of TruPreSale Constructor with invalid parameters
02	Fuzz test of TruPreSale updateWhiteList with invalid parameters
03	Fuzz test of TruPreSale buy with invalid parameters
04	Fuzz test of TruPreSale finalise with invalid parameters
05	Fuzz test of TruPreSale halt with invalid parameters
06	Fuzz test of TruPreSale hasEnded with invalid parameters
07	Fuzz test of TruPreSale send with invalid parameters
08	Fuzz test of TruPreSale sendTransaction with invalid parameters
09	Fuzz test of TruPreSale transferOwnership with invalid parameters
10	Fuzz test of TruPreSale unhalt with invalid parameters
11	Fuzz test of TruPreSale purchasedAmount with invalid parameters
12	Fuzz test of TruPreSale purchaserWhiteList with invalid parameters
13	Fuzz test of TruPreSale tokenAmount with invalid parameters

3.4.3 4.3. TruCrowdSale Fuzzing Tests

Name:	TruCrowdSale Fuzzing Tests
Path:	`./test/Fuzzing_Tests_TruCrowdSale.js`
Detail:	Fuzzing Tests covering the TruCrowdSale.sol Smart Contract
No of Test Cases:	13

	Description
01	Fuzz test of TruCrowdSale Constructor with invalid parameters
02	Fuzz test of TruCrowdSale updateWhiteList with invalid parameters
03	Fuzz test of TruCrowdSale buy with invalid parameters
04	Fuzz test of TruCrowdSale finalise with invalid parameters
05	Fuzz test of TruCrowdSale halt with invalid parameters
06	Fuzz test of TruCrowdSale hasEnded with invalid parameters
07	Fuzz test of TruCrowdSale send with invalid parameters
08	Fuzz test of TruCrowdSale sendTransaction with invalid parameters
09	Fuzz test of TruCrowdSale transferOwnership with invalid parameters
10	Fuzz test of TruCrowdSale unhalt with invalid parameters
11	Fuzz test of TruCrowdSale purchasedAmount with invalid parameters
12	Fuzz test of TruCrowdSale purchaserWhiteList with invalid parameters
13	Fuzz test of TruCrowdSale tokenAmount with invalid parameters

3.5 5. Edge Tests

To fully test edge cases, uncommon scenarios, or non conventional paths in code, Edge Tests have been written to ensure all paths in code are tested fully and for all possible results.

3.5.1 5.1 Supporting Edge Tests

Name:	Supporting Edge Tests
Path:	`./test/Edge_Tests_Supporting.js`
Detail:	Edges Tests covering edge case & failure testing on Supporting Smart Contracts & Libraries
No of Test Cases:	5

	Description
01	Should test all SafeMath functions
02	Should test transferFrom edge case
03	Should test all edge cases for TruSale
04	Should fail to set Migration Agent with
05	Should fail with invalid upgradeMaster Address in constructor

CHAPTER 4

Security and Code Auditing

The following section covers the Security & Code Auditing strategy and implementation for all Smart Contracts in the **Tru Reputation Token** project including supporting Libraries & Smart Contracts.

4.1 1. Strategy

The Security & Code Auditing Strategy for the **Tru Reputation Token** Project is as defined below:

- Due to the inherent financial risk of Cryptocurrency, and the evolving nature of threats and exploits within Solidity and EVM, standardised automated Security Auditing must be leveraged.
- Automated Security Audits are the be generated on each commit to the Repository.
- Auditing will include, as much as practicable, a scan against known vulnerabilities, exploits, and insecure coding patterns.
- Manual Security Audits will be performed by an external third party at least every 3 months after Production Code Release.
- Audits will be reviewed alongside Testing, Fuzz Testing and Code Coverage to ensure Best Practices and code security before being released to a public network.
- The **Tru Reputation Token** Project will not be released without the above items being met.

4.2 2. Auditing Tools

Given the evolving nature of Solidity and the EVM, the tools available for performing Security Auditing are not as fully featured as in other code environments. However, several projects are generally effective when combined with full Unit Testing and Fuzz Testing as part of a multi- layered Security Strategy including manual code reviews, manual Audits, Penetration Testing and bug reporting.

The following tools are used within the **Tru Reputation Token** Project:

Name	Description
EtherScan	EtherScan Verify Contract provides the capability to independently verify that the published source of a Contract matches the instance, ensuring a match at a bytecode level on the Contract and providing assurance to users of it.
CoverAlls	CoverAlls is used as part of the *Project Testing* Strategy to ensure Code Coverage of all utilised code and produces reports detailing the level and degree of code coverage against code execution branches.
Mythril	Mythril is security analysis tool for Ethereum Smart Contracts that uses concolic analysis to detect various types of issues. It can be used to both analyse the code and produce a 'ethermap' of the Smart Contract.
Oyente	Oyente is a tool for analysing Ethereum Smart Contracts and produces a report detailing whether well-known exploits can be achieved in the Contract scanned

Mythril and Oyente Audits are automatically performed on each commit to the Repository for each revision of the code, ensuring a continuous benchmark of Security Validation vs known exploits, and coding patterns that are known to open vulnerabilities.

Note: All Mythril and Oyente Audits can be viewed on the `./audits/` directory, with separate sub-directories for each, and separate sub-directories within them for each version Audited.

4.3 3. Public Instances

The following sub-sections list Public Instances of **Tru Reputation Token** Project Smart Contracts and Libraries, which version they are, whether they have been validated via EtherScan Verify Contract and a relevant EtherScan link.

4.3.1 3.1. Rinkeby TestNet Instances

The following Contract & Library Instances exist on the Rinkeby Test Network:

Name	TruAddress
Source File:	`/src/0.1.9/TruAddressFull.sol`
Type	Library
Version	0.1.9
Address	0xe3e9e6493c568a3e66577254a0931e4da95eda45
Source EtherScan Verified?	Yes

Name	TruReputationToken
Source File:	`/src/0.1.9/TruReputationTokenFull.sol`
Type	Smart Contract
Version	0.1.9
Address	0x3cc6363e5c791f804811e883b0af73cfba1b841d
Source EtherScan Verified?	Yes

Name	TruPreSale
Source File:	`/src/0.1.9/TruPreSaleFull.sol`
Type	Smart Contract
Version	0.1.9
Address	0x9a921ee90d0404c8f3f2eb974c8b3a415da142d5
Source EtherScan Verified?	Yes

Name	TruCrowdSale
Source File:	`/src/0.1.9/TruCrowdSaleFull.sol`
Type	Smart Contract
Version	0.1.9
Address	Not Yet Deployed
Source EtherScan Verified?	Not Yet Deployed

4.3.2 3.1. MainNet Instances

The following Contract & Library Instances exist on the Ethereum Ethereum Main Network:

CHAPTER 5

Supporting Scripts

The following scripts are used in the **Tru Reputation Token** project:

Name	Path	Description
audit.sh	`./scripts/audit.sh`	Automated Security Auditing script
coverage.sh	`./scripts/coverage.sh`	Automated Code Coverage Testing script
devnet.sh	`./scripts/devnet.sh`	Script for controlling Tru-DevNet Network
flattensrc.sh	`./scripts/flattensrc.sh`	Automated flatten source generation script
post-commit.sh	`./scripts/post-commit.sh`	Script for post-commit hook git activities
pre-commit.sh	`./scripts/pre-commit.sh`	Script for pre-commit hook git activities
'testnet.sh'_	`./scripts/testnet.sh`	Script for controlling TestNet TestRPC Network

5.1 audit.sh

Script Path: `./scripts/audit.sh`

Script Description:

> Script used to automate the generation of mythril and oyente audits that are placed in the `./audits/` directory.

Note: Audits are saved into sub-directories for each version of the project (e.g. `./audits/oyente/0.18/`) and the latest version is copied into the *current* directory (e.g. `./audits/oyente/current/`). These audits are performed against the flattened source for the *TruReputationToken*, *TruPreSale* and *TruCrowdSale* Smart Contracts, and the *TruAddress* Library.

Script Parameters:

Parameter	Detail	Usage Example
oyente	Used to generate oyente Audits into `./audits/oyente/`	`./scripts/audit.sh oyente`
mythril	Used to generate mythril Audits into `./audits/mythril/`	`./scripts/audit.sh mythril`
all	Used to generate both mythril and oyente Audits into `./audits`	`./scripts/audit.sh all`

Note: `./scripts/audit.sh all` is executed before each commit to the repository ensuring Security Audits for both mythril and oyente are generated for each version of the project.

Note: `./scripts/audit.sh all` is bound to the `npm run audit` script shortcut.

5.2 coverage.sh

Script Path: `./scripts/coverage.sh`

Script Description:

Script used to automate execution of solidity-coverage coverage testing of the **Tru Reputation Token** project. Results are placed in the `./coverage` directory as Istanbul HTML and are consumed by Coveralls

Script Parameters:

Parameter	Detail	Usage Example
start	Used start the Coverage TestRPC Network	`./scripts/coverage.sh start`
stop	Used stop the Coverage TestRPC Network	`./scripts/coverage.sh stop`
generate	Used perform generate Code Coverage Reporting	`./scripts/coverage.sh generate`

Note: The *coverage.sh* script is automatically executed by Travis CI upon each commit to the **Tru Reputation Token** repository.

Note: `./scripts/coverage.sh generate` is bound to the `npm run coverage` script shortcut.

5.3 devnet.sh

Script Path: `./scripts/coverage.sh`

Script Description:

Script used to setup, maintain and start the Tru DevNet private Geth Ethereum Network.

Script Parameters:

Parameter	Detail	Usage Example
start	Used start the Coverage Tru DevNet Private Geth Network	`./scripts/devnet.sh start`
stop	Used stop the Coverage Tru DevNet Private Geth Network	`./scripts/devnet.sh stop`
add	Used add a new address to the Tru DevNet Private Geth Network	`./scripts/devnet.sh add`
limit	Used to lower the CPU priority of the Geth instance running the Tru DevNet Network	`./scripts/devnet.sh limit`
restore	Used to restore the CPU priority of the Geth instance running the Tru DevNet Network	`./scripts/devnet.sh restore`
test	Used to execute all tests in `test` against the Tru DevNet Network	`./scripts/devnet.sh test`
migrate	Used to execute `truffle migrate` against the Tru DevNet Network	`./scripts/devnet.sh migrate`
console	Used to execute `truffle console` against the Tru DevNet Network	`./scripts/devnet.sh console`

5.4 flattensrc.sh

Script Path: `./scripts/flattensrc.sh`

Script Description:

> Script used to generate consolidated, flat Solidity source code for the *TruReputationToken*, *TruPreSale* and *TruCrowdSale* Smart Contracts, and the *TruAddress* Library that includes all dependencies into single files for each.

Script Parameters:

Parameter	Detail	Usage Example
flatten	Used to flatten all defined Smart Contracts and Libraries	`./scripts/flattensrc.sh flatten`
token	Used to flatten the `TruReputationToken.sol` Smart Contract	`./scripts/flattensrc.sh token`
presale	Used to flatten the `TruPreSale.sol` Smart Contract	`./scripts/flattensrc.sh presale`
crowdsale	Used to flatten the `TruCrowdSale.sol` Smart Contract	`./scripts/flattensrc.sh crowdsale`
address	Used to flatten the `TruAddress.sol` Library	`./scripts/flattensrc.sh address`

Note: Flattened source files are saved into sub-directories for each version of the project (e.g. `./src/0.1.8/TruAddressFull.sol`), and the latest version is copied into the *current* directory (e.g. `./src/current/TruAddressFull.sol`).

5.5 post-commit.sh

Script Path: `./scripts/post-commit.sh`

Script Description:

Script executed in the post-commit trigger in git by leveraging *post-commit* in the package.json. Used primarily to ensure that each version has a tag in the repository.

Script Parameters:

No Parameters

5.6 pre-commit.sh

Script Path: `./scripts/pre-commit.sh`

Script Description:

Script executed in the pre-commit trigger in git by leveraging *pre-commit* in the package.json. Used to ensure that patch version is incremented with each commit, documentation version is up to date and executes `./scripts/audit.sh all`

Script Parameters:

No Parameters

5.7 ./scripts/testnet.sh

Script Path: `./scripts/testnet.sh`

Script Description:

Script used to setup, maintain and start the TestNet TestRPC Ethereum Network.

Script Parameters:

Parameter	Detail	Usage Example
start	Starts the TestNet TestRPC Network	`./scripts/testnet.sh start`
stop	Stop the TestNet TestRPC Network	`./scripts/testnet.sh stop`
restart	Restarts the TestNet TestRPC Network	`./scripts/testnet.sh restart`
status	Shows the running status of the TestNet TestRPC Network	`./scripts/testnet.sh status`
test	Runs full Mocha test suite against the TestNet TestRPC Network	`./scripts/testnet.sh test`
fuzz	Runs full Mocha test suite against the TestNet TestRPC Network 250 times	`./scripts/testnet.sh fuzz`
migrate	executes `truffle migrate` against the TestNet TestRPC Network	`./scripts/testnet.sh migrate`
console	executes `truffle console` against the TestNet TestRPC Network	`./scripts/testnet.sh console`
quicktest	Runs full Mocha test suite against the TestNet TestRPC Network twice	`./scripts/testnet.sh quicktest`

CHAPTER 6

TruReputationToken

Title:	TruReputationToken
Description:	Smart Contract for the Tru Reputation Token
Author:	Ian Bray, Tru Ltd
Solidity Version:	0.4.18
Relative Path:	`./contracts/TruReputationToken.sol`
License:	Apache 2 License
Current Version:	0.1.12

6.1 1. Imports & Dependencies

The following imports and dependencies exist for the *TruReputationToken* Smart Contract:

Name	Description
SafeMath	Zeppelin Solidity Library to perform mathematics safely inside Solidity
TruAddress	Library of helper functions surrounding the Solidity Address type
TruMintableToken	Smart Contract derived from **MintableToken** by Zeppelin Solidity with additional functionality.
TruUpgradeableToken	Smart Contract derived from **UpgradeableToken** by Token Market with additional functionality.

6.2 2. Variables

The following variables exist for the *TruReputationToken* Smart Contract:

Variable	Type	Vis	Details
decimals	uint8	public	Constant variable for number of decimals token supports **Default:** *18*
name	string	public	Constant variable for public name of the token **Default** *Tru Reputation Token*
symbol	string	public	Constant variable for public symbol of the token **Default:** *TRU*
execBoard	address	public	Variable containing address of the Tru Ltd Executive Board **Default:** *0x0*

6.3 3. Enums

There are no enums for the *TruReputationToken* Smart Contract.

6.4 4. Events

The following events exist for the *TruReputationToken* Solidity Library:

Name	Description
BoardAddressChanged	Event to notify when the *execBoard* address changes

6.4.1 BoardAddressChanged

Event Name:	BoardAddressChanged
Description:	Event to notify when the *execBoard* address changes

Usage

The *BoardAddressChanged* event has the following usage syntax and arguments:

	Argument	Type	Indexed?	Details
1	oldAddress	address	Yes	Source wallet that the older tokens are sent from
2	newAddress	address	Yes	Address of the destination for upgraded tokens which is hardcoded to the *upgradeAgent* who sends them back to the originating address
3	executor	address	Yes	Address that executed the *BoardAddressChanged* event

Listing 1: **BoardAddressChanged Usage Example**

```
BoardAddressChanged(0x123456789abcdefghijklmnopqrstuvwxyz98765,
                    0x123456789abcdefghijklmnopqrstuvwxyz01234);
```

6.5 5. Mappings

There are no mappings for the *TruReputationToken* Smart Contract.

6.6 6. Modifiers

The following modifiers exist for the *TruReputationToken* Smart Contract:

Name	Description
onlyExecBoard	Modifier to check the Tru Advisory Board is executing this call

6.6.1 onlyExecBoard

Modifier Name:	onlyExecBoard
Description:	Modifier to check the Tru Advisory Board is executing this call

Code

The code for the *onlyExecBoard* modifier is as follows:

Listing 2: **onlyExecBoard Code**

```
modifier onlyExecBoard() {
    require(msg.sender == execBoard);
    _;
}
```

The *onlyExecBoard* function performs the following:

- Checks that the *msg.sender* matches the *execBoard* variable

6.7 7. Functions

The following functions exist for the *TruReputationToken* Smart Contract:

Name	Description
TruReputationToken Constructor	Constructor for the *TruReputationToken* Smart Contract
changeBoardAddress	Function to change the *execBoard* variable
canUpgrade	Override of *canUpgrade* function
setUpgradeMaster	Override of *setUpgradeMaster* function

6.7.1 TruReputationToken Constructor

Function Name:	TruReputationToken
Description:	Constructor for the *TruReputationToken* Smart Contract
Function Type:	Constructor
Function Visibility:	Public
Function Modifiers:	N/A
Return Type:	None
Return Details:	N/A

Code

The code for the *TruReputationToken Constructor* function is as follows:

Listing 3: **TruReputationToken Constructor Code**

```
function TruReputationToken() public TruUpgradeableToken(msg.sender) {
    execBoard = msg.sender;
    BoardAddressChanged(0x0, msg.sender);
}
```

The *TruReputationToken Constructor* function performs the following:

- Executes the TruUpgradeableToken constructor as part of its construction.
- Sets the initial *execBoard* variable to *msg.sender*
- Fires the *BoardAddressChanged* event

Usage

The *TruReputationToken Constructor* function has the following usage syntax and arguments:

	Argument	Type	Details
1	_upgradeMaster	address	Address to be set as the Upgrade Master

Listing 4: **TruReputationToken Constructor Usage Example**

```
TruReputationToken(0x123456789abcdefghijklmnopqrstuvwxyz98765);
```

6.7.2 changeBoardAddress

Function Name:	changeBoardAddress
Description:	Function to change the *execBoard* variable
Function Type:	N/A
Function Visibility:	Public
Function Modifiers:	*onlyExecBoard*
Return Type:	None
Return Details:	N/A

Code

The code for the *changeBoardAddress* function is as follows:

Listing 5: **changeBoardAddress Code**

```
function changeBoardAddress(address _newAddress) public onlyExecBoard {
    require(TruAddress.isValid(_newAddress) == true);
    require(_newAddress != execBoard);
    address oldAddress = execBoard;
    execBoard = _newAddress;
    BoardAddressChanged(oldAddress, _newAddress);
}
```

The *changeBoardAddress* function performs the following:

- Checks the *_newAddress* argument is a valid Ethereum Address. If not, it will throw
- Checks the *_newAddress* argument is not the same as the current *execBoard* variable. If it is, it will throw;
- Sets the *execBoard* variable to the *_newAddress* argument.
- Fires the *BoardAddressChanged* event

Usage

The *changeBoardAddress* function has the following usage syntax and arguments:

	Argument	Type	Details
1	_newAddress	address	Address to be set as the new Tru Advisory Board Address

Listing 6: **changeBoardAddress Usage Example**

```
changeBoardAddress(0x123456789abcdefghijklmnopqrstuvwxyz98765);
```

6.7.3 canUpgrade

Function Name:	canUpgrade
Description:	Override of *canUpgrade* function
Function Type:	Constant
Function Visibility:	Public
Function Modifiers:	None
Return Type:	bool
Return Details:	Returns true if the token is in an upgradeable state

Code

The code for the *canUpgrade* override function is as follows:

Listing 7: **canUpgrade Code**

```
function canUpgrade() public constant returns(bool) {
    return released && super.canUpgrade();
}
```

The *canUpgrade* function performs the following:

- If the *released* variable and super.canUpgrade() are true, returns true; otherwise returns false

Usage

The *canUpgrade* function has the following usage syntax:

Listing 8: **canUpgrade Usage Example**

```
canUpgrade();
```

6.7.4 setUpgradeMaster

Function Name:	setUpgradeMaster
Description:	Override of *setUpgradeMaster* function
Function Type:	N/A
Function Visibility:	Public
Function Modifiers:	*onlyOwner*
Return Type:	bool
Return Details:	Returns true if the token is in an upgradeable state

Code

The code for the *setUpgradeMaster* override function is as follows:

Listing 9: **setUpgradeMaster Code**

```
function setUpgradeMaster(address master) public onlyOwner {
    super.setUpgradeMaster(master);
}
```

The *setUpgradeMaster* function performs the following:

- Executes the *setUpgradeMaster* function with the *onlyOwner* modifier.

Usage

The *setUpgradeMaster* function has the following usage syntax and arguments:

	Argument	Type	Details
1	_master	address	Address to be set as the new Upgrade Master Contract

Listing 10: **setUpgradeMaster Usage Example**

```
setUpgradeMaster(0x123456789abcdefghijklmnopqrstuvwxyz98765);
```

CHAPTER 7

TruSale

The *TruSale* Smart Contract acts a parent class for the *TruPreSale* and *TruCrowdSale* contracts and contains all logic common to both.

Title:	TruSale
Description:	Parent Smart Contract for all *TruReputationToken* Token Sales
Author:	Ian Bray, Tru Ltd
Solidity Version:	^0.4.18
Relative Path:	`./contracts/TruSale.sol`
License:	Apache 2 License
Current Version:	0.1.12

7.1 1. Imports & Dependencies

The following imports and dependencies exist for the *TruSale* Smart Contract:

Name	Description
Haltable	Modified Token Market Smart Contract that provides a capability to halt a contract.
Ownable	Zeppelin Solidity Smart Contract that provides ownership capabilities to a contract.
SafeMath	Zeppelin Solidity Library to perform mathematics safely inside Solidity
TruAddress	Library of helper functions surrounding the Solidity Address type
TruReputationToken	Smart Contract for the Tru Reputation Token

7.2 2. Variables

The following variables exist for the *TruSale* Smart Contract:

Variable	Type	Vis	Details
truToken	TruReputation-Token	public	Variable for the token being sold in Sale
saleStartTime	uint256	public	Start timestamp of the Sale
saleEndTime	uint256	public	End timestamp of the Sale
purchaser-Count	uint	public	Number of sale purchasers so far **Default:** *0*
multiSigWallet	address	public	Sale wallet address
BASE_RATE	uint256	public	Constant variable of post sale TRU to ETH rate **Default:** *1000*
PRE-SALE_RATE	uint256	public	Constant variable of Pre-Sale TRU to ETH rate **Default:** *1250* - 25% Bonus
SALE_RATE	uint256	public	Constant variable of CrowdSale TRU to ETH rate **Default:** *1125* - 12.5% Bonus
MIN_AMOUNT	uint256	public	Minimum Amount of ETH for an address to participate in Sale **Default:** *1 * 10^18*
MAX_AMOUNT	uint256	public	Maximum ETH buy Amount for a non-Whitelist address **Default:** *20 * 10^18*
weiRaised	uint256	public	Amount raised during Sale in Wei
cap	uint256	public	Cap of the Sale- value set during construction
isCompleted	bool	public	Whether the Sale is complete
isPreSale	bool	public	Whether the Sale is a Pre-Sale
isCrowdSale	bool	public	Whether the Sale is a CrowdSale
soldTokens	uint256	public	Amount of TRU during Sale

7.3 3. Enums

There are no enums for the *TruSale* Smart Contract.

7.4 4. Events

The following events exist for the *TruSale* Smart Contract:

Name	Description
TokenPurchased	Event to notify when a token purchase occurs
WhiteListUpdated	Event to notify when the *purchaseWhiteList* is updated
EndChanged	Event to notify when the *saleEndTime* changes
Completed	Event to notify when the Sale completes

7.4.1 TokenPurchased

Event Name:	TokenPurchased
Description:	EEvent to notify when a token purchase occurs

Usage

The *TokenPurchased* event has the following usage syntax and arguments:

	Argument	**Type**	**Indexed?**	**Details**
1	purchaser	address	Yes	Address being updated on the Whitelist
2	recipient	address	No	Status of the address on the Whitelist
3	weiValue	uint256	No	Amount of ETH spent (in Wei)
4	tokenAmount	uint256	No	Amount of tokens purchased (in smallest decimal)

Listing 1: **TokenPurchased Usage Example**

```
TokenPurchased(0x123456789abcdefghijklmnopqrstuvwxyz98765,
               0x123456789abcdefghijklmnopqrstuvwxyz98765,
               100000000000000000,
               1250000000000000000000);
```

7.4.2 WhiteListUpdated

Event Name:	WhiteListUpdated
Description:	Event to notify when the *purchaseWhiteList* is updated

Usage

The *WhiteListUpdated* event has the following usage syntax and arguments:

	Argument	**Type**	**Indexed?**	**Details**
1	purchaserAddress	address	Yes	Address being updated on the Whitelist
2	whitelistStatus	address	No	Status of the address on the Whitelist
3	executor	address	Yes	Address that executed the *WhiteListUpdated* event

Listing 2: **WhiteListUpdated Usage Example**

```
WhiteListUpdated(0x123456789abcdefghijklmnopqrstuvwxyz98765,
                 true,
                 0x12acd9ef9abcdefghijklmnopqrstuvwxyzghy74);
```

7.4.3 EndChanged

Event Name:	EndChanged
Description:	Event to notify when the *purchaseWhiteList* is updated

Usage

The *EndChanged* event has the following usage syntax and arguments:

	Argument	Type	Indexed?	Details
1	oldEnd	uint256	No	Previous *saleEndTime* timestamp
2	newEnd	uint256	No	Updated *saleEndTime* timestamp
3	executor	address	Yes	Address that executed the *EndChanged* event

Listing 3: **EndChanged Usage Example**

```
EndChanged(1511930475,
           1512016874,
           0x123456789abcdefghijklmnopqrstuvwxyz98765);
```

7.4.4 Completed

Event Name:	Completed
Description:	Event to notify when the Sale completes

Usage

The *Completed* event has the following usage syntax:

	Argument	Type	Indexed?	Details
1	executor	address	Yes	Address that executed the *Completed* event

Listing 4: **Completed Usage Example**

```
Completed(0x123456789abcdefghijklmnopqrstuvwxyz98765);
```

7.5 5. Mappings

The following mappings exist for the *TruSale* Smart Contract:

Name	Mapping Type	Description
purchasedAmount	address => uint256	Mapping of purchased amount in ETH to buying address
tokenAmount	address => uint256	Mapping of purchased amount of TRU to buying address
purchaserWhiteList	address => bool	Mapping of Whitelisted address to their Whitelist status

7.6 6. Modifiers

The following modifiers exist for the *TruSale* Smart Contract:

Name	Description
onlyTokenOwner	Modifier to check if transaction sender is the owner of the Token contract

7.6.1 onlyTokenOwner

Modifier Name:	onlyTokenOwner
Description:	Modifier to check if transaction sender is the owner of the Token contract

Code

The code for the *onlyTokenOwner* modifier is as follows:

Listing 5: **onlyTokenOwner Code**

```
modifier onlyTokenOwner(address _tokenOwner) {
    require(msg.sender == _tokenOwner);
    _;
}
```

The *onlyTokenOwner* function performs the following:

- Checks that the *msg.sender* matches the supplied *_tokenOwner* variable. If not, it will throw.

7.7 7. Functions

The following functions exist for the *TruSale* Smart Contract:

Name	Description
TruSale Constructor	Constructor for the *TruSale* Smart Contract
buy	Function for buying tokens from the Sale
updateWhitelist	Function to add or disable a purchaser from AML Whitelist
changeEndTime	Function to change the end time of the Sale
hasEnded	Function to check whether the Sale has ended
checkSaleValid	Internal function to validate that the Sale is valid
validatePurchase	Internal function to validate the purchase of TRU Tokens
forwardFunds	Internal function to forward all raised funds to the Sale Wallet
createSale	Internal function used to encapsulate more complex constructor logic
buyTokens	Private function execute purchase of TRU Tokens

7.7.1 TruSale Constructor

Function Name:	TruSale
Description:	Constructor for the *TruSale* Smart Contract
Function Type:	Constructor
Function Visibility:	Public
Function Modifiers:	N/A
Return Type:	None
Return Details:	N/A

Code

The code for the *TruSale Constructor* function is as follows:

Listing 6: **TruSale Constructor Code**

```
function TruSale(uint256 _startTime,
                 uint256 _endTime,
                 address _token,
                 address _saleWallet) public {

    require(TruAddress.isValid(_token) == true);

    TruReputationToken tToken = TruReputationToken(_token);
    address tokenOwner = tToken.owner();

    createSale(_startTime, _endTime, _token, _saleWallet, tokenOwner);
}
```

The *TruSale Constructor* function performs the following:

- Checks the *_token* argument is a valid Ethereum address.
- Gets the owner of the *_token* TruReputationToken object
- Executes the *createSale* function with the *tokenOwner* variable as an argument.

Usage

The *TruSale Constructor* function has the following usage syntax and arguments:

	Argument	Type	Details
1	_startTime	uint256	Sale start timestamp
2	_endTime	uint256	Sale end timestamp
3	_token	address	Address of TruReputationToken Contract
4	_saleWallet	address	Address of sale wallet

Listing 7: **TruSale Constructor Usage Example**

```
TruSale(1511930475,
        1512016874,
        0x123456789abcdefghijklmnopqrstuvwxyz98765,
        0x987654321abcdefghijklmnopqrstuvwxyz12345);
```

7.7.2 buy

Function Name:	buy
Description:	Function for buying tokens from the Sale
Function Type:	N/A
Function Visibility:	Public payable
Function Modifiers:	*stopInEmergency*
Return Type:	N/A
Return Details:	N/A

Code

The code for the *buy* function is as follows:

Listing 8: **buy Code**

```
function buy() public payable stopInEmergency {
    // Check that the Sale is still open and the Cap has not been reached
    require(checkSaleValid());

    validatePurchase(msg.sender);
}
```

Note: the *buy* function is a Solidity payable functino- as such, ETH is sent to the function to allow the purchase of tokens during a sale. This function can be halted via the stop-in-emergency modifier as part of the *Haltable* characteristics of this Contract.

The *buy* function performs the following:

- The modifier *stopInEmergency* checks that the Sale has not been halted. If it has, it will throw.
- Checks the *checkSaleValid* function returns true. If not, it will throw.
- executes the *validatePurchase* function.

Usage

The *buy* function has the following usage syntax:

Listing 9: **buy Usage Example**

```
buy({value: 1000000000000000000});
```

7.7.3 updateWhitelist

Function Name:	updateWhitelist
Description:	Function to add or disable a purchaser from AML Whitelist
Function Type:	N/A
Function Visibility:	Public
Function Modifiers:	*onlyOwner*
Return Type:	None
Return Details:	N/A

Code

The code for the *updateWhitelist* function is as follows:

Listing 10: **updateWhitelist Code**

```
function updateWhitelist(address _purchaser, uint _status) public onlyOwner {
    require(TruAddress.isValid(_purchaser) == true);
    bool boolStatus = false;
    if (_status == 0) {
        boolStatus = false;
    } else if (_status == 1) {
        boolStatus = true;
    } else {
        revert();
    }

    WhiteListUpdated(_purchaser, boolStatus);
    purchaserWhiteList[_purchaser] = boolStatus;
}
```

Note: The *updateWhitelist* function uses uint for the *status* argument because fuzz testing found that bool arguments on public functions in Solidity could be interpreted as true when supplied with a random string.

In the interest of type safety and defensive development this was set to uint with **0** being **false** and **1** being **true**, all other values are ignored.

Be very careful using bool on public functions in Solidity.

The *updateWhitelist* function performs the following:

- Validates the *_purchaser* argument is a valid Ethereum address.
- Checks the *_status* argument is either 0 or 1. If 0, sets *boolStatus* to false, if 1, sets *boolStatus* to true. If else, it will throw.
- Fires the *WhiteListUpdated* event
- Sets the *_purchaser* to the *boolStatus* on the *purchaseWhiteList*

Usage

The *updateWhitelist* function has the following usage syntax and arguments:

	Argument	Type | Details
1	_purchaser	uint256 | Address of the purchaser to add or update on the Whitelist
2	_status	uint | Status on the Whitelist- 0 for disabled, 1 for enabled

Listing 11: **updateWhitelist Usage Example**

```
updateWhitelist(0x987654321abcdefghijklmnopqrstuvwxyz12345, 1);
```

7.7.4 changeEndTime

Function Name:	changeEndTime
Description:	Function to change the end time of the Sale
Function Type:	N/A
Function Visibility:	Public
Function Modifiers:	*onlyOwner*
Return Type:	None
Return Details:	N/A

Code

The code for the *changeEndTime* function is as follows:

Listing 12: **changeEndTime Code**

```
function changeEndTime(uint256 _endTime) public onlyOwner {

    // _endTime must be greater than or equal to saleStartTime
    require(_endTime >= saleStartTime);

    // Fire Event for time Change
    EndChanged(saleEndTime, _endTime);

    // Change the Sale End Time
    saleEndTime = _endTime;
}
```

Note: The *changeEndTime* function has been included to allow a Sale's end time to be altered after the start. This is addressed in *SALREQ 012* and behaves in the following way:

1. If the End Time is moved before the current block timestamp, it will automatically close the Sale fully and finally.

2. If the End Time is moved beyond the current end time, it will extend the time remaining in the Sale. This is useful if issues with the network are encountered and should only be used will full communication to purchasers prior to the change.

The *changeEndTime* function performs the following:

- Checks the *_endTime* argument is equal to or greater than the *saleStartTime* variable. If not, it will throw.
- Fire the *EndChanged* event.
- Set the *saleEndTime* variable to the *_endTime* argument.

Usage

The *changeEndTime* function has the following usage syntax and arguments:

	Argument	Type | Details
1	_endTime	uint256 | New end timestamp for Sale

Listing 13: **changeEndTime Usage Example**

```
changeEndTime(1511930475);
```

7.7.5 hasEnded

Function Name:	hasEnded
Description:	Function to check whether the Sale has ended
Function Type:	Constant
Function Visibility:	Public
Function Modifiers:	N/A
Return Type:	bool
Return Details:	Returns true if the Sale has ended; false if it has not

Code

The code for the *hasEnded* function is as follows:

Listing 14: **hasEnded Code**

```
function hasEnded() public constant returns (bool) {
    bool isCapHit = weiRaised >= cap;
    bool isExpired = now > saleEndTime;
    return isExpired || isCapHit;
}
```

The *hasEnded* function performs the following:

- Checks that the *weiRaised* variable is less than the *cap* variable.
- Checks that the current block timestamp is less than the *saleEndTime* timestamp
- If either of the previous checks are true, the Sale has ended. Otherwise the Sale has not ended.

Usage

The *hasEnded* function has the following usage syntax:

Listing 15: **hasEnded Usage Example**

```
hasEnded();
```

7.7.6 checkSaleValid

Function Name:	checkSaleValid
Description:	Internal function to validate that the Sale is valid
Function Type:	Constant
Function Visibility:	Internal
Function Modifiers:	N/A
Return Type:	bool
Return Details:	Returns true if the Sale is still open; false if it is not

Code

The code for the *checkSaleValid* function is as follows:

Listing 16: **checkSaleValid Code**

```
function checkSaleValid() internal constant returns (bool) {
    bool afterStart = now >= saleStartTime;
    bool beforeEnd = now <= saleEndTime;
    bool capNotHit = weiRaised.add(msg.value) <= cap;
    return afterStart && beforeEnd && capNotHit;
}
```

The *checkSaleValid* function performs the following:

- Checks the Sale has started. If it has not, will return false.
- Checks the Sale has not ended. If it has, will return false.
- Checks the cap has not been hit, if it has, will return false.

Usage

The *checkSaleValid* function has the following usage syntax:

Listing 17: **checkSaleValid Usage Example**

```
checkSaleValid();
```

7.7.7 validatePurchase

Function Name:	validatePurchase
Description:	Internal function to validate the purchase of TRU Tokens
Function Type:	N/A
Function Visibility:	Internal
Function Modifiers:	*stopInEmergency*
Return Type:	N/A
Return Details:	N/A

Code

The code for the *validatePurchase* function is as follows:

```
function validatePurchase(address _purchaser) internal stopInEmergency {

    // _purchaser must be valid
    require(TruAddress.isValid(_purchaser) == true);

    // Value must be greater than 0
    require(msg.value > 0);

    buyTokens(_purchaser);
}
```

Note: The *validatePurchase* function acts as the both a pre-validation step for a purchase, and a point at which the Sale can be halted as per the *Haltable* Smart Contract.

The *validatePurchase* function performs the following:

- Validates that the *_purchaser* argument is a valid Ethereum Address.
- Validates that the *msg.value* is greater than 0
- Executes the *buyTokens* function.

Usage

The *validatePurchase* function has the following usage syntax:

Listing 18: **validatePurchase Usage Example**

```
validatePurchase(0x987654321abcdefghijklmnopqrstuvwxyz12345);
```

7.7.8 forwardFunds

Function Name:	forwardFunds
Description:	Internal function to forward all raised funds to the Sale Wallet
Function Type:	N/A
Function Visibility:	Internal
Function Modifiers:	N/A
Return Type:	N/A
Return Details:	N/A

Code

The code for the *forwardFunds* function is as follows:

```
function forwardFunds() internal {
    multiSigWallet.transfer(msg.value);
}
```

The *forwardFunds* function performs the following:

- Transfers any new funds away from the *TruSale* Smart Contract, to the Sale Wallet reflected in the *multiSigWallet* variable.

Usage

The *forwardFunds* function has the following usage syntax:

Listing 19: **forwardFunds Usage Example**

```
forwardFunds();
```

7.7.9 createSale

Function Name:	createSale
Description:	Internal function used to encapsulate more complex constructor logic
Function Type:	N/A
Function Visibility:	Internal
Function Modifiers:	*onlyTokenOwner*
Return Type:	N/A
Return Details:	N/A

Code

The code for the *createSale* function is as follows:

Listing 20: **createSale Code**

```
function createSale(
    uint256 _startTime,
    uint256 _endTime,
    address _token,
    address _saleWallet,
    address _tokenOwner)
internal onlyTokenOwner(_tokenOwner) {
    // _startTime must be greater than or equal to now
    require(now <= _startTime);

    // _endTime must be greater than or equal to _startTime
    require(_endTime >= _startTime);

    // _salletWallet must be valid
    require(TruAddress.isValid(_saleWallet) == true);

    truToken = TruReputationToken(_token);
    multiSigWallet = _saleWallet;
    saleStartTime = _startTime;
    saleEndTime = _endTime;
}
```

Note: The *createSale* argument uses the *onlyTokenOwner* modifier to ensure that no instance of the *TruSale* can be created for *TruReputationToken* unless they are the owner of that contract. If that modifier is passed, the rest of the logic is processed to construct the *TruSale* instance.

The *createSale* function performs the following:

- Ensures the *_startTime* timestamp argument is greater than the latest block timestamp.
- Ensures the *_endTime* timestamp argument is greater than the *_startTime* timestamp argument.
- Ensures the *_saleWallet* argument is a valid Ethereum Address.
- Sets the truToken variable to the instance of TruReputationToken from the _token argument.
- Sets the *multiSigWallet* variable to the *_saleWallet* argument.
- Sets the *saleStartTime* variable to the *_startTime* argument.
- Sets the *saleEndTime* variable to the *_endTime* argument.

Usage

The *createSale* function has the following usage syntax:

Listing 21: **createSale Usage Example**

```
createSale(1511930475,
           1512016874,
           0x123456789abcdefghijklmnopqrstuvwxyz98765,,
           0x465328375xyzacefgijklmnopqrstuvwxyz66712,
           0xa57htuju9abcdefghijehtitthtjiohjtoi02447);
```

7.7.10 buyTokens

Function Name:	buyTokens
Description:	Private function execute purchase of TRU Tokens
Function Type:	N/A
Function Visibility:	Private
Function Modifiers:	N/A
Return Type:	N/A
Return Details:	N/A

Code

The code for the *buyTokens* function is as follows:

Listing 22: **buyTokens Code**

```
function buyTokens(address _purchaser) private {
    uint256 weiTotal = msg.value;

    // If the Total wei is less than the minimum purchase, reject
    require(weiTotal >= MIN_AMOUNT);

    // If the Total wei is greater than the maximum stake, purchasers must be on the␣
↪whitelist
    if (weiTotal > MAX_AMOUNT) {
        require(purchaserWhiteList[msg.sender]);
    }

    // Prevention to stop circumvention of Maximum Amount without being on the␣
↪Whitelist
    if (purchasedAmount[msg.sender] != 0 && !purchaserWhiteList[msg.sender]) {
        uint256 totalPurchased = purchasedAmount[msg.sender];
        totalPurchased = totalPurchased.add(weiTotal);
        require(totalPurchased < MAX_AMOUNT);
    }

    uint256 tokenRate = BASE_RATE;

    if (isPreSale) {
        tokenRate = PRESALE_RATE;
    }
    if (isCrowdSale) {
        tokenRate = SALE_RATE;
    }

    // Multiply Wei x Rate to get Number of Tokens to create (as a 10^18 subunit)
    uint256 noOfTokens = weiTotal.mul(tokenRate);

    // Add the wei to the running total
    weiRaised = weiRaised.add(weiTotal);

    // If the purchaser address has not purchased already, add them to the list
    if (purchasedAmount[msg.sender] == 0) {
        purchaserCount++;
    }
```

(continues on next page)

(continued from previous page)

```
    soldTokens = soldTokens.add(noOfTokens);

    purchasedAmount[msg.sender] = purchasedAmount[msg.sender].add(msg.value);
    tokenAmount[msg.sender] = tokenAmount[msg.sender].add(noOfTokens);

    // Mint the Tokens to the Purchaser
    truToken.mint(_purchaser, noOfTokens);
    TokenPurchased(msg.sender,
    _purchaser,
    weiTotal,
    noOfTokens);
    forwardFunds();
}
```

The *buyTokens* function performs the following:

- Checks that the sent amount (*msg.value*) is equal to or greater than the *MIN_AMOUNT* variable. If it is not, it will throw.
- Checks if the sent amount (*msg.value*) is greater than the *MAX_AMOUNT* variable. If it is, it will perform a further check to see if the sender is on the Whitelist- if they are, it will proceed, if not it will throw. If the amount is less than or equal to the *MAX_AMOUNT* variable, it will proceed.
- Checks that the cumulative total of this purchase, and any prior purchases do not exceed the *MAX_AMOUNT* variable if the purchaser is not on the Whitelist. If it is, it will throw.
- Sets the Sale Rate to the default of the *BASE_RATE* variable.
- If the *isPreSale* variable is true sets the Sale Rate to *PRESALE_RATE* variable.
- If the *isCrowdSale* variable is true sets the Sale Rate to *SALE_RATE* variable.
- Calculates the number of tokens purchased.
- Increments the *purchaserCount* variable if this is the first purchase from this address.
- Adds the calculated token count to the *soldTokens* variable.
- Adds the *msg.value* to the *purchasedAmount* mapping for the purchaser.
- Adds the token amount to the *tokenAmount* mapping for the purchaser.
- Mints the token amount to the purchaser's address.
- Fires the *TokenPurchased* event.
- Executes the *forwardFunds* function.

Usage

The *buyTokens* function has the following usage syntax:

Listing 23: **buyTokens Usage Example**

```
buyTokens(0xa57htuju9abcdefghijehtitthtjiohjtoi02447);
```

CHAPTER 8

TruPreSale

The *TruPreSale* Smart Contract acts a child class to the *TruSale* and is used for the main CrowdSale of the *TruReputationToken*.

Title:	TruPreSale
Description:	Smart Contract for the Pre-Sale of the *TruReputationToken*.
Author:	Ian Bray, Tru Ltd
Solidity Version:	0.4.18
Relative Path:	`./contracts/TruPreSale.sol`
License:	Apache 2 License
Current Version:	0.1.12

8.1 1. Imports & Dependencies

The following imports and dependencies exist for the *TruPreSale* Smart Contract:

Name	Description
TruSale	Parent Smart Contract for all *TruReputationToken* Token Sales
TruReputationToken	Smart Contract for the Tru Reputation Token
SafeMath	Zeppelin Solidity Library to perform mathematics safely inside Solidity

8.2 2. Variables

The following variables exist for the *TruPreSale* Smart Contract:

Variable	Type	Vis	Details
PRESALE_CAP	uint256	public	Variable for the Pre-Sale cap **Default:** *8000 * 10^18*

8.3 3. Enums

There are no enums for the *TruPreSale* Smart Contract.

8.4 4. Events

There are no events for the *TruPreSale* Smart Contract.

8.5 5. Mappings

There are no mappings for the *TruPreSale* Smart Contract.

8.6 6. Modifiers

There are no modifiers for the *TruPreSale* Smart Contract.

8.7 7. Functions

The following functions exist for the *TruPreSale* Smart Contract:

Name	Description
TruPreSale Constructor	Constructor for the *TruPreSale* Smart Contract
finalise	Function to finalise Pre-Sale.
completion	Internal function to complete Pre-Sale.

8.7.1 TruPreSale Constructor

Function Name:	TruPreSale
Description:	Constructor for the *TruPreSale* Smart Contract
Function Type:	Constructor
Function Visibility:	Public
Function Modifiers:	N/A
Return Type:	None
Return Details:	N/A

Code

The code for the *TruPreSale Constructor* function is as follows:

Listing 1: **TruPreSale Constructor Code**

```
unction TruPreSale(
    uint256 _startTime,
    uint256 _endTime,
    address _token,
    address _saleWallet) public TruSale(_startTime, _endTime, _token, _saleWallet)
{
        isPreSale = true;
        isCrowdSale = false;
        cap = PRESALE_CAP;
}
```

The *TruPreSale Constructor* function performs the following:

- Executes the super *TruSale Constructor* function.
- Sets the *isPreSale* variable to **true**.
- Sets the *isCrowdSale* variable to **false**.
- Set the *cap* variable to equal the *PRESALE_CAP* variable value.

Usage

The *TruPreSale Constructor* function has the following usage syntax and arguments:

	Argument	Type	Details
1	_startTime	uint256	Sale start timestamp
2	_endTime	uint256	Sale end timestamp
3	_token	address	Address of TruReputationToken Contract
4	_saleWallet	address	Address of *TruPreSale* wallet

Listing 2: **TruPreSale Constructor Usage Example**

```
 TruPreSale(1511930475,
              1512016874,
              0x123456789abcdefghijklmnopqrstuvwxyz98765,
              0x987654321abcdefghijklmnopqrstuvwxyz12345);
```

8.7.2 finalise

Function Name:	finalise
Description:	Function to finalise Pre-Sale.
Function Type:	N/A
Function Visibility:	Public
Function Modifiers:	ref:*ownable-only-owner*
Return Type:	None
Return Details:	N/A

Code

The code for the *finalise* function is as follows:

Listing 3: **finalise Code**

```
function finalise() public onlyOwner {
    require(!isCompleted);
    require(hasEnded());

    completion();
    Completed();

    isCompleted = true;
}
```

The *finalise* function performs the following:

- Checks that the *isCompleted* variable is set to false. If not, it will throw.
- Checks the *hasEnded* function returns true. If not, it will throw.
- Executes the *completion* function.
- Fires the *Completed* event.
- Sets *isCompleted* variable to true.

Usage

The *finalise* function has the following usage syntax:

Listing 4: **finalise Usage Example**

```
finalise();
```

8.7.3 completion

Function Name:	completion
Description:	Internal function to complete Pre-Sale.
Function Type:	N/A
Function Visibility:	Internal
Function Modifiers:	N/A
Return Type:	None
Return Details:	N/A

Code

The code for the *completion* function is as follows:

Listing 5: **completion Code**

```
function completion() internal {

    // Double sold pool to allocate to Tru Resource Pools
    uint256 poolTokens = truToken.totalSupply();

```

(continues on next page)

(continued from previous page)

```
    // Issue poolTokens to multisig wallet
    truToken.mint(multiSigWallet, poolTokens);
    truToken.finishMinting(true, false);
    truToken.transferOwnership(msg.sender);
}
```

The *completion* function performs the following:

- Calculates the number of tokens sold in this Pre-Sale and mints the same amount again into the *multiSigWallet* Sale wallet for use by Tru Ltd as per the Tru Reputation Protocol Whitepaper.
- Executes the *finishMinting* function to end Pre-Sale minting and await CrowdSale minting
- Transfers ownership of the *TruReputationToken* back to the executing account now the Pre-Sale is complete.

Usage

The *completion* function has the following usage syntax:

Listing 6: **completion Usage Example**

```
completion();
```

CHAPTER 9

TruCrowdSale

The *TruCrowdSale* Smart Contract acts a child class to the *TruSale* and is used for the main CrowdSale of the *TruReputationToken*.

Title:	TruCrowdSale
Description:	Smart Contract for the CrowdSale of the *TruReputationToken*.
Author:	Ian Bray, Tru Ltd
Solidity Version:	0.4.18
Relative Path:	`./contracts/TruCrowdSale.sol`
License:	Apache 2 License
Current Version:	0.1.12

9.1 1. Imports & Dependencies

The following imports and dependencies exist for the *TruCrowdSale* Smart Contract:

Name	Description
TruSale	Parent Smart Contract for all *TruReputationToken* Token Sales
TruReputationToken	Smart Contract for the Tru Reputation Token
SafeMath	Zeppelin Solidity Library to perform mathematics safely inside Solidity

9.2 2. Variables

The following variables exist for the *TruCrowdSale* Smart Contract:

Variable	Type	Vis	Details
TOTAL_CAP	uint256	public	Variable for the Total cap for the Crowdsale & Pre-Sale
existingSupply	uint256	private	Variable containing the existing *TruReputationToken* supply.

9.3 3. Enums

There are no enums for the *TruCrowdSale* Smart Contract.

9.4 4. Events

There are no events for the *TruCrowdSale* Smart Contract.

9.5 5. Mappings

There are no mappings for the *TruCrowdSale* Smart Contract.

9.6 6. Modifiers

There are no modifiers for the *TruCrowdSale* Smart Contract.

9.7 7. Functions

The following functions exist for the *TruCrowdSale* Smart Contract:

Name	Description
TruCrowdSale Constructor	Constructor for the *TruCrowdSale* Smart Contract
finalise	Function to finalise CrowdSale.
completion	Internal function to complete CrowdSale.

9.7.1 TruCrowdSale Constructor

Function Name:	TruCrowdSale
Description:	Constructor for the *TruCrowdSale* Smart Contract
Function Type:	Constructor
Function Visibility:	Public
Function Modifiers:	N/A
Return Type:	None
Return Details:	N/A

Code

The code for the *TruCrowdSale Constructor* function is as follows:

Listing 1: **TruCrowdSale Constructor Code**

```
function TruCrowdSale(
    uint256 _startTime,
    uint256 _endTime,
    address _token,
    address _saleWallet,
    uint256 _currentSupply,
    uint256 _currentRaise) public TruSale(_startTime, _endTime, _token, _saleWallet)
    {
        isPreSale = false;
        isCrowdSale = true;
        uint256 remainingCap = TOTAL_CAP.sub(_currentRaise);
        cap = remainingCap;
        existingSupply = _currentSupply;
}
```

The *TruCrowdSale Constructor* function performs the following:

- Executes the super *TruSale Constructor* function.
- Sets the *isPreSale* variable to **false**.
- Sets the *isCrowdSale* variable to **true**.
- Calculates the *cap* variable by removing the *_currentRaise* argument from the *TOTAL_CAP* variable.
- Sets *existingSupply* variable to the *_currentSupply* argument.

Usage

The *TruCrowdSale Constructor* function has the following usage syntax and arguments:

	Argument	**Type**	**Details**
1	_startTime	uint256	Sale start timestamp
2	_endTime	uint256	Sale end timestamp
3	_token	address	Address of TruReputationToken Contract
4	_saleWallet	address	Address of *TruCrowdSale* wallet
5	_currentSupply	uint256	Current amount of *TruReputationToken* tokens issued.
6	_currentRaise	uint256	Current amount of ETH raised in the *TruPreSale*

Listing 2: **TruCrowdSale Constructor Usage Example**

```
TruCrowdSale(1511930475,
             1512016874,
             0x123456789abcdefghijklmnopqrstuvwxyz98765,
             0x987654321abcdefghijklmnopqrstuvwxyz12345,
             8000000000000000000000,
             100000000000000000000000000);
```

9.7.2 finalise

Function Name:	finalise
Description:	Function to finalise CrowdSale.
Function Type:	N/A
Function Visibility:	Public
Function Modifiers:	ref:*ownable-only-owner*
Return Type:	None
Return Details:	N/A

Code

The code for the *finalise* function is as follows:

Listing 3: **finalise Code**

```
function finalise() public onlyOwner {
    require(!isCompleted);
    require(hasEnded());

    completion();
    Completed();

    isCompleted = true;
}
```

The *finalise* function performs the following:

- Checks that the *isCompleted* variable is set to false. If not, it will throw.
- Checks the *hasEnded* function returns true. If not, it will throw.
- Executes the *completion* function.
- Fires the *Completed* event.
- Sets *isCompleted* variable to true.

Usage

The *finalise* function has the following usage syntax:

Listing 4: **finalise Usage Example**

```
finalise();
```

9.7.3 completion

Function Name:	completion
Description:	Internal function to complete CrowdSale.
Function Type:	N/A
Function Visibility:	Internal
Function Modifiers:	N/A
Return Type:	None
Return Details:	N/A

Code

The code for the *completion* function is as follows:

Listing 5: **completion Code**

```
function completion() internal {

    // Double sold pool to allocate to Tru Resource Pools
    uint256 poolTokens = truToken.totalSupply();
    poolTokens = poolTokens.sub(existingSupply);

    // Issue poolTokens to multisig wallet
    truToken.mint(multiSigWallet, poolTokens);
    truToken.finishMinting(false, true);
    truToken.transferOwnership(msg.sender);
    truToken.releaseTokenTransfer();
}
```

The *completion* function performs the following:

- Calculates the number of tokens sold in this CrowdSale and mints the same amount again into the *multiSigWallet* Sale wallet for use by Tru Ltd as per the Tru Reputation Protocol Whitepaper.
- Executes the *finishMinting* function to finalise all minting activity for the *TruReputationToken*
- Transfers ownership of the *TruReputationToken* back to the executing account now the Crowdsale is complete.
- Executes *releaseTokenTransfer* function.

Usage

The *completion* function has the following usage syntax:

Listing 6: **completion Usage Example**

```
completion();
```

CHAPTER 10

BasicToken

Title:	BasicToken
Description:	Zeppelin Solidity Smart Contract that implements a Basic form of the ERC-20 standard without allowances, approvals, or transferFrom
Author:	Smart Contract Solutions, Inc.
Solidity Version:	^0.4.18
Relative Path:	`./contracts/supporting/BasicToken.sol`
License:	MIT License
Current Version:	1.4.0
Original Source:	BasicToken Source

10.1 1. Imports & Dependencies

The following imports and dependencies exist for the *BasicToken* Smart Contract :

Name	Description
ERC20Basic	Zeppelin Solidity Smart Contract for a Basic ERC-20 Compliance
SafeMath	Zeppelin Solidity Library to perform mathematics safely inside Solidity

10.2 2. Variables

There are no variables for the *BasicToken* Smart Contract.

10.3 3. Enums

There are no enums for the *BasicToken* Smart Contract.

10.4 4. Events

There are no events for the *BasicToken* Smart Contract.

10.5 5. Mappings

The following mappings exist for the *BasicToken* Smart Contract:

Name	Mapping Type	Description
balances	address => uint256	Mapping to track token balance of an address

10.6 6. Modifiers

There are no modifiers for the *BasicToken* Smart Contract.

10.7 7. Functions

The following functions exist for the *BasicToken* Smart Contract:

Name	Description
transfer	Function to transfer tokens.
balanceOf	Function to get the token balance of a given address

10.7.1 transfer

Function Name:	transfer
Description:	Function to transfer tokens
Function Type:	N/A
Function Visibility:	Public
Function Modifiers:	N/A
Return Type:	bool
Return Details:	returns true upon successful transfer

Code

The code for the *transfer* function is as follows:

Listing 1: **transfer 1.4.0 Code**

```
function transfer(address _to, uint256 _value) public returns (bool) {
    require(_to != address(0));
    require(_value <= balances[msg.sender]);

    // SafeMath.sub will throw if there is not enough balance.
    balances[msg.sender] = balances[msg.sender].sub(_value);
    balances[_to] = balances[_to].add(_value);
    Transfer(msg.sender, _to, _value);
    return true;
}
```

The *transfer* function performs the following:

- Checks the *_to* argument is a valid Ethereum address. If not, it will throw.
- Checks that the *_value* argument is less than or equal to the *msg.sender* token balance. If not, it will throw
- Removes the *_value* from the *msg.sender* token balance. If the balance is insufficient, it will throw
- Adds the *_value* to the *_to* token balance.
- Fires the *Transfer* event
- Returns true

Usage

The *transfer* function has the following usage syntax and arguments:

	Argument	Type	Details
1	_to	address	Address to be transfer tokens to
1	_value	uint256	Amount of tokens to transfer

Listing 2: **transfer Usage Example**

```
transfer(0x123456789abcdefghijklmnopqrstuvwxyz98765, 100);
```

10.7.2 balanceOf

Function Name:	balanceOf
Description:	Function to get the token balance of an address
Function Type:	View
Function Visibility:	Public
Function Modifiers:	N/A
Return Type:	uint256
Return Details:	returns token balance of address

Code

The code for the *balanceOf* function is as follows:

Listing 3: **balanceOf 1.4.0 Code**

```
function balanceOf(address _owner) public view returns (uint256 balance) {
    return balances[_owner];
}
```

The *balanceOf* function performs the following:

- returns the balance of the supplied *_owner* address

Usage

The *balanceOf* function has the following usage syntax and arguments:

	Argument	Type	Details
1	_owner	address	Address check the token balance of

Listing 4: **balanceOf Usage Example**

```
balanceOf(0x123456789abcdefghijklmnopqrstuvwxyz98765);
```

CHAPTER 11

ERC20

Title:	ERC20
Description:	Zeppelin Solidity Smart Contract that provides the interface required to implement an ERC20 compliant token.
Author:	Smart Contract Solutions, Inc.
Solidity Version:	^0.4.18
Relative Path:	`./contracts/supporting/ERC20.sol`
License:	MIT License
Current Version:	1.4.0
Original Source:	ERC20 Source

11.1 1. Imports & Dependencies

The following imports and dependencies exist for the *ERC20* Smart Contract:

Name	Description
ERC20Basic	Zeppelin Solidity Smart Contract for a Basic ERC-20 Compliance

11.2 2. Variables

There are no variables for the *ERC20* Smart Contract.

11.3 3. Enums

There are no enums for the *ERC20* Smart Contract.

11.4 4. Events

The following events exist for the *ERC20* Smart Contract:

Name	Description
Approval	Event to track when approval is granted to a spender on a given address

11.4.1 Approval

Event Name:	Approval
Description:	Event to track when approval is granted to a spender on a given address

Usage

The *Approval* event has the following usage syntax and arguments:

	Argument	Type	Indexed?	Details
1	owner	address	Yes	Address that is granting approval
2	spender	address	Yes	Address that has been approved
3	value	uint256	No	Amount of tokens spender is allowed to spend

Listing 1: **Approval Usage Example**

```
Approval(0x123456789abcdefghijklmnopqrstuvwxyz98765,
         0x123456789abcdefghijklmnopqrstuvwxyz12345,
         100);
```

11.5 5. Mappings

The are no mappings for the *ERC20* Smart Contract.

11.6 6. Modifiers

There are no modifiers for the *ERC20* Smart Contract.

11.7 7. Functions

The following functions exist for the *ERC20* Smart Contract:

Name	Description
allowance	Function to get the approved allowance for a transfer of tokens from an address by a spender address
approve	Function to approve a particular allowance to be transferred by that spender address on the target address
transferFrom	Function transfer tokens from an address to another invoked by an authorised spender address

11.7.1 allowance

Function Name:	allowance
Description:	Function to get the approved allowance for a transfer of tokens from an address by a spender address
Function Type:	View
Function Visibility:	Public
Function Modifiers:	N/A
Return Type:	uint256
Return Details:	returns the remaining allowance the spender has on the target address

Code

The code for the *allowance* function is an interface and it is defined as follows:

Listing 2: **allowance 1.4.0 Code**

```
function allowance(address owner, address spender) public view returns (uint256);
```

Usage

The *allowance* function has the following usage syntax and arguments:

	Argument	Type	Details
1	owner	address	Address that spender has been given an allowance on
2	spender	address	Address of the spender

Listing 3: **allowance Usage Example**

```
allowance(0x123456789abcdefghijklmnopqrstuvwxyz98765,
          0x123456789abcdefghijklmnopqrstuvwxyz12345);
```

11.7.2 approve

Function Name:	approve
Description:	Function to approve a spender address to have a particular allowance to be transferred or spent by that spender address on the target address
Function Type:	View
Function Visibility:	N/A
Function Modifiers:	N/A
Return Type:	bool
Return Details:	returns a bool to denote success or failure to approve

Code

The code for the *approve* function is an interface and it is defined as follows:

Listing 4: **approve 1.4.0 Code**

```
function approve(address spender, uint256 value) public returns (bool);
```

Usage

The *approve* function has the following usage syntax and arguments:

	Argument	Type	Details
1	spender	address	Address be granted an allowance

Listing 5: **approve Usage Example**

```
approve(0x123456789abcdefghijklmnopqrstuvwxyz98765, 100);
```

11.7.3 transferFrom

Function Name:	transferFrom
Description:	Function transfer tokens from an address to another invoked by an authorised spender address
Function Type:	N/A
Function Visibility:	Public
Function Modifiers:	N/A
Return Type:	bool
Return Details:	returns a bool to denote success or failure to transfer

Code

The code for the *transferFrom* function is an interface and it is defined as follows:

Listing 6: **transferFrom 1.4.0 Code**

```
function transferFrom(address from, address to, uint256 value) public returns (bool);
```

Usage

The *transferFrom* function has the following usage syntax and arguments:

	Argument	Type	Details
1	from	address	Address to transfer tokens from
2	to	address	Address to send tokens to
3	value	uint256	Amount of tokens to transfer

Listing 7: **transferFrom Usage Example**

```
transferFrom(0x123456789abcdefghijklmnopqrstuvwxyz98765,
             0x123456789abcdefghijklmnopqrstuvwxyz54321,
             100);
```

CHAPTER 12

ERC20Basic

Title:	ERC20Basic
Description:	Zeppelin Solidity Smart Contract that provides a basic interface required to implement an ERC20 compliant token.
Author:	Smart Contract Solutions, Inc.
Solidity Version:	^0.4.18
Relative Path:	`./contracts/supporting/ERC20Basic.sol`
License:	MIT License
Current Version:	1.4.0
Original Source:	ERC20Basic Source

12.1 1. Imports & Dependencies

There are no imports or dependencies for the *ERC20Basic* Smart Contract.

12.2 2. Variables

The following variables exist for the *ERC20Basic* Smart Contract:

Variable	Type	Vis	Details
totalSupply	uint256	public	Variable to provide total count of all token in circulation for this token

12.3 3. Enums

There are no enums for the *ERC20Basic* Smart Contract.

12.4 4. Events

The following events exist for the *ERC20Basic* Smart Contract:

Name	Description
Transfer	Event to track when a transfer of tokens occurs between addresses

12.4.1 Transfer

Event Name:	Transfer
Description:	Event to track when a transfer of tokens occurs between addresses

Usage

The *Transfer* event has the following usage syntax and arguments:

	Argument	Type	Indexed?	Details
1	from	address	Yes	Address where tokens are being transferred from
2	to	address	Yes	Address where tokens are being transferred to
3	value	uint256	No	Amount of tokens to transfer

Listing 1: **Transfer Usage Example**

```
Transfer(0x123456789abcdefghijklmnopqrstuvwxyz98765,
         0x123456789abcdefghijklmnopqrstuvwxyz12345,
         100);
```

12.5 5. Mappings

The are no mappings for the *ERC20Basic* Smart Contract.

12.6 6. Modifiers

There are no modifiers for the *ERC20Basic* Smart Contract.

12.7 7. Functions

The following functions exist for the *ERC20Basic* Smart Contract:

Name	Description
balanceOf	Function to get the token balance of a supplied address
transfer	Function to allow transferring tokens from one address to another

12.7.1 balanceOf

Function Name:	balanceOf
Description:	Function to get the token balance of a supplied address
Function Type:	View
Function Visibility:	Public
Function Modifiers:	N/A
Return Type:	uint256
Return Details:	returns the token balance of the supplied address

Code

The code for the *balanceOf* function is an interface and it is defined as follows:

Listing 2: **balanceOf 1.4.0 Code**

```
function balanceOf(address who) public view returns (uint256);
```

Usage

The *balanceOf* function has the following usage syntax and arguments:

	Argument	Type	Details
1	who	address	Address to retrieve the token balance of

Listing 3: **allowance Usage Example**

```
balanceOf(0x123456789abcdefghijklmnopqrstuvwxyz98765);
```

12.7.2 transfer

Function Name:	transfer
Description:	Function to allow transferring tokens from one address to another
Function Type:	N/A
Function Visibility:	N/A
Function Modifiers:	N/A
Return Type:	bool
Return Details:	returns a bool to denote success or failure to transfer tokens

Code

The code for the *transfer* function is an interface and it is defined as follows:

Listing 4: **transfer 1.4.0 Code**

```
function transfer(address to, uint256 value) public returns (bool);
```

Usage

The *transfer* function has the following usage syntax and arguments:

	Argument	Type	Details
1	to	address	Address to transfer tokens to
2	value	uint256	Amount of tokens to transfer

Listing 5: **transfer Usage Example**

```
transfer(0x123456789abcdefghijklmnopqrstuvwxyz98765, 100);
```

CHAPTER 13

Haltable

Title:	Haltable
Description:	Modified Token Market Smart Contract that provides a capability to halt a contract (namely a CrowdSale). Updated by Tru Ltd.
Author:	TokenMarket Ltd/Updated by Ian Bray, Tru Ltd
Solidity Version:	0.4.18
Relative Path:	`./contracts/supporting/Haltable.sol`
License:	Apache 2 License
Current Version:	0.1.12
Original Source:	Haltable Source

13.1 1. Imports & Dependencies

The following imports and dependencies exist for the *Haltable* Smart Contract:

Name	Description
Ownable	Zeppelin Solidity Smart Contract for Ownership capabilities in a token

13.2 2. Variables

The following variables exist for the *Haltable* Smart Contract:

Variable	Type	Vis	Details
halted	bool	public	Variable to indicate whether the contract is halted or not

13.3 3. Enums

There are no enums for the *Haltable* Smart Contract.

13.4 4. Events

The following events exist for the *Haltable* Smart Contract:

Name	Description
HaltStatus	Event to track halted status changes

13.4.1 HaltStatus

Event Name:	HaltStatus
Description:	Event to track halted status changes

Usage

The *HaltStatus* event has the following usage syntax and arguments:

	Argument	Type	Indexed?	Details
1	status	bool	No	Whether the contract is halted or not

Listing 1: **HaltStatus Usage Example**

```
HaltStatus(true);
```

13.5 5. Mappings

The are no mappings for the *Haltable* Smart Contract.

13.6 6. Modifiers

The following modifiers exist for the *Haltable* Smart Contract:

Name	Description
stopInEmergency	Modifier that requires the contract is not halted
onlyInEmergency	Modifier that requires the contract is halted

13.6.1 stopInEmergency

Modifier Name:	stopInEmergency
Description:	Modifier that requires the contract is not halted

Code

The code for the *stopInEmergency* modifier is as follows:

Listing 2: **stopInEmergency Code**

```
modifier stopInEmergency {
    require(!halted);
    _;
}
```

The *stopInEmergency* function performs the following:

- Checks that the *halted* variable is false otherwise it throws

13.6.2 onlyInEmergency

Modifier Name:	onlyInEmergency
Description:	Modifier that requires the contract is halted

Code

The code for the *onlyInEmergency* modifier is as follows:

Listing 3: **onlyInEmergency Code**

```
modifier onlyInEmergency {
    require(halted);
    _;
}
```

The *onlyInEmergency* function performs the following:

- Checks that the *halted* variable is true otherwise it throws

13.7 7. Functions

The following functions exist for the *Haltable* Smart Contract:

Name	**Description**
halt	Function to halt the contract
unhalt	Function to unhalt the contract

13.7.1 halt

Function Name:	halt
Description:	Function to halt the contract
Function Type:	N/A
Function Visibility:	External
Function Modifiers:	*onlyOwner*
Return Type:	None
Return Details:	N/A

Code

The code for the *halt* function is as follows:

Listing 4: **halt Code**

```
function halt() external onlyOwner {
    halted = true;
    HaltStatus(halted);
}
```

The *halt* function performs the following:

- Sets the *halted* variable to true
- Fires the *HaltStatus* event

Usage

The *halt* function has the following usage syntax:

Listing 5: **halt Usage Example**

```
halt();
```

13.7.2 unhalt

Function Name:	unhalt
Description:	Function to unhalt the contract
Function Type:	N/A
Function Visibility:	External
Function Modifiers:	*onlyOwner*, *onlyInEmergency*
Return Type:	None
Return Details:	N/A

Code

The code for the *unhalt* function is as follows:

Listing 6: **unhalt Code**

```
function unhalt() external onlyOwner onlyInEmergency {
    halted = false;
    HaltStatus(halted);
}
```

The *unhalt* function performs the following:

- Sets the *halted* variable to false
- Fires the *HaltStatus* event

Usage

The *unhalt* function has the following usage syntax:

Listing 7: **unhalt Usage Example**

```
unhalt();
```

CHAPTER 14

Ownable

Title:	Ownable
Description:	Zeppelin Solidity Smart Contract that provides ownership capabilities to a contract.
Author:	Smart Contract Solutions, Inc.
Solidity Version:	^0.4.18
Relative Path:	`./contracts/supporting/Ownable.sol`
License:	MIT License
Current Version:	1.4.0
Original Source:	Ownable Source

14.1 1. Imports & Dependencies

There are no imports and dependencies for the *Ownable* Smart Contract.

14.2 2. Variables

The following variables exist for the *Ownable* Smart Contract:

Variable	Type	Vis	Details
owner	address	public	Variable containing the address of the contract owner

14.3 3. Enums

There are no enums for the *Ownable* Smart Contract.

14.4 4. Events

The following events exist for the *Ownable* Smart Contract:

Name	Description
OwnershipTransferred	Event to track change of ownership

14.4.1 OwnershipTransferred

Event Name:	OwnershipTransferred
Description:	Event to track change of ownership

Usage

The *OwnershipTransferred* event has the following usage syntax and arguments:

	Argument	Type	Indexed?	Details
1	previousOwner	address	Yes	Address of the previous owner
2	newOwner	address	Yes	Address of the new owner

Listing 1: **OwnershipTransferred Usage Example**

```
OwnershipTransferred(0x123456789abcdefghijklmnopqrstuvwxyz98765,
                     0x123456789abcdefghijklmnopqrstuvwxyz54321);
```

14.5 5. Mappings

The are no mappings for the *Ownable* Smart Contract.

14.6 6. Modifiers

The following modifiers exist for the *Ownable* Smart Contract:

Name	Description
onlyOwner	Modifier that requires the contract owner is the sender of the transaction

14.6.1 onlyOwner

Modifier Name:	onlyOwner
Description:	Modifier that requires the contract owner is the sender of the transaction

Code

The code for the *onlyOwner* modifier is as follows:

Listing 2: **onlyOwner 1.4.0 Code**

```
modifier onlyOwner() {
    require(msg.sender == owner);
    _;
}
```

The *onlyOwner* function performs the following:

- Checks that the transaction sender address is the same as the *owner* address variable otherwise it throws

14.7 7. Functions

The following functions exist for the *Ownable* Smart Contract:

Name	Description
Ownable Constructor	Constructor Function for Ownable Contract
transferOwnership	Function transfer the contract ownership

14.7.1 Ownable Constructor

Function Name:	Ownable
Description:	Constructor for the *Ownable* Smart Contract
Function Type:	Constructor
Function Visibility:	Public
Function Modifiers:	N/A
Return Type:	None
Return Details:	N/A

Code

The code for the *Ownable Constructor* function is as follows:

Listing 3: **Ownable Constructor 1.4.0 Code**

```
function Ownable() public {
    owner = msg.sender;
}
```

The *Ownable Constructor* function performs the following:

- Sets the *owner* variable *msg.sender*

Usage

The *Ownable Constructor* function has the following usage syntax:

Listing 4: **Ownable Constructor Usage Example**

```
Ownable();
```

14.7.2 transferOwnership

Function Name:	transferOwnership
Description:	Function transfer the contract ownership
Function Type:	N/A
Function Visibility:	Public
Function Modifiers:	*onlyOwner*
Return Type:	None
Return Details:	N/A

Code

The code for the *transferOwnership* function is as follows:

Listing 5: **transferOwnership 1.4.0 Code**

```
function transferOwnership(address newOwner) public onlyOwner {
    require(newOwner != address(0));
    OwnershipTransferred(owner, newOwner);
    owner = newOwner;
}
```

The *transferOwnership* function performs the following:

- Validates that the supplied *newOwner* argument is a valid Ethereum address. If it is not, it will throw.
- Fires the *OwnershipTransferred* event.
- sets the *owner* to the *newOwner* argument value.

Usage

The *transferOwnership* function has the following usage syntax:

Listing 6: **transferOwnership Usage Example**

```
transferOwnership(0x123456789abcdefghijklmnopqrstuvwxyz98765);
```

CHAPTER 15

ReleaseableToken

Title:	ReleaseableToken
Description:	Smart Contract derived from **ReleaseableToken** by Token Market with additional functionality for the **TruReputationToken**.
Author:	Ian Bray, Tru Ltd
Solidity Version:	0.4.18
Relative Path:	`./contracts/supporting/ReleaseableToken.sol`
License:	Apache 2 License
Current Version:	0.1.12
Original Source:	ReleaseableToken Source

15.1 1. Imports & Dependencies

The following imports and dependencies exist for the *ReleaseableToken* Solidity Library:

Name	Description
Ownable	Zeppelin Solidity Smart Contract that provides ownership capabilities to a contract.
StandardToken	Zeppelin Solidity Smart Contract for a Standard ERC-20 Token

15.2 2. Variables

The following variables exist for the *ReleaseableToken* Smart Contract:

Variable	Type	Vis	Details
releaseAgent	address	public	Variable containing the address of the Release Agent
released	bool	public	Variable for whether the token is released or not **Default:** *false*

15.3 3. Enums

There are no enums for the *ReleaseableToken* Smart Contract.

15.4 4. Events

The following events exist for the *ReleaseableToken* Smart Contract:

Name	Description
Released	Event to notify when a token is released
ReleaseAgentSet	Event to notify when a releaseAgent is set
TransferAgentSet	Event to notify when a Transfer Agent is set or updated

15.4.1 Released

Event Name:	Released
Description:	Event to notify when a token is released

Usage

The *Released* event has the following usage syntax:

Listing 1: **Released Usage Example**

```
Released();
```

15.4.2 ReleaseAgentSet

Event Name:	ReleaseAgentSet
Description:	Event to notify when a releaseAgent is set

Usage

The *ReleaseAgentSet* event has the following usage syntax and arguments:

	Argument	Type	Indexed?	Details
1	releaseAgent	address	Yes	Address of new *releaseAgent*

Listing 2: **ReleaseAgentSet Usage Example**

```
ReleaseAgentSet(0x123456789abcdefghijklmnopqrstuvwxyz98765);
```

15.4.3 TransferAgentSet

Event Name:	TransferAgentSet
Description:	Event to notify when a Transfer Agent is set or updated

Usage

The *TransferAgentSet* event has the following usage syntax and arguments:

	Argument	**Type**	**Indexed?**	**Details**
1	transferAgent	address	Yes	Address of new Transfer Agent
2	status	bool	Yes	Whether Transfer Agent is enabled or disabled

Listing 3: **TransferAgentSet Usage Example**

```
TransferAgentSet(0x123456789abcdefghijklmnopqrstuvwxyz98765, true);
```

15.5 5. Mappings

The following mappings exist for the *ReleaseableToken* Smart Contract:

Name	**Mapping Type**	**Description**
transferAgents	address => uint256	Mapping to status of transfer agents

15.6 6. Modifiers

The following modifiers exist for the *ReleaseableToken* Smart Contract:

Name	**Description**
canTransfer	Modifier that checks whether token is in a transferable state
inReleaseState	Modifier that checks whether token is in a given released state
onlyReleaseAgent	Modifier that checks whether the executor is the *releaseAgent*

15.6.1 canTransfer

Modifier Name:	canTransfer
Description:	Modifier that checks whether token is in a transferable state

Code

The code for the *canTransfer* modifier is as follows:

Listing 4: **canTransfer Code**

```
modifier canTransfer(address _sender) {
    require(released || transferAgents[_sender]);
    _;
}
```

The *canTransfer* function performs the following:

- Checks that the *released* variable is true and that the *_sender* argument is in the *transferAgents* mapping otherwise it throws

15.6.2 inReleaseState

Modifier Name:	inReleaseState
Description:	Modifier that checks whether token is in a given released state

Code

The code for the *inReleaseState* modifier is as follows:

Listing 5: **inReleaseState Code**

```
modifier inReleaseState(bool releaseState) {
    require(releaseState == released);
    _;
}
```

The *inReleaseState* function performs the following:

- Checks that the supplied *releaseState* argument matches the *released* variable otherwise it throws

15.6.3 onlyReleaseAgent

Modifier Name:	onlyReleaseAgent
Description:	Modifier that checks whether the executor is the *releaseAgent*

Code

The code for the *onlyReleaseAgent* modifier is as follows:

Listing 6: **onlyReleaseAgent Code**

```
modifier onlyReleaseAgent() {
    require(msg.sender == releaseAgent);
    _;
}
```

The *onlyReleaseAgent* function performs the following:

- Checks that the transaction sender address matches the *releaseAgent* address otherwise it throws

15.7 7. Functions

The following functions exist for the *ReleaseableToken* Smart Contract:

Name	Description
setReleaseAgent	Function to set the* *releaseAgent* variable
setTransferAgent	Function to set or update the* *transferAgents* mapping
releaseTokenTransfer	Function to release the token
transfer	Function to override *transfer* function
transferFrom	Function to override *transferFrom* function

15.7.1 setReleaseAgent

Function Name:	setReleaseAgent
Description:	Function to set the* *releaseAgent* variable
Function Type:	N/A
Function Visibility:	Public
Function Modifiers:	*onlyOwner*, *inReleaseState*
Return Type:	None
Return Details:	N/A

Code

The code for the *setReleaseAgent* function is as follows:

Listing 7: **setReleaseAgent Code**

```
function setReleaseAgent(address addr) public onlyOwner inReleaseState(false) {
    ReleaseAgentSet(addr);
    // We don't do interface check here as we might want to a normal wallet address␣
↪to act as a release agent
    releaseAgent = addr;
}
```

The *setReleaseAgent* function performs the following:

- Fires the *ReleaseAgentSet* event
- Sets the *releaseAgent* variable to the *addr* argument

Usage

The *setReleaseAgent* function has the following usage syntax:

Listing 8: **setReleaseAgent Usage Example**

```
setReleaseAgent(0x123456789abcdefghijklmnopqrstuvwxyz98765);
```

15.7.2 setTransferAgent

Function Name:	setTransferAgent
Description:	Function to set or update the* *transferAgents* mapping
Function Type:	N/A
Function Visibility:	Public
Function Modifiers:	*onlyOwner*, *inReleaseState*
Return Type:	None
Return Details:	N/A

Code

The code for the *setTransferAgent* function is as follows:

Listing 9: **setTransferAgent Code**

```
function setTransferAgent(address addr, bool state) public onlyOwner␣
↪inReleaseState(false) {
    TransferAgentSet(addr, state);
    transferAgents[addr] = state;
}
```

The *setTransferAgent* function performs the following:

- Fires the *TransferAgentSet* event
- Add the supplied *addr* and *state* to the *transferAgents* mapping

Usage

The *setTransferAgent* function has the following usage syntax:

Listing 10: **setTransferAgent Usage Example**

```
setTransferAgent(0x123456789abcdefghijklmnopqrstuvwxyz98765, true);
```

15.7.3 releaseTokenTransfer

Function Name:	releaseTokenTransfer
Description:	Function to release the token
Function Type:	N/A
Function Visibility:	Public
Function Modifiers:	*onlyReleaseAgent*
Return Type:	None
Return Details:	N/A

Code

The code for the *releaseTokenTransfer* function is as follows:

Listing 11: **releaseTokenTransfer Code**

```
function releaseTokenTransfer() public onlyReleaseAgent {
    Released();
    released = true;
}
```

The *releaseTokenTransfer* function performs the following:

- Fires the *Released* event
- Sets the *released* variable to true

Usage

The *releaseTokenTransfer* function has the following usage syntax:

Listing 12: **releaseTokenTransfer Usage Example**

```
releaseTokenTransfer();
```

15.7.4 transfer

Function Name:	transfer
Description:	Function to override transfer function
Function Type:	N/A
Function Visibility:	Public
Function Modifiers:	*canTransfer*
Return Type:	bool
Return Details:	Returns whether the transfer was successful or not

Code

The code for the *transfer* function is as follows:

Listing 13: **transfer Code**

```
function transfer(address _to,
                  uint _value) public canTransfer(msg.sender) returns (bool success) {
    return super.transfer(_to, _value);
}
```

The *transfer* function performs the following:

- calls the *transfer* super function

Usage

The *transfer* function has the following usage syntax and arguments:

	Argument	Type	Details
1	_to	address	Address to be sent *_value* to
2	_value	uint	Value of tokens to send to *_to* address

Listing 14: **transfer Usage Example**

```
transfer(0x123456789abcdefghijklmnopqrstuvwxyz98765, true);
```

15.7.5 transferFrom

Function Name:	transferFrom
Description:	Function to override transferFrom function
Function Type:	N/A
Function Visibility:	Public
Function Modifiers:	*canTransfer*
Return Type:	bool
Return Details:	Returns whether the transferFrom was successful or not

Code

The code for the *transferFrom* function is as follows:

Listing 15: **transferFrom Code**

```
function transferFrom(address _from,
                      address _to,
                      uint _value) public canTransfer(_from) returns (bool success) {
    return super.transferFrom(_from, _to, _value);
}
```

The *transferFrom* function performs the following:

- calls the *transferFrom* super function

Usage

The *transferFrom* function has the following usage syntax and arguments:

	Argument	Type	Details
1	_fro	address	Address to be sent *_value* from
2	_to	address	Address to be sent *_value* to
3	_value	uint	Value of tokens to send to *_to* address

Listing 16: **transferFrom Usage Example**

```
transferFrom(0x123456789abcdefghijklmnopqrstuvwxyz98765,
             0x423456789abcdefghijklmnopqrstuvwxyz12345,
             true);
```

CHAPTER 16

SafeMath

Title:	SafeMath
Description:	Zeppelin Solidity Library for Math operations with safety checks throws on error.
Author:	Smart Contract Solutions, Inc.
Solidity Version:	^0.4.18
Relative Path:	`./contracts/supporting/SafeMath.sol`
License:	MIT License
Current Version:	1.4.0
Original Source:	SafeMath Source

No modifications have been made to this Solidity Library from the original source.

16.1 1. Imports & Dependencies

There are no imports and dependencies exist for the *SafeMath* Solidity Library.

16.2 2. Variables

There are no variables for the *SafeMath* Solidity Library.

16.3 3. Enums

There are no enums for the *SafeMath* Solidity Library.

16.4 4. Events

There are no events for the *SafeMath* Solidity Library.

16.5 5. Mappings

There are no mappings for the *SafeMath* Solidity Library.

16.6 6. Modifiers

There are no modifiers for the *SafeMath* Solidity Library.

16.7 7. Functions

The following functions exist for the *SafeMath* Smart Contract:

Name	Description
mul	Function to safely multiply two numbers
div	Function to safely divide one number from another
sub	Function to safely subtract one number from another
add	Function to safely add two numbers

16.7.1 mul

Function Name:	mul
Description:	Function to safely multiply two numbers
Function Type:	Pure
Function Visibility:	Internal
Function Modifiers:	None
Return Type:	uin256
Return Details:	Returns the result of the multiplication

Code

The code for the *mul* function is as follows:

Listing 1: **mul 1.4.0 Code**

```
function mul(uint256 a, uint256 b) internal pure returns (uint256) {
    if (a == 0) {
        return 0;
    }
    uint256 c = a * b;
    assert(c / a == b);
    return c;
}
```

The *mul* function performs the following:

- if the *a* argument is zero, if it is returns zero
- Multiply *a* argument by *b* argument
- Checks that the result divided by *a* argument equals the *b* argument. If not, it will throw
- Return the result

Usage

The *mul* function has the following usage syntax:

Listing 2: **mul Usage Example**

```
mul(2,2);
```

16.7.2 div

Function Name:	div
Description:	Function to safely divide one number from another
Function Type:	Pure
Function Visibility:	Internal
Function Modifiers:	None
Return Type:	uin256
Return Details:	Returns the result of the division

Code

The code for the *div* function is as follows:

Listing 3: **div 1.4.0 Code**

```
function div(uint256 a, uint256 b) internal pure returns (uint256) {

    uint256 c = a / b;

    return c;
}
```

The *div* function performs the following:

- Divide *a* argument by *b* argument
- Return the result

Usage

The *div* function has the following usage syntax:

Listing 4: **div Usage Example**

```
div(2,2);
```

16.7.3 sub

Function Name:	sub
Description:	Function to safely subtract one number from another
Function Type:	Pure
Function Visibility:	Internal
Function Modifiers:	None
Return Type:	uin256
Return Details:	Returns the result of the subtraction

Code

The code for the *sub* function is as follows:

Listing 5: **sub 1.4.0 Code**

```
function sub(uint256 a, uint256 b) internal pure returns (uint256) {
    assert(b <= a);
    return a - b;
}
```

The *sub* function performs the following:

- Checks the *b* argument is equal to or less than the *a* argument. If not, it will throw
- Calculate and result the *a* argument minus the *b* argument

Usage

The *sub* function has the following usage syntax:

Listing 6: **sub Usage Example**

```
sub(2,1);
```

16.7.4 add

Function Name:	add
Description:	Function to safely add two numbers
Function Type:	Pure
Function Visibility:	Internal
Function Modifiers:	None
Return Type:	uin256
Return Details:	Returns the result of the addition

Code

The code for the *add* function is as follows:

Listing 7: **add 1.4.0 Code**

```
function add(uint256 a, uint256 b) internal pure returns (uint256) {
    uint256 c = a + b;
    assert(c >= a);
    return c;
}
```

The *add* function performs the following:

- Adds *a* argument to *b* argument
- Checks that the result is greater than the *a* argument. If not, it will throw.
- Returns the result

Usage

The *add* function has the following usage syntax:

Listing 8: **add Usage Example**

```
add(2,2);
```

CHAPTER 17

StandardToken

Title:	StandardToken
Description:	Zeppelin Solidity Smart Contract that provides a standard ERC-20 compliant token.
Author:	Smart Contract Solutions, Inc.
Solidity Version:	^0.4.18
Relative Path:	`./contracts/supporting/StandardToken.sol`
License:	MIT License
Current Version:	1.4.0
Original Source:	StandardToken Source

17.1 1. Imports & Dependencies

The following imports and dependencies exist for the *StandardToken* Smart Contract:

Name	Description
Basic-Token	Zeppelin Solidity Smart Contract that implements a Basic form of the ERC-20 standard without allowances, approvals, or transferFrom
ERC20	Zeppelin Solidity Smart Contract that provides the interface required to implement an ERC20 compliant token.

17.2 2. Variables

There are no variables for the *StandardToken* Smart Contract.

17.3 3. Enums

There are no enums for the *StandardToken* Smart Contract.

17.4 4. Events

TThere are no events for the *StandardToken* Smart Contract.

17.5 5. Mappings

The following mappings exist for the *StandardToken* Smart Contract:

Name	Mapping Type	Description
allowed	address => mapping(address => uint256)	Mapping to allowance authorisation

17.6 6. Modifiers

There are no modifiers for the *StandardToken* Smart Contract.

17.7 7. Functions

The following functions exist for the *StandardToken* Smart Contract:

Name	Description
transferFrom	Function transfer tokens from an address to another invoked by an authorised spender address
approve	Function to approve a particular allowance to be transferred by that spender address on the target address
allowance	Function to get the approved allowance for a transfer of tokens from an address by a spender address
increaseApproval	Function to allow increase approved allowance for a spender on a given address
decreaseApproval	Function to allow decrease approved allowance for a spender on a given address

17.7.1 transferFrom

Function Name:	transferFrom
Description:	Function transfer tokens from an address to another invoked by an authorised spender address
Function Type:	N/A
Function Visibility:	Public
Function Modifiers:	N/A
Return Type:	bool
Return Details:	returns where the transfer was successful or not

Code

The code for the *transferFrom* function is as follows:

Listing 1: **transferFrom 1.4.0 Code**

```
function transferFrom(address _from, address _to, uint256 _value) public returns␣
↪(bool) {
    require(_to != address(0));
    require(_value <= balances[_from]);
    require(_value <= allowed[_from][msg.sender]);

    balances[_from] = balances[_from].sub(_value);
    balances[_to] = balances[_to].add(_value);
    allowed[_from][msg.sender] = allowed[_from][msg.sender].sub(_value);
    Transfer(_from, _to, _value);
    return true;
}
```

The *transferFrom* function performs the following:

- Checks the *_to* argument is a valid Ethereum address. If not, it will throw.
- Checks that the *_value* argument is less than or equal to the *_from* token balance. If not, it will throw
- Checks that *_value* argument is less than or equal to the *allowed* balance for the *msg.sender*. If not it will throw.
- Removes the *_value* from the *_from* token balance. If the balance is insufficient, it will throw
- Adds the *_value* to the *_to* token balance.
- Removes the *_value* from the allowance for this spender on this address.
- Fires the *Transfer* event
- Returns true

Usage

The *transferFrom* function has the following usage syntax and arguments:

	Argument	Type	Details
1	_from	address	Address transfer tokens from
2	_to	address	Address transfer tokens to
3	_value	uint256	Number of tokens to transfer

Listing 2: **transferFrom Usage Example**

```
transferFrom(0x123456789abcdefghijklmnopqrstuvwxyz98765,
             0x543456789abcdefghijklmnopqrstuvwxyz12234,
             100);
```

17.7.2 approve

Function Name:	approve
Description:	Function to approve a particular allowance to be transferred by that spender address on the target address
Function Type:	N/A
Function Visibility:	Public
Function Modifiers:	N/A
Return Type:	bool
Return Details:	Returns where the approval was successful or not

Code

The code for the *approve* function is as follows:

Listing 3: **approve 1.4.0 Code**

```
function approve(address _spender, uint256 _value) public returns (bool) {
    allowed[msg.sender][_spender] = _value;
    Approval(msg.sender, _spender, _value);
    return true;
}
```

The *approve* function performs the following:

- Sets the allowance for the *_spender* on the *msg.sender* address to the *_value*
- Fires the *Approval* event
- Returns true

Usage

The *approve* function has the following usage syntax and arguments:

	Argument	Type	Details
1	_spender	address	Address to grant approval to
2	_to	address	Allowance ot grant spender

Listing 4: **approve Usage Example**

```
approve(0x123456789abcdefghijklmnopqrstuvwxyz98765,100);
```

17.7.3 allowance

Function Name:	allowance
Description:	Function to approve a particular allowance to be transferred by that spender address on the target address
Function Type:	View
Function Visibility:	Public
Function Modifiers:	N/A
Return Type:	uint256
Return Details:	Returns the Current balance of approved tokens an address can transfer

Code

The code for the *allowance* function is as follows:

Listing 5: **allowance 1.4.0 Code**

```
function allowance(address _owner, address _spender) public view returns (uint256) {
    return allowed[_owner][_spender];
}
```

The *allowance* function performs the following:

- Returns true

Usage

The *allowance* function has the following usage syntax and arguments:

	Argument	Type	Details
1	_owner	address	Address subject to allowance
2	_spender	address	Address granted an allowance

Listing 6: **allowance Usage Example**

```
allowance(0x123456789abcdefghijklmnopqrstuvwxyz98765,
          0x543456789abcdefghijklmnopqrstuvwxyz12234);
```

17.7.4 increaseApproval

Function Name:	increaseApproval
Description:	Function to increase the existing approved allowance of a spender address on the target address
Function Type:	N/A
Function Visibility:	Public
Function Modifiers:	N/A
Return Type:	bool
Return Details:	Current balance of approved tokens an address can transfer

Code

The code for the *increaseApproval* function is as follows:

Listing 7: **increaseApproval 1.4.0 Code**

```
function increaseApproval(address _spender, uint _addedValue) public returns (bool) {
    allowed[msg.sender][_spender] = allowed[msg.sender][_spender].add(_addedValue);
    Approval(msg.sender, _spender, allowed[msg.sender][_spender]);
    return true;
}
```

The *increaseApproval* function performs the following:

- Adds the *_addedValue* argument to the current allowance
- Fires the *Approval* event
- Returns true

Usage

The *increaseApproval* function has the following usage syntax and arguments:

	Argument	Type	Details
1	_spender	address	Address to increase the allowance for
2	_addedValue	address	Amount to add to the allowance

Listing 8: **increaseApproval Usage Example**

```
increaseApproval(0x123456789abcdefghijklmnopqrstuvwxyz98765,
                 100);
```

17.7.5 decreaseApproval

Function Name:	decreaseApproval
Description:	Function to increase the existing approved allowance of a spender address on the target address
Function Type:	N/A
Function Visibility:	Public
Function Modifiers:	N/A
Return Type:	bool
Return Details:	Current balance of approved tokens an address can transfer

Code

The code for the *increaseApproval* function is as follows:

Listing 9: **decreaseApproval 1.4.0 Code**

```
function decreaseApproval(address _spender, uint _subtractedValue) public returns
↪(bool) {
    uint oldValue = allowed[msg.sender][_spender];
    if (_subtractedValue > oldValue) {
        allowed[msg.sender][_spender] = 0;
    } else {
        allowed[msg.sender][_spender] = oldValue.sub(_subtractedValue);
    }
    Approval(msg.sender, _spender, allowed[msg.sender][_spender]);
    return true;
}
```

The *decreaseApproval* function performs the following:

- Calculates the current approved allowance
- If the current value is less than the *_subtractedValue* argument, the allowance is set to zero
- Otherwise it removes the *_subtractedValue* argument from the allowance
- Fires the *Approval* event
- Returns true

Usage

The *decreaseApproval* function has the following usage syntax and arguments:

	Argument	Type	Details
1	_spender	address	Address to decrease the allowance for
2	_subtractedValue	address	Amount to remove from the allowance

Listing 10: **decreaseApproval Usage Example**

```
decreaseApproval(0x123456789abcdefghijklmnopqrstuvwxyz98765,
                 100);
```

CHAPTER 18

TruAddress

Title:	TruAddress
Description:	Library of helper functions surrounding the Solidity Address type
Author:	Ian Bray, Tru Ltd
Solidity Version:	0.4.18
Relative Path:	`./contracts/supporting/TruAddress.sol`
License:	Apache 2 License
Current Version:	0.1.12

18.1 1. Imports & Dependencies

The following imports and dependencies exist for the *TruAddress* Solidity Library:

Name	Description
SafeMath	Zeppelin Solidity Library to perform mathematics safely inside Solidity

18.2 2. Variables

There are no variables for the *TruAddress* Solidity Library.

18.3 3. Enums

There are no enums for the *TruAddress* Solidity Library.

18.4 4. Events

There are no events for the *TruAddress* Solidity Library.

18.5 5. Mappings

There are no mappings for the *TruAddress* Solidity Library.

18.6 6. Modifiers

There are no modifiers for the *TruAddress* Solidity Library.

18.7 7. Functions

The following functions exist for the *TruAddress* Solidity Library:

Name	Description
isValid	Function to validate a supplied ethereum address
toString	Function to convert an Address to a String
addressLength	Function to return the length of a given Address

18.7.1 isValid

Function Name:	isValid
Description:	Function to validate a supplied address is the correct length & format
Function Type:	Pure
Function Visibility:	Public
Function Modifiers:	N/A
Return Type:	Bool
Return Details:	Returns true for valid input address; false for invalid input address

Code

The code for the *isValid* is as follows:

Listing 1: **isValid Code**

```
function isValid(address input) public pure returns (bool) {
    uint addrLength = addressLength(input);
    return ((addrLength == 20) && (input != address(0)));
}
```

The *isValid* function performs the following:

- Retrieves the address length
- returns a bool check that the address is both 20 characters long and not an empty address

Usage

The *isValid* function has the following usage syntax and arguments:

	Argument	Type	Details
1	input	address	Address to be validated

Listing 2: **isValid Usage Example**

```
isValid(0x123456789abcdefghijklmnopqrstuvwxyz98765);
```

18.7.2 toString

Function Name:	toString
Description:	Function to convert an address to a string
Function Type:	Pure
Function Visibility:	Internal
Function Modifiers:	N/A
Return Type:	String
Return Details:	Returns the address in string format

Code

The code for the *toString* is as follows:

Listing 3: **toString Code**

```
function toString(address input) internal pure returns (string) {
    bytes memory byteArray = new bytes(20);
    for (uint i = 0; i < 20; i++) {
        byteArray[i] = byte(uint8(uint(input) / (2**(8*(19 - i)))));
    }
    return string(byteArray);
}
```

The *toString* function performs the following:

- Creates a 20 byte array
- iterates through the address and converts each byte
- returns the byteArray as a string

Usage

The *toString* function has the following usage syntax and arguments:

	Argument	Type	Details
1	input	address	Address to be converted to a string

Listing 4: **toString Usage Example**

```
toString(0x123456789abcdefghijklmnopqrstuvwxyz98765);
```

18.7.3 addressLength

Function Name:	addressLength
Description:	Function to return the length of an address
Function Type:	Pure
Function Visibility:	Internal
Function Modifiers:	N/A
Return Type:	String
Return Details:	Returns the length of the supplied address

Code

The code for the *addressLength* is as follows:

Listing 5: **addressLength Code**

```
function addressLength(address input) internal pure returns (uint) {
    string memory addressStr = toString(input);
    return bytes(addressStr).length;
}
```

The *addressLength* function performs the following:

- Converts the supplied address to a string
- returns the byte length of the string

Usage

The *addressLength* function has the following usage syntax and arguments:

	Argument	Type	Details
1	input	address	Address to calculate the length of

Listing 6: **addressLength Usage Example**

```
addressLength(0x123456789abcdefghijklmnopqrstuvwxyz98765);
```

CHAPTER 19

TruMintableToken

Title:	TruMintableToken
Description:	Smart Contract derived from MintableToken by Zeppelin Solidity with additional functionality for the **TruReputationToken**.
Author:	Ian Bray, Tru Ltd; derived from MintableToken
Solidity Version:	^0.4.18
Relative Path:	`./contracts/supporting/TruMintableToken.sol`
License:	Apache 2 License
Current Version:	0.1.12
Original Source:	MintableToken

19.1 1. Imports & Dependencies

The following imports and dependencies exist for the *TruMintableToken* Smart Contract:

Name	Description
SafeMath	Zeppelin Solidity Library to perform mathematics safely inside Solidity
TruAddress	Solidity Library of helper functions surrounding the Address type in Solidity.
ReleaseableToken	Token Market Contract that allows control over when a Token can be released.

19.2 2. Variables

The following variables exist for the *TruMintableToken* Smart Contract:

Variable	Type	Vis	Details
mintingFinished	bool	public	Variable to mark if minting is finished for this token **Default**: *false*
preSaleComplete	bool	public	Variable to mark if the Pre-Sale is complete for this **Default**: *false*
saleComplete	bool	public	Variable to mark if the CrowdSale is complete for this **Default**: *false*

19.3 3. Enums

There are no enums for the *TruMintableToken* Smart Contract.

19.4 4. Events

The following events for the *TruMintableToken* Smart Contract:

Name	Description
Minted	Event to track when tokens are minted
MintFinished	Event to notify when minting is finalised and finished
PreSaleComplete	Event to notify when a Pre-Sale is complete
SaleComplete	Event to notify when a CrowdSale is complete

19.4.1 Minted

Event Name:	Minted
Description:	Event to track when tokens are minted

Usage

The *Minted* event has the following usage syntax and arguments:

	Argument	Type	Indexed?	Details
1	_to	address	Yes	Address tokens have been minted to
2	_amount	uint256	No	Amount of tokens minted

Listing 1: **Minted Usage Example**

```
Minted(0x123456789abcdefghijklmnopqrstuvwxyz98765, 100);
```

19.4.2 MintFinished

Event Name:	MintFinished
Description:	Event to notify when minting is finalised and finished

Usage

The *MintFinished* event has the following usage syntax and arguments:

	Argument	Type	Indexed?	Details
1	_executor	address	Yes	Address that executed the *MintFinished* event

Listing 2: **MintFinished Usage Example**

```
MintFinished(0x123456789abcdefghijklmnopqrstuvwxyz98765);
```

19.4.3 PreSaleComplete

Event Name:	PreSaleComplete
Description:	Event to notify when a Pre-Sale is complete

Usage

The *PreSaleComplete* event has the following usage syntax and arguments:

	Argument	Type	Indexed?	Details
1	_executor	address	Yes	Address that executed the *PreSaleComplete* event

Listing 3: **PreSaleComplete Usage Example**

```
PreSaleComplete(0x123456789abcdefghijklmnopqrstuvwxyz98765);
```

19.4.4 SaleComplete

Event Name:	SaleComplete
Description:	Event to notify when a CrowdSale is complete

Usage

The *SaleComplete* event has the following usage syntax and arguments:

	Argument	Type	Indexed?	Details
1	_executor	address	Yes	Address that executed the *SaleComplete* event

Listing 4: **SaleComplete Usage Example**

```
SaleComplete(0x123456789abcdefghijklmnopqrstuvwxyz98765);
```

19.5 5. Mappings

There are no mappings for the *TruMintableToken* Smart Contract.

19.6 6. Modifiers

The following modifiers exist for the *TruMintableToken* Smart Contract:

Name	Description
canMint	Modifier to check the Token can mint

19.6.1 canMint

Modifier Name:	canMint
Description:	Modifier to check if minting has finished for this token or not

Code

The code for the *canMint* modifier is as follows:

Listing 5: **canMint Code**

```
modifier canMint() {
    require(!mintingFinished);
    _;
}
```

The *canMint* function performs the following:

- Checks that the *mintingFinished* variable is false otherwise it throws

19.7 7. Functions

The following functions exist for the *TruMintableToken* Smart Contract:

Name	Description
mint	Function to mint tokens
finishMinting	Function to stop minting new tokens.

19.7.1 mint

Function Name:	mint
Description:	Function to mint tokens
Function Type:	Pure
Function Visibility:	Public
Function Modifiers:	*onlyOwner*, *canMint*
Return Type:	Bool
Return Details:	Returns whether mint completed successfully

Code

The code for the *mint* function is as follows:

Listing 6: **mint Code**

```
function mint(address _to, uint256 _amount) public onlyOwner canMint returns (bool) {
    require(_amount > 0);
    require(TruAddress.isValid(_to) == true);

    totalSupply = totalSupply.add(_amount);
    balances[_to] = balances[_to].add(_amount);
    Minted(_to, _amount);
    Transfer(0x0, _to, _amount);
    return true;
}
```

The *mint* function performs the following:

- Checks the supplied *_amount* is greater than 0
- Checks the supplied *_to* address is valid
- Adds the newly minted amount to the totalSupply of tokens
- Transfers the newly minted tokens to the recipient
- Fires the *Minted* event
- Fires the *Transfer* event
- returns true

Usage

The *mint* function has the following usage syntax and arguments:

	Argument	Type	Details
1	_to	address	Address to mint tokens to
2	_amount	uint256	Amount of tokens to mint

Listing 7: **mint Usage Example**

```
mint(0x123456789abcdefghijklmnopqrstuvwxyz98765);
```

19.7.2 finishMinting

Function Name:	finishMinting
Description:	Function to mint tokens
Function Type:	Pure
Function Visibility:	Public
Function Modifiers:	*onlyOwner*, *canMint*
Return Type:	Bool
Return Details:	Returns whether mint completed successfully

Code

The code for the *finishMinting* function is as follows:

Listing 8: **finishMinting Code**

```
function finishMinting(bool _presale, bool _sale) public onlyOwner returns (bool) {
    require(_sale != _presale);

    if (_presale == true) {
        preSaleComplete = true;
        PreSaleComplete();
        return true;
    }

    require(preSaleComplete == true);
    saleComplete = true;
    SaleComplete();
    mintingFinished = true;
    MintFinished();
    return true;
}
```

The *finishMinting* function performs the following:

- Ensures that the *_presale* and *_sale* argument do not match (one must be true, the other false)
- If _presale argument is true, change the *preSaleComplete* variable to true, fire the *PreSaleComplete* event and return true
- If _sale argument is true, change the *saleComplete* variable to true, fire the *SaleComplete* event, set the *mintingFinished* variable to true, fire the *MintFinished* event and return true

Usage

The *finishMinting* function has the following usage syntax and arguments:

	Argument	Type	Details
1	_presale	bool	Whether this call is from the Pre-Sale or not
2	_sale	bool	Whether this call is from the CrowdSale or not

Listing 9: **finishMinting Usage Example**

```
finishMinting(true, false);
```

CHAPTER 20

TruUpgradeableToken

Title:	TruUpgradeableToken
Description:	Smart Contract derived from UpgradeableToken by Token Market with additional functionality for the **TruReputationToken**.
Author:	Ian Bray, Tru Ltd
Solidity Version:	0.4.18
Relative Path:	`./contracts/supporting/TruUpgradeableToken.sol`
License:	Apache 2 License
Current Version:	0.1.12
Original Source:	UpgradeableToken

20.1 1. Imports & Dependencies

The following imports and dependencies exist for the *TruUpgradeableToken* Solidity Library:

Name	Description
SafeMath	Zeppelin Solidity Library to perform mathematics safely inside Solidity
StandardToken	Zeppelin Solidity Smart Contract for a Standard ERC-20 Token
TruUpgradeableToken	Library of helper functions surrounding the Solidity Address type
UpgradeAgent	Token Market Smart Contract used to facilitate upgrading of tokens

20.2 2. Variables

The following variables exist for the *TruUpgradeableToken* Smart Contract:

Variable	Type	Vis	Details
upgradeMaster	address	public	Variable containing the address of the wallet designated as the Upgrade Master
upgradeAgent	UpgradeAgent	public	Variable containing the UpgradeAgent Contract Instance
totalUpgraded	uint256	public	Variable to track the number of tokens that have been upgraded

20.3 3. Enums

The following enums exist for the *TruUpgradeableToken* Solidity Library:

Enum	Description
UpgradeState	Enum of the different states an UpgradeableToken can be in.

20.3.1 UpgradeState

The following enum states exist for the *UpgradeState* enum:

Enum States	Detail
Unknown	Token upgrade is in an Unknown State- fallback state not used
NotAllowed	The child contract has not reached a condition where the upgrade can begin
WaitingForAgent	Token allows upgrade, but an *upgradeAgent* has not been set
ReadyToUpgrade	The *upgradeAgent* is set, but no tokens have been upgraded yet
Upgrading	The *upgradeAgent* is set, and balance holders can upgrade their tokens

20.4 4. Events

The following events exist for the *TruUpgradeableToken* Solidity Library:

Name	Description
Upgrade	Event to notify when a token holder upgrades their tokens
UpgradeAgentSet	Event to notify when an upgradeAgent is set
NewUpgradedAmount	Event to notify the new total number of tokens that have been upgraded

20.4.1 Upgrade

Event Name:	Upgrade
Description:	Event to notify when a token holder upgrades their tokens

Usage

The *Upgrade* event has the following usage syntax and arguments:

	Argument	Type	Indexed?	Details
1	from	address	Yes	Source wallet that the older tokens are sent from
2	to	address	Yes	Address of the destination for upgraded tokens which is hardcoded to the *upgradeAgent* who sends them back to the originating address
3	upgradeValue	uint256	No	Number of tokens to upgrade

Listing 1: **Upgrade Usage Example**

```
Upgrade(0x123456789abcdefghijklmnopqrstuvwxyz98765,
        0x123456789abcdefghijklmnopqrstuvwxyz01234,
        100);
```

20.4.2 UpgradeAgentSet

Event Name:	UpgradeAgentSet
Description:	Event to notify when an upgradeAgent is set

Usage

The *UpgradeAgentSet* event has the following usage syntax and arguments:

	Argument	Type	Indexed?	Details
1	agent	address	Yes	Address of new *upgradeAgent*
2	executor	address	Yes	Address that executed the *UpgradeAgentSet* event

Listing 2: **UpgradeAgentSet Usage Example**

```
UpgradeAgentSet(0x123456789abcdefghijklmnopqrstuvwxyz98765,
                0x123456789abcdefghijklmnopqrstuvwxyz01234);
```

20.4.3 NewUpgradedAmount

Event Name:	NewUpgradedAmount
Description:	Event to notify when an upgradeAgent is set

Usage

The *NewUpgradedAmount* event has the following usage syntax and arguments:

	Argument	Type	Indexed?	Details
1	originalBalance	uint256	No	Balance of Upgrade Tokens before
2	newBalance	uint256	No	Balance of Upgrade Tokens after
3	executor	address	Yes	Address that executed the *NewUpgradedAmount* event

Listing 3: **NewUpgradedAmount Usage Example**

```
NewUpgradedAmount(50, 100);
```

20.5 5. Mappings

There are no mappings for the *TruUpgradeableToken* Smart Contract.

20.6 6. Modifiers

The following modifiers exist for the *TruUpgradeableToken* Smart Contract:

Name	Description
onlyUpgradeMaster	Modifier to check the Upgrade Master is executing this call

20.6.1 onlyUpgradeMaster

Modifier Name:	onlyUpgradeMaster
Description:	Modifier to check the Upgrade Master is executing this call

Code

The code for the *onlyUpgradeMaster* modifier is as follows:

Listing 4: **onlyUpgradeMaster Code**

```
modifier onlyUpgradeMaster() {
    require(msg.sender == upgradeMaster);
    _;
}
```

The *onlyUpgradeMaster* function performs the following:

- Checks that the *msg.sender* matches the *upgradeMaster* variable

20.7 7. Functions

The following functions exist for the *TruUpgradeableToken* Smart Contract:

Name	Description
TruUpgradeableToken Constructor	Constructor for the *TruUpgradeableToken* Smart Contract
upgrade	Function to upgrade tokens.
setUpgradeAgent	Function to set the *upgradeAgent* variable
getUpgradeState	Function to get the current *UpgradeState* for the token
setUpgradeMaster	Function to change the *upgradeMaster* variable
canUpgrade	Function to get whether the token can be upgraded

20.7.1 TruUpgradeableToken Constructor

Function Name:	TruUpgradeableToken
Description:	Constructor for the *TruUpgradeableToken* Smart Contract
Function Type:	Constructor
Function Visibility:	Public
Function Modifiers:	N/A
Return Type:	None
Return Details:	N/A

Code

The code for the *TruUpgradeableToken Constructor* function is as follows:

Listing 5: **TruUpgradeableToken Constructor Code**

```
function TruUpgradeableToken(address _upgradeMaster) public {
    require(TruAddress.isValid(_upgradeMaster) == true);
    upgradeMaster = _upgradeMaster;
}
```

The *TruUpgradeableToken Constructor* function performs the following:

- Checks the *_upgradeMaster* is a valid Ethereum address.
- Sets the *upgradeMaster* variable to the *_upgradeMaster* argument value.

Usage

The *TruUpgradeableToken Constructor* function has the following usage syntax and arguments:

	Argument	Type	Details
1	_upgradeMaster	address	Address to be set as the Upgrade Master

Listing 6: **TruUpgradeableToken Constructor Usage Example**

```
TruUpgradeableToken(0x123456789abcdefghijklmnopqrstuvwxyz98765);
```

20.7.2 upgrade

Function Name:	upgrade
Description:	Function to upgrade tokens
Function Type:	N/A
Function Visibility:	Public
Function Modifiers:	N/A
Return Type:	None
Return Details:	N/A

Code

The code for the *upgrade* function is as follows:

Listing 7: **upgrade Code**

```
function upgrade(uint256 value) public {
    UpgradeState state = getUpgradeState();
    require((state == UpgradeState.ReadyToUpgrade) || (state == UpgradeState.
↪Upgrading));
    require(value > 0);
    require(balances[msg.sender] >= value);

    uint256 upgradedAmount = totalUpgraded.add(value);
    assert(upgradedAmount >= value);

    uint256 senderBalance = balances[msg.sender];
    uint256 newSenderBalance = senderBalance.sub(value);
    uint256 newTotalSupply = totalSupply.sub(value);
    balances[msg.sender] = newSenderBalance;
    totalSupply = newTotalSupply;
    NewUpgradedAmount(totalUpgraded, newTotalSupply);
    totalUpgraded = upgradedAmount;
    // Upgrade agent reissues the tokens
    upgradeAgent.upgradeFrom(msg.sender, value);
    Upgrade(msg.sender, upgradeAgent, value);
}
```

The *upgrade* function performs the following:

- Checks the *UpgradeState* is either ReadyToUpgrade or Upgrading
- Checks the upgrade amount *value* is greater than zero
- Checks that the send has a balance of greater than or equal to the upgrade *value*
- Adds the *value* to the *totalUpgraded* variable and checks that this new value is equal to or greater than the *value* to be upgraded.
- Removes the *value* from the senders balance

- Removes the *value* from the token's totalSupply
- Fires the *NewUpgradedAmount* event
- Initiates the Upgrade Agent's upgradeFrom functionality to deliver the *value* in upgraded tokens to the sender.
- Fires the *Upgrade* event

Usage

The *upgrade* function has the following usage syntax and arguments:

	Argument	Type	Details
1	_value	uint256	Amount of tokens to be upgraded

Listing 8: **upgrade Usage Example**

```
upgrade(100);
```

20.7.3 setUpgradeAgent

Function Name:	setUpgradeAgent
Description:	Function to set the *upgradeAgent* variable
Function Type:	N/A
Function Visibility:	Public
Function Modifiers:	*onlyUpgradeMaster*
Return Type:	None
Return Details:	N/A

Code

The code for the *setUpgradeAgent* function is as follows:

Listing 9: **setUpgradeAgent Code**

```
function setUpgradeAgent(address _agent) public onlyUpgradeMaster {
    require(TruAddress.isValid(_agent) == true);
    require(canUpgrade());
    require(getUpgradeState() != UpgradeState.Upgrading);

    UpgradeAgent newUAgent = UpgradeAgent(_agent);

    require(newUAgent.isUpgradeAgent());
    require(newUAgent.originalSupply() == totalSupply);

    UpgradeAgentSet(upgradeAgent);

    upgradeAgent = newUAgent;
}
```

The *setUpgradeAgent* function performs the following:

- Checks the *_agent* address is valid. If not, the function will throw.

- Checks that the token can upgrade via the *canUpgrade* function. If not, the function will throw.
- Checks that that *UpgradeState* is not *Upgrading* (and therefore in the middle of an upgrade). If not, the function will throw.
- **Checks that the specified Upgrade Agent contract is an Upgrade Agent. If not, the function will** throw.
- Checks that the Upgrade Agent's original supply matches the current total supply of the token. If not, the function will throw.
- Fires the *UpgradeAgentSet* event.
- Sets the *upgradeAgent* variable.

Usage

The *setUpgradeAgent* function has the following usage syntax and arguments:

	Argument	Type	Details
1	_agent	address	Address of the new Upgrade Agent	

Listing 10: **setUpgradeAgent Usage Example**

```
setUpgradeAgent(0x123456789abcdefghijklmnopqrstuvwxyz98765);
```

20.7.4 getUpgradeState

Function Name:	getUpgradeState
Description:	Function to get the current *UpgradeState* of the token
Function Type:	Constant
Function Visibility:	Public
Function Modifiers:	N/A
Return Type:	UpgradeState
Return Details:	Returns UpgradeState as a uint (0, 1, 2, 3 or 4)

Code

The code for the *getUpgradeState* function is as follows:

Listing 11: **getUpgradeState Code**

```
function getUpgradeState() public constant returns(UpgradeState) {
    if (!canUpgrade())
        return UpgradeState.NotAllowed;
    else if (TruAddress.isValid(upgradeAgent) == false)
        return UpgradeState.WaitingForAgent;
    else if (totalUpgraded == 0)
        return UpgradeState.ReadyToUpgrade;
    else
        return UpgradeState.Upgrading;
}
```

The *getUpgradeState* function performs the following:

- the *canUpgrade* function to see if it is true. If it is false, returns NotAllowed *UpgradeState*
- Checks the *upgradeAgent* address is valid and set. If it is not, returns WaitingForAgent *UpgradeState*
- Checks that the *totalUpgraded** is zero. If it is true, return ReadyToUpgrade *UpgradeState*
- Else return Upgrading *UpgradeState*

Usage

The *getUpgradeState* function has the following usage syntax:

Listing 12: **getUpgradeState Usage Example**

```
getUpgradeState();
```

20.7.5 setUpgradeMaster

Function Name:	setUpgradeMaster
Description:	Function to change the *upgradeMaster* variable
Function Type:	N/A
Function Visibility:	Public
Function Modifiers:	*onlyUpgradeMaster*
Return Type:	UpgradeState
Return Details:	Returns UpgradeState as a uint (0, 1, 2, 3 or 4)

Code

The code for the *setUpgradeMaster* function is as follows:

Listing 13: **setUpgradeMaster Code**

```
function setUpgradeMaster(address _master) public onlyUpgradeMaster {
    require(TruAddress.isValid(_master) == true);
    upgradeMaster = _master;
}
```

The *setUpgradeMaster* function performs the following:

- Checks the *_master* argument is a valid Ethereum Address. If it is not, it will throw.
- Sets the *upgradeMaster* variable to the *_master* argument.

Usage

The *setUpgradeMaster* function has the following usage syntax and arguments:

	Argument	Type	Details
1	_master	address	Address of the new Upgrade Master

Listing 14: **setUpgradeAgent Usage Example**

```
setUpgradeMaster(0x123456789abcdefghijklmnopqrstuvwxyz98765);
```

20.7.6 canUpgrade

Function Name:	canUpgrade
Description:	Function to get whether the token can be upgraded or not
Function Type:	Constant
Function Visibility:	Public
Function Modifiers:	N/A
Return Type:	bool
Return Details:	Returns true as a default; customised in child contracts to fit required conditions

Code

The code for the *canUpgrade* function is as follows:

Listing 15: **canUpgrade Code**

```
function canUpgrade() public constant returns(bool) {
    return true;
}
```

The *canUpgrade* function performs the following:

- returns true. This functionality is overridden in child contracts to provide conditionality for this result.

Usage

The *canUpgrade* function has the following usage syntax:

Listing 16: **getUpgradeState Usage Example**

```
canUpgrade();
```

CHAPTER 21

UpgradeAgent

Title:	UpgradeAgent
Description:	Contract interface derived from **UpgradeAgent** by Token Market
Author:	TokenMarket Ltd/Updated by Ian Bray, Tru Ltd
Solidity Version:	^0.4.18
Relative Path:	`./contracts/supporting/UpgradeAgent.sol`
License:	Apache 2 License
Current Version:	0.1.12
Original Source:	UpgradeAgent Source

21.1 1. Imports & Dependencies

There are no imports or dependencies for the *UpgradeAgent* Smart Contract.

21.2 2. Variables

The following variables exist for the *UpgradeAgent* Smart Contract:

Variable	Type	Vis	Details
originalSupply	uint256	public	Variable containing the original token count of the the pre-upgrade token

21.3 3. Enums

There are no enums for the *UpgradeAgent* Smart Contract.

21.4 4. Events

There are no events for the *UpgradeAgent* Smart Contract.

21.5 5. Mappings

The are no mappings for the *UpgradeAgent* Smart Contract.

21.6 6. Modifiers

There are no modifiers for the *UpgradeAgent* Smart Contract.

21.7 7. Functions

The following functions exist for the *UpgradeAgent* Smart Contract:

Name	Description
isUpgradeAgent	Interface function for checking if an UpgradeAgent
upgradeFrom	Interface function for upgrading tokens

21.7.1 isUpgradeAgent

Function Name:	isUpgradeAgent
Description:	Interface function for checking if an UpgradeAgent
Function Type:	Pure
Function Visibility:	Public
Function Modifiers:	N/A
Return Type:	bool
Return Details:	returns whether an UpgradeAgent or not

Code

The code for the *isUpgradeAgent* function is an interface and it is defined as follows:

Listing 1: **isUpgradeAgent Code**

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

Usage

The *isUpgradeAgent* function has the following usage syntax:

Listing 2: **isUpgradeAgent Usage Example**

```
isUpgradeAgent();
```

21.7.2 upgradeFrom

Function Name:	upgradeFrom
Description:	Interface function to upgrade from one token to another
Function Type:	Pure
Function Visibility:	Public
Function Modifiers:	N/A
Return Type:	bool
Return Details:	returns whether an UpgradeAgent or not

Code

The code for the *upgradeFrom* function is an interface and it is defined as follows:

Listing 3: **isUpgradeAgent Code**

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

Usage

The *upgradeFrom* function has the following usage syntax and arguments:

	Argument	Type	Details
1	_from	address	Address to transfer upgrade tokens from
2	_value	uint256	Amount of tokens to upgrade

Listing 4: **upgradeFrom Usage Example**

```
upgradeFrom(0x123456789abcdefghijklmnopqrstuvwxyz98765, 100);
```

CHAPTER 22

Acknowledgments

Tru Ltd would like the make the following acknowledgments:

22.1 Open Zepplin

The Tru Reputation Token Project makes extensive use and has been inspired by the Zeppelin Solidity by Open Zeppelin. Specifically the following Smart Contracts and Libraries are used by the Tru Reputation Token:

Name	Modified?
BasicToken.sol	No
ERC20.sol	No
ERC20Basic.sol	No
MintableToken.sol	Yes
Ownable.sol	No
SafeMath.sol	No
StandardToken.sol	No

To ensure security, and as part of good community practice, the coverage testing in this Repository covers all non-trivial libraries consumed from the Zeppelin Solidity framework, and will feedback any issues encountered with the framework during any and all testing.

All Open Zeppelin Smart Contracts, libraries and supporting functionality used within this work are licensed under the MIT License.

22.2 TokenMarket

All TokenMarket Smart Contracts, libraries and supporting functionality used within this work are licensed under the Apache 2.0 License. The following items are covered by these terms:

Name	Modified?
Haltable.sol	Yes
ReleasableToken.sol	Yes
UpgradeableToken.sol	Yes
UpgradeAgent.sol	Yes

The original unmodified source files are under copyright of **TokenMarket Ltd** and can be obtained in the TokenMarket ICO Github Repository

CHAPTER 23

Useful Links

23.1 Solidity Links

We at Tru Ltd have found the following links and resources insightful and useful during the development of this project and hope you also find utility from them:

- Solidity Documentation
- Github Solidity Repo
- Zeppelin-Solidity Documentation
- Truffle Framework Documentation

CHAPTER 24

Contact Information

Feel free to contact us directly using the following channels:

Tru Reputation Protocol Sub-Reddit

 Tru Reputation Protocol Slack Community

 Tru Ltd

Tru Reputation Protocol Telegram Group Chat

Tru Reputation Token

CHAPTER 25

Contribution Guidelines

Whilst this project has been specifically crafted for Tru Ltd's needs, we encourage everyone to report any bugs found - including documentation issues - via Tru Reputation Token Github Issues Page

Please feel free to fork and modify the code as per the Apache 2 License.

CHAPTER 26

Legal Notice

Tru Ltd is registered in England and Wales, No. 09659526