SCS Documentation

Overview

This documentation is for the smart contract system pertaining to the Aventus token sale. This includes:

- An interface of the ERC20 token standard (*ERC20.sol*)
- The Aventus token (AVT) contract (*AVTToken.sol*) inheriting from this interface
- A safe math contract that ensures arithmetic integrity (*SafeMath.sol*)
- The contract governing the Aventus token sale (*TokenSale.sol*)
- The Gnosis/ConsenSys multisig wallet (*MultisigWallet.sol*)

Note: “Presale” and “Private Sale” have been used interchangeably throughout this document. This time period pertains to the 2 day window before the public crowd sale (July 17 until crowd sale date of July 19), where only the `prebuy()` function in the `TokenSale.sol` smart contract can be called by the private sale address (`privAddress`).

ERC20.sol

This contract implements the ERC20 token standard interface described here. It was built for the sole purpose of inheritance by the Aventus Token contract.

AVTToken.sol

The Aventus token contract, inheriting from the ERC20 token standard interface. This contract will be created by the minter of AVT tokens, which is the Aventus token sale contract.

Contract variables

- **`uint public constant ILLIQUID_PERIOD`:** For how long illiquid tokens are illiquid for (1 year) before they can be transferred again. This period starts *after* the `endMintingTime`.

- **`uint public endMintingTime`:** The time after which no more minting of AVT can occur (this will be set by the Aventus token sale contract and is the end time of the token sale).

- **`address public minter`:** The contract who mints new AVT (the Aventus token sale contract).

- **`uint public supply:`** Total AVT supply, in terms of its lowest denomination miniAVT (1 AVT = 10,000 miniAVT). This will be 10,000,000 [or 10^7] AVT or in its lowest denomination, 100,000,000,000 [or 10^{11}] miniAVT.

- **`mapping(address => uint) public balances:`** A mapping storing the balance in miniAVT of a given address.

- **`mapping(address => uint) public illiquidBalances:`** A mapping storing the illiquid balance in miniAVT (e.g. not transferrable until after the `ILLIQUID_PERIOD`) of a given address.

- **`mapping(address => mapping(address => uint)) public allowed:`** A mapping storing, for a given address, a mapping of addresses allowed to spend certain amounts of its balance (in `balances`) on its behalf.

Functions

- **`AVTToken(uint _endMintingTime):`** Constructor. This contract is created by the Aventus token sale contract. The message sender gets set as the `minter`, and only argument passed in is the `endMintingTime`.

- **`balanceOf(address _owner):`** Returns the balance of a given address, checking the `balances` mapping.

- **`totalSupply():`** Returns the current total supply.

- **`illiquidBalanceOf(address _owner):`** Returns the illiquid balance of a given address, checking the `illiquidBalances` mapping.

- **`createToken(address _recipient, uint _value):`** Called by the token sale contract to mint new AVT tokens. Checks that (a) current time is within the minting time interval; and (b) that only the minter is calling (e.g. the token sale contract). It then increments the balance of the recipient meant to receive the token (e.g. the contributor, in the `balanaces` mapping), and increments the total supply, both by the token value denominated in miniAVT. Note, these tokens are transferrable as soon as the minting period is over (e.g. when the crowd sale ends). Returns `true` if successful.

- **`createIlliquidToken(address _recipient, uint _value):`** Called also by the token sale contract to mint new illiquid AVT tokens (non-transferrable for the `ILLIQUID_PERIOD`). Functions exactly as the `createToken` function, but instead of incrementing the recipient's balance in the `balances` mapping, it increments its balance in the `illiquidBalances` mapping. Returns `true` if successful.

- **`makeLiquid():`** Makes the illiquid balance of the message sender liquid (e.g. decrements his balance in `illiquidBalances`, and increments it by that amount in balances). Ensures using the modifier isLiquid that the current time is greater than the endMintingTime combined with the illiquid period.

- **transfer(address _to, uint _value):** Implements the ERC20 transfer function. Allows the message sender to transfer _value AVT (denominated in miniAVT) to address _to. First checks that the message sender has sufficient funds (using the modifier hasFunds), and checks that the tokens are transferrable (e.g. the current time is after the minting period using the isTransferable modifier). Fires an ERC20 transfer event, indexed by the from and to addresses, and also storing the value transferred. Returns true if successful.

- **transferFrom(address _from, address _to, uint _value):** Implements the ERC20 transferFrom function, which lets the message sender transfer _value on behalf of address _from to address _to. Apart from checking that tokens are transferrable as in the transfer function (e.g. that the current time is greater than the minting time end), and checking that _from has sufficient funds _value to send, it also checks that the message sender has sufficient allowance to send _value to _to on _from's behalf (using the hasAllowance modifier). Also fires the ERC20 transfer event, indexed by the from and to addresses, and also storing the value transferred. Returns true if successful.

- **approve(address _spender, uint _value)**: Allow address _spender to spend _value (in miniAVT) from the message sender's balance (stored in balances) on his behalf, by changing the value of the spender in the mapping stored in the allowed mapping for the message sender. To avoid the issues described here (e.g. race attacks), we ensure that every approve call only happens if either the allowance of the given _spender is zero or that an allowance of zero is being passed in (e.g. one can never change their allowance from N > 0 to M > 0). In the function, an ERC20 Approval event is fired, indexed by the approver (the message sender), the spender, and also revealing the value approved. Returns true if successful.

- **allowance(address _owner, address _spender):** Returns the value a given _owner address has let a _spender address spend.

- **():** Default function - throws if ether is sent to the contract.

TokenSale.sol

This is the contract that governs the Aventus token sale. Its responsibility is to take investments in Ether from investors, and mint new AVT (60% of the ultimate total supply) accordingly. It also mints the illiquid AVT allocation of 15% to the team and advisors (illiquid for a year), and mints the remaining 25% liquid AVT that will be allocated to corporate partners, new user incentives, and bounties.

The contract inherits from `SafeMath.sol`, so that it can perform all arithmetic operations for determining the amount of AVT to be sold to a given address given its contribution in Ether with maximal integrity.

Contract variables

- AVT prices and allocations
    - **`uint public constant PRICE_PRIV`:** The private sale price in AVT (denominated in miniAVT) per Ether. This must be at a 25% discount to the `PRICE_BASE` below.
    - **`uint public constant PRICE_BASE`:** The public sale price of AVT (denominated in miniAVT) per Ether.
    - **`uint public constant MAX_SUPPLY`:** The maximum supply of AVT that can be minted. This will be 10,000,000 AVT or 100,000,000,000 AVT (denominated in miniAVT).
    - **`uint public constant ALLOC_CROWDSALE`:** The maximum amount of AVT (denominated in miniAVT) that will be sold in the crowd sale (6,000,000 AVT or 60% of the maximum total supply).
    - **`uint public constant ALLOC_LIQUID`:** The remaining liquid allocation of AVT (denominated in miniAVT) (new user incentive pot, social and bug bounties, and corporate partners), which will be 25% of the maximum total supply (e.g. 2,500,000 AVT).
    - **`uint public constant ALLOC_ILLIQUID`:** The illiquid allocation of AVT (denominated in miniAVT), locked for a year (advisors and team), which will be 15% of the maximum total supply (e.g. 1,500,000 AVT).
    - **`uint public constant PRIV_ALLOC_MAX`:** The maximum AVT (denominated in miniAVT) that can be minted and sold to private sale contributors, 3,000,000 AVT (or 50% of the crowd sale AVT allocation).
- Token sale times
    - **`uint public privateStartTime`:** The time at which the private sale starts.
    - **`uint public publicStartTime`:** The time at which the public sale starts (which is the same time at which the private sale ends).
    - **`uint public publicEndTime`:** The time at which the public sale ends.
- Important addresses

    - **`uint public privAddress`:** The multisig wallet which will be allocated the AVT raised in the private sale. It will also store all Ether raised in the private sale and deposit it only a few hours before the public sale starts.

    - **`uint public multisigAddress`:** The multisig wallet that will store all Ether raised.

    - **`uint public aventusAddress`:** The Aventus address that will be initially allocated both the remaining liquid (25%) and illiquid (15%) portions of the total AVT supply.

    - **`uint public avtToken`:** The AVT token address, that is created from and assigned through this contract by the `aventusAddress`.

- Crowdsale stats

    - **`uint public etherRaised`:** The total ether raised to date during the entire crowd sale (both private and public).

    - **`uint public avtSold`:** The total AVT (denominated in miniAVT) sold to date during the entire crowd sale (both private and public).

    - **`uint public privPortionRaised`:** The total AVT (denominated in miniAVT) sold to date during the private sale.

Events

- **`TokenAddress(AVTToken indexed _token)`**: Fires when the AVTToken is created so that we know its address.
- **`PreBuy(uint _amount)`**: Fires when a private sale contribution occurs, notifying us of the amount in AVT (denominated in miniAVT) that was sold.
- **`Buy(address indexed _recipient, uint _amount)`**: Fires when a public sale contribution occurs. Notifies us of the recipient of AVT along with the amount in AVT (denominated in miniAVT) that was sold.

Functions

- **`TokenSale(address _priv, address _aventus, address _multisig, uint _publicStartTime, uint _privateStartTime)`**: The constructor. Sets all of the contract variables to what was input, and sets the `publicEndTime` to 5 days after the `publicStartTime`.

- **`setupAVT()`**: Creates and stores the AVTToken contract and mints the illiquid and liquid token allocations to the Aventus address. A check is present to ensure that only the `aventusAddress` can call this function, so no attacks occur between the creation of this contract and the setting up of the token contract. To save execution costs, we do not put a check for ensuring that the AVTToken address is zero before calling this function, since only our address can call into this function (and we know to only call this once). `TokenAddress` event is also fired so we can store the AVT address.

- **`processPurchase(uint _rate, uint _remaining)`**: This is a private function called both by `prebuy()` and the default function for public contribution. First, using the `_rate` and the `msg.value`, it calculates the amount of AVT (denominated in miniAVT) to mint to the message sender using the `safeDiv` function in the SafeMath contract. First, the function ensures that the amount of AVT to be minted is less than the remaining available amount of AVT that can be minted. Next, the raised Ether is sent to the `multisigWallet` address. Finally, AVT is minted to the message sender, and the statistics are updated. The amount of AVT sold is returned once all of the previous calls are successful.

- **`preBuy()`**: This is a `payable` function for accepting solely private sale contributions. First, a check is performed to see that the current time is within the private sale time limits (using the `isPreCrowdsale` modifier). Next, there are checks for the message sender being the private address (`isPrivAddress` modifier), and for ensuring that the AVTToken address has been set (using the `isAVTSetup` modifier). In the function, the amount is calculated using the `processPurchase` function, with the private sale rate (`PRICE_PRIV`) and the `_remaining` argument as the difference between the maximum private allocation (`PRIV_ALLOC_MAX`) and the amount that has been raised till now in the private sale (`privPortionRaised`). A `PreBuy` event is fired, and the `privPortionRaised` is incremented by the amount of AVT sold.

- **`()`**: This is the default, `payable` function for accepting solely public sale contributions. It ensures that the current time is within the public crowd sale time period (`isCrowdSale` modifier), and that the AVTToken address has been set (using the `isAVTSetup` modifier). In the function, the amount is calculated using the `processPurchase` function, with the public sale rate (`PRICE_BASE) and the _remaining` argument as the difference between the the maximum crowdsale amount (`ALLOC_CROWDSALE`) and the amount that has been raised till now in total (`avtSold`). Finally, a `Buy` event is fired.

SafeMath.sol

This is a contract used for safe arithmetic logic. It is used by the TokenSale contract (`TokenSale.sol`) for calculating the amount of AVT (denominated in miniAVT) that should be sold to a given address given their Ether contribution.

Functions

- **`safeMul(uint a, uint b)`**: Takes in two integers, and outputs the result (as an integer) of the safe multiplication operation. It ensures no arithmetic overflow. The conditions are that either integer a is zero, or if not, that the product `c` of integers `a` and `b` divided by a is indeed `b`, to prevent overflow.

- **`safeDiv(uint a, uint b)`**: This ensures sound division of `a` by `b`. First, we ensure that `b` is not zero so that there is no error when dividing by zero. We then ensure that there is no remainder unaccounted for.

MultisigWallet.sol

This is the ConsenSys multisig wallet. Detailed documentation for this contract has not been written considering you guys should have it already. This wallet will be used for two contract instances:

(a) the private sale address (`privAddress` member in `TokenSale.sol`) that will be sending funds to the token sale contract (`TokenSale.sol`). The private sale multisig wallet will have 3 keys, held by: Aventus Systems Limited, Global Advisors Jersey Limited, and Brave New Coin. The way this interaction will work, is that presale contributions in Ether will go to the private sale multisig address. Then, during the 2 days of private sale (see the *Overview* section above), only the `privAddress` multisig wallet will be able to send presale contributions using the `preBuy` function in the token sale smart contract. This multisig wallet will be configured to have the 3 keys, with a 2 / 3 majority needed to confirm and execute any transaction.

(b) the multisig wallet address (`multisigAddress` member in `TokenSale.sol`) that will receive raised funds from the token sale contract. This multisig wallet will have 5 keys, held by: Alan Vey (Aventus founder #1), Annika Monari (Aventus founder #2), Global Advisors Jersey Limited, Techemy Limited (Brave New Coin), and Professor William Knottenbelt. It will require a 3 out of 5 majority to send transactions from this wallet.