
SPECIFICATION OF THE STOKR SMART CONTRACTS

SICOS

CONTENTS

1 Abstract	6
2 Ownable Contracts	7
2.1 Purpose	7
2.2 Functionality	7
2.2.1 Ownership Transfer	7
2.2.2 Ownership Claiming	7
3 Whitelist Contract	8
3.1 Purpose	8
3.2 Roles	8
3.2.1 Whitelist Owner	8
3.2.2 Whitelist Admins	8
3.2.3 Whitelisted Investors	9
3.3 Functionality	9
3.3.1 Ownership	9
3.3.2 Admin Management	9
3.3.3 Investor Management	10
4 Crowdsale Contract	11
4.1 Purpose	11
4.2 Roles	11
4.2.1 Crowdsale Owner	11
4.2.2 Investors	11
4.2.3 Rate Source	12
4.2.4 Token	12
4.2.5 Company Wallet	13
4.2.6 Reserve Account	13
4.3 Functionality	13
4.3.1 Ownership	13
4.3.2 Public vs. Private Sale	14
4.3.3 Purchase With Ether	15
4.3.4 Softcap / Sale Goal	16
4.3.5 Change of the Opening Time	17

4.3.6	Change of the Closing Time	17
4.3.7	Finalization	18
4.3.8	Refunding	19
4.4	Lifecycle	20
4.4.1	Before Start of Public Sale Phase	20
4.4.2	Public Sale Phase	20
4.4.3	Between End of Public Sale Phase and Finalization	20
4.4.4	After Finalization	21
5	Token Contract	22
5.1	Purpose	22
5.2	Roles	22
5.2.1	Token Owner	22
5.2.2	Token Minter	22
5.2.3	Whitelist	23
5.2.4	Profit Depositor	23
5.2.5	Profit Distributor	24
5.2.6	Token Recoverer	24
5.2.7	Token Holders	24
5.2.8	Third Party Trustees	25
5.3	Functionality	26
5.3.1	Ownership	26
5.3.2	Role Management	26
5.3.3	Whitelist Checks	27
5.3.4	Minting and Fixing Total Supply	27
5.3.5	Profit Sharing	28
5.3.6	Token Transfers	30
5.3.7	Token Recovery	31
5.3.8	Destruction	32
5.4	Lifecycle	32
5.4.1	Minting Phase	32
5.4.2	Trading Phase	33
6	Project Manager	34
6.1	Purpose	34

6.2	Roles	34
6.2.1	Project Manager Owner	34
6.2.2	Whitelist	34
6.2.3	Token Factory	35
6.2.4	Crowdsale Factory	35
6.2.5	Rate Admin	36
6.3	Functionality	36
6.3.1	Role Management	36
6.3.2	Project Creation	37
6.3.3	Rate Source	37

1 ABSTRACT

STOKR is as crowd-investing platform based on smart contracts on the [Ethereum](#) blockchain. On *STOKR* many projects will be deployed with the same smart contract structure (see [project manager](#)). There will be a shared [whitelist](#) for all projects on *STOKR*. Each project has its own [ERC20 Token](#) and its [crowdsale](#) contract instance. The token contract is able to distribute profits generated by the projects to the token holders.

2 OWNABLE CONTRACTS

2.1 PURPOSE

All of the following contracts are ownable, that is they have an *owner* address—either another contract instance or an externally owned account—assigned which has special permissions on the contract. An owner is initially (i.e. upon deployment) the contract deployer, but may [transfer the ownership](#) to another address.

2.2 FUNCTIONALITY

This functionality is shared by all following contracts.

2.2.1 OWNERSHIP TRANSFER

Transfer the contract instance's ownership to a new address.

Note: The actual ownership transfer doesn't happen immediately. The designated new owner has to [claim](#) first, to become the effective new owner. Thus, transferring the ownership to an invalid address is avoided.

Function

```
transferOwnerShip(address)
```

Restrictions

only by current owner

Emitted events

```
OwnershipTransferred(address, address)
```

2.2.2 OWNERSHIP CLAIMING

Claiming the ownership.

Function

```
claimOwnership()
```

Restrictions

only by the designated newOwner

3 WHITELIST CONTRACT

3.1 PURPOSE

Provide a central list of investors' [Ethereum](#) addresses, who are allowed to buy or sell tokens.

3.2 ROLES

3.2.1 WHITELIST OWNER

Initially

the [Whitelist](#) contract deployer

Number

one at a time

Assignment

- by deploying the contract
- by gaining ownership from previous owner

Permissions

- [transfer ownership](#) to another account
- assign/unassign [whitelist admins](#)

3.2.2 WHITELIST ADMINS

Authority which manages the whitelist.

Initially

none

Number

zero or more

Assignment

by getting [added or removed](#) by the [whitelist owner](#)

Permissions

add or remove [investors](#) to/from the [whitelist](#)

3.2.3 WHITELISTED INVESTORS

Addresses of investors, who have completed and passed the KYC process.

Initially

none

Number

zero or more

Assignment

by getting [added or removed](#) from the [whitelist](#) by a [whitelist admin](#)

Permissions

none

3.3 FUNCTIONALITY

3.3.1 OWNERSHIP

see [Ownable](#)

3.3.2 ADMIN MANAGEMENT

ADDING ADMINS

A single whitelist admin can be added.

Function

`addAdmin(address)`

Restrictions

only by [whitelist owner](#)

Emitted events

`AdminAdded(address)`

REMOVING ADMINS

A single whitelist admin can be removed.

Function

`removeAdmin(address)`

Restrictions

only by [whitelist owner](#)

Emitted events

AdminRemoved(address)

3.3.3 INVESTOR MANAGEMENT

WHITELISTING INVESTORS

Several investors at once can be added to the whitelist.

Function

addToWhitelist(address[])

Restrictions

only by a [whitelist admin](#)

Emitted events

InvestorAdded(address, address)

UNWHITELISTING INVESTORS

Several investors at once can be removed from the whitelist.

Function

removeFromWhitelist(address[])

Restrictions

only by a [whitelist admin](#)

Emitted events

InvestorRemoved(address, address)

4 CROWDSALE CONTRACT

4.1 PURPOSE

Enable token purchase by investors and check if the sale has to be considered a success or a failure. Refund investors upon sale failure.

4.2 ROLES

4.2.1 CROWDSALE OWNER

Authority who administers the crowdsale.

Initially

the [Crowdsale](#) contract deployer

Number

one at a time

Assignment

- by deploying the contract
- by gaining ownership from previous owner

Permissions

- [change the time schedule](#) of the public offering
- [transfer ownership](#) to another account
- distribute tokens to [investors](#) who paid in fiat currency (e.g. EUR) off-chain
- [finalize](#) the sale, but only
 1. once
 2. after the [public sale](#) has [closed](#)

4.2.2 INVESTORS

Addresses who purchase tokens either via public or private sale.

Initially

all accounts that are whitelisted in the corresponding [whitelist](#)

Number

zero or more

Assignment

by being added or removed to/from the corresponding [whitelist](#)

Permissions

- [purchase tokens](#) with Ether in public sale, but only
 1. while the sale is [open](#)
 2. if there are enough tokens available for [public sale](#)
 3. if the amount of newly purchased tokens is at least a [predefined minimum](#)
- [claim refunds](#), thus withdraw invested Ether, but only
 1. after [finalization](#)
 2. if the [sale goal](#) was missed

4.2.3 RATE SOURCE

External contract which delivers the actual price of an Ether in Euro cents.

Initially

given to constructor

Number

one

Assignment

only once upon contract deployment

Permissions

none

4.2.4 TOKEN

Reference to the [token](#) instance which is sold by this crowdsale.

Initially

given to constructor

Number

one

Assignment

only once upon contract deployment

Permissions

none

4.2.5 COMPANY WALLET

Address which will receive all Ether that were paid by [investors](#) to [buy tokens](#) during [public sale](#) if the [goal](#) was reached. This is meant to be a multisig wallet.

Initially

given to constructor

Number

one

Assignment

only once upon contract deployment

Permissions

none

4.2.6 RESERVE ACCOUNT

Address which will receive some additional amount of tokens (depending on the total amount of sold tokens) upon [finalization](#) of a [successful sale](#). This is meant to be some vesting contract address.

Initially

given to constructor

Number

one

Assignment

only once upon contract deployment

Permissions

none

4.3 FUNCTIONALITY

4.3.1 OWNERSHIPsee [Ownable](#)

4.3.2 PUBLIC VS. PRIVATE SALE

The crowdsale is divided into two distinct sales—a public and a private sale—each being limited by its own *cap*. The cap is the maximum total amount of tokens that can be bought in the respective sale.

Both caps are given to the constructor upon deployment (parameters `tokenCapOfPublicSale` and `tokenCapOfPrivateSale`) in token units (quantity of 10^{18} tokens).

[Whitelisted investors](#) can purchase tokens in public or private sale off-chain by paying in fiat currency (e.g. Euro). To make those investors receive their tokens, the crowdsale instance's [owner](#) has to [distribute](#) them manually.

Additionally, as long as the public crowdsale is [open](#), investors can [purchase](#) tokens directly by sending Ether to the crowdsale instance.

TOKEN DISTRIBUTION

[Whitelisted investors](#) can purchase tokens in public or private sale off-chain by paying in fiat currency (e.g. Euro). To make these investors receive their tokens, the crowdsale instance's [owner](#) has to distribute them manually via one of the two token distribution methods.

Note From the crowdsale smart contract's point of view the only distinction between public or private sale is, which pool of available tokens the newly minted tokens will be taken from.

Token distribution via public sale:

Function

```
distributeTokensViaPublicSale(address[], uint[])
```

Restrictions

- only by [owner](#)
- only if crowdsale wasn't [finalized](#) yet
- only if sum of token amounts to distribute doesn't exceed the amount of remaining tokens for public sale

Emitted events

- `TokenDistribution(address, uint)`
- `Minted(address, uint)`
- `Transfer(0x0, address, uint)`

Token Distribution via private sale:

Function

```
distributeTokensViaPrivateSale(address[], uint[])
```

Restrictions

- only by [owner](#)
- only if crowdsale wasn't [finalized](#) yet
- only if sum of token amounts to distribute doesn't exceed the amount of remaining tokens for private sale

Emitted events

- `TokenDistribution(address, uint)`
- `Minted(address, uint)`
- `Transfer(0x0, address, uint)`

4.3.3 PURCHASE WITH ETHER

As long as the public crowdsale is [open](#), investors can [purchase](#) tokens directly by sending Ether to the crowdsale instance.

PURCHASE MINIMUM

When paying with Ether, the amount of tokens bought within a transaction has to be at least some predefined lower limit which is given to constructor upon deployment (parameter `tokenPurchaseMinimum`) in token units (quantity of 10^{-18} tokens).

Companies may decide to not define a lower threshold by setting `tokenPurchaseMinimum` to zero.

PURCHASE LIMIT

During the start phase of the public sale, the total amount of tokens a single investor can buy with Ether may be limited (parameter `tokenPurchaseLimit`).

When the start phase has ended (parameter `limitEndTime`) investors may buy as much tokens as they want as long as the total cap for tokens purchasable in public sale was not reached.

TOKEN PRICE

The price of a token denominated in EUR-cents has to be given to the constructor upon deployment (parameter `tokenPrice`).

The token price does not depend on the current Ether price. Therefore an external contract—an instance of `RateSource`—is called upon every token purchase with Ether to get the current Ether rate and adjust accordingly.

The amount of bought tokens is calculated as follows:

Let be

- *value*: the sent Ether [in wei, i.e. 10^{-18} Ether],
- *rate*: the current Ether rate [in €-cent per Ether],
- *price*: the token price [in €-cent per token],
- *amount*: the purchased token amount [in token units, i.e. 10^{-18} tokens],

then

$$\text{amount} := \text{value} \times \text{rate} \div \text{price}$$

[Whitelisted investors](#) may purchase tokens via the [public sale](#) by sending Ether to the crowdsale instance.

Functions

- payable `buyTokens()`
- payable fallback function

Restrictions

- only by [whitelisted investors](#)
- only while the public sale is [open](#) that is from opening time till closing time
- only if amount of bought tokens in a single transaction is at least a predefined minimum purchase amount
- only if amount of bought tokens doesn't exceed the amount of remaining tokens for public sale

Emitted events

- `TokenPurchase(address, uint, uint)`
- `Minted(address, uint)`
- `Transfer(0x0, address, uint)`

4.3.4 SOFTCAP / SALE GOAL

Upon crowdsale deployment a predefined softcap (parameter `tokenGoal`) can be given to the constructor, which determines a goal, i.e. the minimum amount of sold

token in both—public and private—sales, that has to be reached for the crowdsale to become a success.

If the goal was reached, the sale is considered successful, otherwise it is considered a failure.

During sale period, as long as the goal wasn't reached, invested Ether are accumulated in the crowdsale contract. After the goal was reached, all invested Ether are transferred to the [company wallet](#) address

Companies may decide to not define a goal by setting `tokenGoal` to zero.

4.3.5 CHANGE OF THE OPENING TIME

The Opening time of the public crowdsale can be changed by the crowdsale [owner](#). This operation won't affect the duration of the offering sale as the closing time will be shifted, too.

Function

```
changeOpeningTime(uint)
```

Restrictions

- only by crowdsale [owner](#)
- only if the public sale has not started yet
- only if openingTime is in the future

Emitted events

- `OpeningTimeChange(uint, uint)`
- `ClosingTimeChange(uint, uint)`

4.3.6 CHANGE OF THE CLOSING TIME

The Closing time of the public crowdsale can be changed by the crowdsale [owner](#). This allows the crowdsale [owner](#) to decrease the offering period in case the crowdsale is already fully sold. It also allows to extend the offering period to a maximum of 183 days. In case the offering period is already longer than 183 days it can not be further extended.

Function

```
changeClosingTime(uint)
```

Restrictions

- only by crowdsale [owner](#)
- only before the public sale has closed
- only if closingTime is in the future
- only if offering period is longer than 0 afterwards
- only if offering period is not longer than 183 days afterwards

Emitted events

- ClosingTimeChange(uint, uint)

4.3.7 FINALIZATION

After the crowdsale has ended, it has to be finalized manually by the crowdsale [owner](#). Depending on whether the [sale goal](#) was reached, this will either

- if the goal was reached:
 1. mint additional tokens, i.e. a predefined percentage of total amount of sold tokens, for the benefit of the [reserve account](#)
 2. [finish minting](#) of the token, thus fixing its total supply
- if the goal was missed:
 1. destroy the [token](#) instance
 2. enable the [refunding](#) of investors who [paid by sending Ether directly](#) to the crowdsale instance.

Function

`finalize()`

Restrictions

- only by crowdsale [owner](#)
- only after the public sale has closed
- only once

Emitted events

Depending on the success of sale:

1. if goal was reached
 - Minted(address, uint)
 - Transfer(0x0, address, uint)
 - Finalization()

2. if goal was missed
 - Finalization()
 - TokenDestroyed()

4.3.8 REFUNDING

If the [sale goal](#) was not reached, the crowdsale is considered a failure and [investors](#) who [purchased](#) tokens by sending Ether directly to the crowdsale instance can claim a refund.

DISTRIBUTE REFUNDS

Anybody may distribute the refunds to investors, so that the latter don't need to withdraw them by themselves.

Function

```
distributeRefunds(address[])
```

Restrictions

- only if the crowdsale was [finalized](#)
- only if the [sale goal](#) was missed

Emitted events

```
InvestorRefund(address, uint)
```

CLAIMING REFUNDS

Investors may get their invested Ether back by claiming them by themselves.

Function

```
claimRefund()
```

Restrictions

- only if the crowdsale was [finalized](#)
- only if the [sale goal](#) was missed

Emitted events

```
InvestorRefund(address, uint)
```

4.4 LIFECYCLE

4.4.1 BEFORE START OF PUBLIC SALE PHASE

Start time

at deployment

End time

at openingTime

4.4.2 PUBLIC SALE PHASE

The public sale is open for [investors](#) who want to [buy token with Ether](#) from opening time till closing time given as Ethereum block timestamps (UNIX epoch) to constructor upon deployment (parameters openingTime, closingTime, limitEndTime).

During the start phase of a public sale the total amount of tokens a buyer may purchase can be limited:

Start time

at openingTime

End time

at limitEndTime

During the remaining time of a public sale the amount of tokens a buyer can purchase is only constrained by the total cap.

Start time

at limitEndTime

End time

at closingTime

A limitEndTime that lies either before openingTime or after closingTime is perfectly valid and either never or always limits the individual total amount of tokens bought with Ether.

4.4.3 BETWEEN END OF PUBLIC SALE PHASE AND FINALIZATION

After the public sale has closed, the crowdsale [owner](#) is expected to manually [finalize](#) the crowdsale. Before doing so, he/she must [distribute](#) tokens to [investors](#) who have

purchased them off-chain.

Start time

at `closingTime`

End time

upon call of `finalize()`

4.4.4 AFTER FINALIZATION

After [finalization](#) no tokens can be minted anymore, i.e. the [total token supply](#) is [fixed](#). Depending on whether the [sale was successful](#) or not, [token holder](#) either may trade their tokens and withdraw their profit shares, or [claim refunds](#).

Start time

upon call of `finalize()`

End time

never

5 TOKEN CONTRACT

5.1 PURPOSE

The Token contract is an [ERC20](#) compliant profit sharing token.

5.2 ROLES

5.2.1 TOKEN OWNER

Role administering authority.

Initially

the [Token](#) contract deployer

Number

one at a time

Assignment

- by deploying the contract
- by gaining ownership from previous owner

Permissions

- [transfer ownership](#) to another account
- assign [minter](#), but only once
- assign [profit depositor](#)
- assign [profit distributor](#)
- assign [token recoverer](#)

5.2.2 TOKEN MINTER

Authority who is allowed to [mint](#) some amount of tokens for the benefit of whitelisted investors and to finish minting, thus fixing the token's total supply.

Initially

none

Number

zero or one

Assignment

only once by the [token owner](#)

Note, this is expected to be a [Crowdsale](#) instance

Permissions

- [mint tokens](#) for the benefit of some investor, but only
 1. while minting was not finished
 2. the investor is whitelisted
- finish minting, but only once
- destroy the token contract

5.2.3 WHITELIST

A [Whitelist](#) contract instance which restricts the addresses who are able to send or receive tokens.

Initially

given to constructor

Number

one at a time

Assignment

by the [token owner](#)

Permissions

none

5.2.4 PROFIT DEPOSITOR

Authority who is able to [deposit company profits](#) (in Ether) into the token contract instance.

Initially

given to constructor

Number

one at a time

Assignment

by the [token owner](#)

Permissions

deposit profits, i.e. store some Ether amount into the Token instance

5.2.5 PROFIT DISTRIBUTOR

Authority who is able to [distribute profit shares](#) (in Ether) to token holders on their behalf, thus the latter don't need to withdraw their profit shares by themselves.

Initially

none

Number

one at a time

Assignment

by the [Token owner](#)

Permissions

distribute profit shares to [token holders](#), i.e. transfer some fraction of the Ether amount deposited in the token to investors according to their amount of tokens they hold

5.2.6 TOKEN RECOVERER

Authority who may [assign a new address](#) to a token holder's account.

Initially

given to constructor

Number

one at a time

Assignment

by the [Token owner](#)

Permissions

transfer investor account data from one address to another, but only if the new (destination) address was not already assigned to an investor's account

5.2.7 TOKEN HOLDERS

Addresses who hold some amount of tokens, thus have an account within the token contract instance.

Initially

none

Number

zero or more

Assignment

by first being [whitelisted](#) and then by purchasing tokens in the public or private [crowdsale](#) or receiving tokens transferred from another token holder

Permissions

- withdraw some fraction from profit deposited in the token, but only
 1. up to the amount of share he/she is owed
 2. if the [minting](#) was finished, i.e. the total supply of the [token doesn't change](#) anymore
- transfer some tokens to another [whitelisted investor](#)
- approve some third party trustee to transfer some amount of tokens from the token holder's account to another [whitelisted investor](#)

Note A token holder may not always be also a [whitelisted investor](#), because he/she may get removed from the [whitelist](#) after the receipt of tokens or because the token instance's [whitelist](#) was exchanged.

5.2.8 THIRD PARTY TRUSTEES

Addresses who are allowed by a token holder to transfer some limited amount of tokens on the token holder's behalf.

Initially

none

Number

zero or more

Assignment

by getting approved by a [token holder](#)

Permissions

transfer up to some allowed amount of tokens from the [token holder's](#) account to some other [whitelisted investor](#)

5.3 FUNCTIONALITY

5.3.1 OWNERSHIP

see [Ownable](#)

5.3.2 ROLE MANAGEMENT

The token instance's [owner](#) can set the addresses of the following authorities (roles):

1. [Minter](#), that is the related [crowdsale](#) contract instance

Function

`setMinter(address)`

Restrictions

- only by [owner](#)
- only once

Emitted events

none

2. [Profit depositor](#)

Function

`setProfitDepositor(address)`

Restrictions

only by [owner](#)

Emitted events

`ProfitDistributorChange(address)`

3. [Profit distributor](#)

Function

`setProfitDistributor(address)`

Restrictions

only by [owner](#)

Emitted events

`ProfitDepositorChange(address)`

4. [Token recoverer](#)

Function

`setTokenRecoverer(address)`

Restrictions

only by [owner](#)

Emitted events

`TokenRecovererChange(address, address)`

5.3.3 WHITELIST CHECKS

Prior to being able to send or receive any amount of tokens, both—the token sender and the token recipient—have to be [whitelisted investors](#) in the [Whitelist](#) instance that is assigned to the [token contract](#).

This holds true for all function related to [token minting](#) and [token transfers](#).

5.3.4 MINTING AND FIXING TOTAL SUPPLY

MINTING TOKENS

The token's total supply is initially not fixed. That means, the [minter](#) can create new tokens out of thin air and book them to any [whitelisted investor](#).

Function

`mint(address, uint)`

Restrictions

- only by [minter](#)
- if total supply wasn't fixed
- token recipient must be [whitelisted](#)

Emitted events

- `Minted(address, uint)`
- `Transfer(0x0, address, uint)`

FIXING TOTAL SUPPLY

As long as the total token supply is not fixed, token holders won't be able to withdraw their Ether share of profits deposited in the token contract instance. The minter will eventually finish the minting, thus fixing the total supply of the token. This will happen upon [crowdsale](#) finalization.

Function

`finishMinting()`

Restrictions

- only by [minter](#)
- if total supply wasn't fixed yet

Emitted events

MintFinished()

Note Fixing the token's total supply is *irreversible*.

5.3.5 PROFIT SHARING

PROFIT DEPOSIT

Profits are distributed to [token holder](#) by depositing Ether in the token contract, that is, the token's [profit depositor](#) authority sends Ether to the token contract instance.

Functions

- payable depositProfits()
- payable fallback function

Restrictions

- only by [profit depositor](#)
- only after the token's total supply was fixed
- only if the token's total supply is greater than zero

Emitted events

ProfitDeposit(address, uint)

PROFIT SHARE

Profits, i.e. the Ether currently stored in the token contract, are distributed to [token holders](#) according to their share of the token's total supply.

The token contract ensures that regardless of whether a token holder withdraws his/her profit share immediately after new profits were deposited or waits for several profit deposits to have happened, even if tokens were transferred in the meantime, the profit share will he/she can claim always reflects his/her token share.

Profit Share Calculation

The profit share of an investor is related to his/her token share:

Let be

- *totalSupply*: the total supply of tokens,

- $balance_{Inv}$: the amount of tokens held by the investor,
- $\Delta totalProfits$: the additional profits [in Ether] which were deposited into the token instance since the investor's last profit share withdrawal,
- $\Delta profitShare_{Inv}$: the owing profit share [in Ether], that is additional profit share since the investor's last profit share withdrawal,

then

$$balance_{Inv} \div totalSupply = \Delta profitShare_{Inv} \div \Delta totalProfits$$

Note A prerequisite to correct profit share calculation is a non-changing token's total supply. This means, profit shares won't get calculated and aren't withdrawable as long as the total supply wasn't fixed, that is, while the [crowdsale](#) is still ongoing.

Determining Owing Profit Shares

The currently owed profit share of an investor can be determined:

Function

`profitShareOwing(address)`

Restrictions

none, but until after the token's total supply was fixed (the [crowdsale](#) was finalized) this function won't calculate the correct value but return 0

Updating Individual Profit Shares

Whenever a profit share withdrawal or a token transfer takes place, the profit shares of the involved token holding parties have to be updated first. Profit share updating means, the owed profit share of a token holder up to this moment are calculated and saved.

Function

`updateProfitShare(address)`

Restrictions

only after the token's total supply was fixed

Emitted events

`ProfitShareUpdate(address, uint)`

Profit Share Withdrawal

There are three possible ways for [token holders](#) to get their owed profit share withdrawn from the token.

1. Withdraw by themselves and receive the profit share (Ether) on their Ethereum address

Function

```
withdrawProfitShare()
```

Restrictions

only after the token's total supply was fixed

Emitted events

```
ProfitShareWithdrawal(address, address, uint)
```

2. Withdraw by themselves but send the profit share (Ether) to another Ethereum address

Function

```
withdrawProfitShareTo(address)
```

Restrictions

only after the token's total supply was fixed

Emitted events

```
ProfitShareWithdrawal(address, address, uint)
```

3. Bulk Withdrawal by the [profit distributor](#) authority which sends the Ether to the [token holders'](#) Ethereum addresses

Function

```
withdrawProfitShares(address[])
```

Restrictions

- only by [profit distributor](#)
- only after the token's total supply was fixed

Emitted events

```
ProfitShareWithdrawal(address, address, uint)
```

5.3.6 TOKEN TRANSFERS

Token transfers are possible via the [ERC20](#) functions:

1. A [token holder](#) can send tokens to another [whitelisted investor](#)

Function

```
transfer(address, uint)
```

Restrictions

- only if the sender is a [whitelisted investor](#)
- only if the recipients is a [whitelisted investor](#)
- only if the total supply is fixed

Emitted events

```
Transfer(address, address, uint)
```

2. A [token holder](#) can approve a [third party trustee](#) to transfer up to some amount of tokens from the token holders account to any [whitelisted investor](#)

Function

```
approve(address, uint)
```

Restrictions

- only if the sender is a [whitelisted investor](#)
- only if the total supply is fixed

Emitted events

```
Approval(address, address, uint)
```

3. A [third party trustee](#) can transfer up to some allowed limit of an [token holder's](#) account to another [whitelisted investor](#)

Function

```
transferFrom(address, address, uint)
```

Restrictions

- only by a [trustee](#) approved by the [token holder](#)
- only if the token amount is below the allowed amount
- only if the [token holder](#) is a [whitelisted investor](#)
- only if the recipient is a [whitelisted investor](#)

Emitted events

```
Transfer(address, address, uint)
```

5.3.7 TOKEN RECOVERY

In case a [token holders](#) wants to change his/her Ethereum address, e.g. if he/she lost his/her private key or his account was compromised, the [token recoverer](#) authority can attach the account related data to the token holder's new address.

Function

```
recoverToken(address, address)
```

Restrictions

- only by [token recoverer](#)
- only if the new address isn't already assigned to some account data within this token instance

Emitted events

- `TokenRecovery(address, address)`
- `Transfer(address, address, uint)`

5.3.8 DESTRUCTION

If the token sale wasn't successful, that is the total amount of tokens sold in both (public and private) sales of the [crowdsale](#) hasn't reached the predefined goal, the token contract instance gets destructed, i.e. removed from the Ethereum ledger.

Any profits deposited in the token instance will be transferred to the [token owner](#) upon destruction.

Function

`destruct()`

Restrictions

only by [minter](#), i.e. the related [crowdsale](#) instance

Emitted events

none

5.4 LIFECYCLE

5.4.1 MINTING PHASE

As long as the token's [crowdsale](#) was not finalized tokens can be [minted](#), i.e. created out of thin air by the [minter](#) authority, which is supposed to be the [crowdsale instance](#) itself.

Start time

at deployment

End time

- upon call of `finishMinting()`
- upon token [destruction](#)

5.4.2 TRADING PHASE

After the [minting](#) was finished, the token's total supply is [fixed](#), thus [Token holder](#) can withdraw their [profit share](#) and are allowed to trade ([transfer](#)) their tokens with others.

Start time

upon call of `finishMinting()`

End time

never

6 PROJECT MANAGER

6.1 PURPOSE

Keep a central list of and create new *project*, where a project is a data structure containing:

1. the name of the project,
2. the associated [whitelist](#) contract instance,
3. the associated [token](#) contract instance,
4. the associated [crowdsale](#) contract instance.

Additionally, the project manager acts as the rate source for the managed crowdsale contracts.

6.2 ROLES

6.2.1 PROJECT MANAGER OWNER

Initially

the [Project Manager](#) contract deployer

Number

one at a time

Assignment

- by deploying the contract
- by gaining ownership from previous owner

Permissions

- [transfer ownership](#) to another account
- set [current whitelist](#)
- set [token factory](#)
- set [crowdsale factory](#)
- set [rate admin](#)

6.2.2 WHITELIST

Current [whitelist](#) which will be initially used in newly created projects.

Initially

none

Number

zero or one

Assignmentonly by [project manager owner](#)**Permissions**

update current Ether rate

6.2.3 TOKEN FACTORY

A helper contract to deploy new [token contracts](#).

Initially

none

Number

zero or one

Assignmentonly by [project manager owner](#)**Permissions**

update current Ether rate

6.2.4 CROWDSALE FACTORY

A helper contract to deploy new [crowdsale contracts](#).

Initially

none

Number

zero or one

Assignmentonly by [project manager owner](#)**Permissions**

update current Ether rate

6.2.5 RATE ADMIN

Authority who adjusts the ether rate to the current Ether price in Euro cents.

Initially

none

Number

zero or one

Assignment

only by [project manager owner](#)

Permissions

update current Ether rate

6.3 FUNCTIONALITY

6.3.1 ROLE MANAGEMENT

The project manager instance's [owner](#) can set the addresses of the following roles:

1. [Current whitelist](#)

Function

`setWhitelist(address)`

Restrictions

only by [owner](#)

Emitted events

none

2. [Token factory](#)

Function

`setTokenFactory(address)`

Restrictions

only by [owner](#)

Emitted events

none

3. [Crowdsale factory](#)

Function

`setCrowdsaleFactory(address)`

Restrictions

only by [owner](#)

Emitted events

none

4. [Rate admin](#)**Function**

`setRateAdmin(address)`

Restrictions

only by [owner](#)

Emitted events

`RateAdminChange(address, address)`

6.3.2 PROJECT CREATION

Creating a project involves several steps:

1. deploy a [token](#) instance which is assigned to the [current whitelist](#),
2. deploy a [crowdsale](#) instance which is assigned to the token and the project manager as its [rate source](#)
3. set the crowdsale to be the [minter](#) of its token
4. save the project (the three contracts) along with the project name in the projects list

The project manager allows to easily set up a new project. It utilizes two helper factory contracts.

Function

`createNewProject(...)`

Restrictions

only by [project manager owner](#)

Emitted events

none

6.3.3 RATE SOURCE

As the [token price](#) denominated in EUR-cents should be constant, the Ether rate has to be adjusted regularly to the current Ether price (in EUR-cents).

[Tokens](#) which were deployed by the project manager will have this contract set as their rate source.

READING CURRENT RATE

Function

`etherRate()`

Restrictions

none

UPDATING CURRENT RATE

Function

`setEtherRate(uint)`

Restrictions

only by [rate admin](#)

Emitted events

`RateChange(uint, uint)`