Yoritex Token Crowdsale Smart Contract

January, 2018

YORITEX TOKEN CROWDSALE SMART CONTRACT

```
pragma solidity ^0.4.18;


/**
 * @title SafeMath
 * @dev Math operations with safety checks that throw on error
 */
library SafeMath {
  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;
  }

  function div(uint256 a, uint256 b) internal pure returns (uint256) {
    // assert(b > 0); // Solidity automatically throws when dividing by 0
    uint256 c = a / b;
    // assert(a == b * c + a % b); // There is no case in which this doesn't hold
    return c;
  }

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

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

/**
 * @title ERC20Basic
 * @dev Simpler version of ERC20 interface
 * @dev see https://github.com/ethereum/EIPs/issues/179
 */
contract ERC20Basic {
  uint256 public totalSupply;
  function balanceOf(address who) public view returns (uint256);
  function transfer(address to, uint256 value) public returns (bool);
  event Transfer(address indexed from, address indexed to, uint256 value);
}

/**
 * @title ERC20 interface
 * @dev see https://github.com/ethereum/EIPs/issues/20
 */
contract ERC20 is ERC20Basic {
  function allowance(address owner, address spender) public view returns (uint256);
  function transferFrom(address from, address to, uint256 value) public returns (bool);
  function approve(address spender, uint256 value) public returns (bool);
  event Approval(address indexed owner, address indexed spender, uint256 value);
}
pragma solidity ^0.4.18;


/**
 * @title Basic token
 * @dev Basic version of StandardToken, with no allowances.
 */
contract BasicToken is ERC20Basic {
  using SafeMath for uint256;

  mapping(address => uint256) balances;

  /**
  * @dev transfer token for a specified address
  * @param _to The address to transfer to.
  * @param _value The amount to be transferred.
  */
  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;
  }
```

```
  /**
  * @dev Gets the balance of the specified address.
  * @param _owner The address to query the the balance of.
  * @return An uint256 representing the amount owned by the passed address.
  */
  function balanceOf(address _owner) public view returns (uint256 balance) {
    return balances[_owner];
  }

}

/**
 * @title Standard ERC20 token
 *
 * @dev Implementation of the basic standard token.
 * @dev https://github.com/ethereum/EIPs/issues/20
 * @dev Based on code by FirstBlood: https://github.com/Firstbloodio/token/blob/master/smart_contract/FirstBloodToken.sol
 */
contract StandardToken is ERC20, BasicToken {

  mapping (address => mapping (address => uint256)) internal allowed;


  /**
   * @dev Transfer tokens from one address to another
   * @param _from address The address which you want to send tokens from
   * @param _to address The address which you want to transfer to
   * @param _value uint256 the amount of tokens to be transferred
   */
  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;
  }

  /**
   * @dev Approve the passed address to spend the specified amount of tokens on behalf of msg.sender.
   *
   * Beware that changing an allowance with this method brings the risk that someone may use both the old
   * and the new allowance by unfortunate transaction ordering. One possible solution to mitigate this
   * race condition is to first reduce the spender's allowance to 0 and set the desired value afterwards:
   * https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
   * @param _spender The address which will spend the funds.
   * @param _value The amount of tokens to be spent.
   */
  function approve(address _spender, uint256 _value) public returns (bool) {
    allowed[msg.sender][_spender] = _value;
    Approval(msg.sender, _spender, _value);
    return true;
  }

  /**
   * @dev Function to check the amount of tokens that an owner allowed to a spender.
   * @param _owner address The address which owns the funds.
   * @param _spender address The address which will spend the funds.
   * @return A uint256 specifying the amount of tokens still available for the spender.
   */
  function allowance(address _owner, address _spender) public view returns (uint256) {
    return allowed[_owner][_spender];
  }

  /**
   * approve should be called when allowed[_spender] == 0. To increment
   * allowed value is better to use this function to avoid 2 calls (and wait until
   * the first transaction is mined)
   * From MonolithDAO Token.sol
   */
  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;
  }

  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;
  }

}


/**
 * @title Ownable
 * @dev The Ownable contract has an owner address, and provides basic authorization control
 * functions, this simplifies the implementation of "user permissions".
 */
contract Ownable {
  address public owner;


  event OwnershipTransferred(address indexed previousOwner, address indexed newOwner);


  /**
   * @dev The Ownable constructor sets the original `owner` of the contract to the sender
   * account.
   */
  function Ownable() public {
    owner = msg.sender;
  }


  /**
   * @dev Throws if called by any account other than the owner.
   */
  modifier onlyOwner() {
    require(msg.sender == owner);
    _;
  }


  /**
   * @dev Allows the current owner to transfer control of the contract to a newOwner.
   * @param newOwner The address to transfer ownership to.
   */
  function transferOwnership(address newOwner) public onlyOwner {
    require(newOwner != address(0));
    OwnershipTransferred(owner, newOwner);
    owner = newOwner;
  }

}


/**
 * @title Mintable token
 * @dev Simple ERC20 Token example, with mintable token creation
 * @dev Issue: * https://github.com/OpenZeppelin/zeppelin-solidity/issues/120
 * Based on code by TokenMarketNet: https://github.com/TokenMarketNet/ico/blob/master/contracts/MintableToken.sol
 */

contract MintableToken is StandardToken, Ownable {
  event Mint(address indexed to, uint256 amount);
  event MintFinished();

  bool public mintingFinished = false;


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

  /**
   * @dev Function to mint tokens
   * @param _to The address that will receive the minted tokens.
   * @param _amount The amount of tokens to mint.
   * @return A boolean that indicates if the operation was successful.
   */
  function mint(address _to, uint256 _amount) onlyOwner canMint public returns (bool) {
    totalSupply = totalSupply.add(_amount);
    balances[_to] = balances[_to].add(_amount);
    Mint(_to, _amount);
    Transfer(address(0), _to, _amount);
    return true;
  }

  /**
   * @dev Function to stop minting new tokens.
   * @return True if the operation was successful.
```

```
   */
  function finishMinting() onlyOwner canMint public returns (bool) {
    mintingFinished = true;
    MintFinished();
    return true;
  }
}

/**
 * @title YRXToken
 * @dev ERC20 based Token, where all tokens are pre-assigned to the creator.
 * Note they can later distribute these tokens as they wish using `transfer` and other
 * `StandardToken` functions.
 */
contract YRXToken is MintableToken {
    string public constant name = "Yoritex Token";
    string public constant symbol = "YRX";
    uint8 public constant decimals = 18;
    address public crowdsaleAddress;

    uint256 public constant INITIAL_SUPPLY = 510000000 * 1 ether;

    modifier nonZeroAddress(address _to) {               // Ensures an address is provided
        require(_to != 0x0);
        _;
    }

    modifier nonZeroAmount(uint _amount) {               // Ensures a non-zero amount
        require(_amount > 0);
        _;
    }

    modifier nonZeroValue() {                            // Ensures a non-zero value is passed
        require(msg.value > 0);
        _;
    }

    modifier onlyCrowdsale() {                           // Ensures only crowdfund can call the function
        require(msg.sender == crowdsaleAddress);
        _;
    }

    /**
     * @dev Constructor that gives msg.sender all of existing tokens.
     */
    function YRXToken() public {
        totalSupply = INITIAL_SUPPLY;
        balances[msg.sender] = totalSupply;
    }

    // -----------------------------------------------------
    // Sets the crowdsale address, can only be done once
    // -----------------------------------------------------
    function setCrowdsaleAddress(address _crowdsaleAddress) external onlyOwner nonZeroAddress(_crowdsaleAddress) returns (bool
success){
        require(crowdsaleAddress == 0x0);
        crowdsaleAddress = _crowdsaleAddress;
        decrementBalance(owner, totalSupply);
        addToBalance(crowdsaleAddress, totalSupply);
        Transfer(0x0, _crowdsaleAddress, totalSupply);
        return true;
    }

    // -----------------------------------------------------
    // Function for the Crowdsale to transfer tokens
    // -----------------------------------------------------
    function transferFromCrowdsale(address _to, uint256 _amount) external onlyCrowdsale nonZeroAmount(_amount) nonZeroAddress(_to)
returns (bool success) {
        require(balanceOf(crowdsaleAddress) >= _amount);
        decrementBalance(crowdsaleAddress, _amount);
        addToBalance(_to, _amount);
        Transfer(0x0, _to, _amount);
        return true;
    }

    // -----------------------------------------------------
    // Adds to balance
    // -----------------------------------------------------
    function addToBalance(address _address, uint _amount) internal {
        balances[_address] = balances[_address].add(_amount);
    }

    // -----------------------------------------------------
    // Removes from balance
    // -----------------------------------------------------
    function decrementBalance(address _address, uint _amount) internal {
```

```
        balances[_address] = balances[_address].sub(_amount);
    }

}

/**
 * @title YRXCrowdsale
 */

contract YRXCrowdsale is Ownable {
  using SafeMath for uint256;

  // true for finalised presale
  bool public isPreSaleFinalised;
  // true for finalised crowdsale
  bool public isFinalised;
  // The token being sold
  YRXToken public YRX;
  // address where funds are collected
  address public wallet;

  // amount of raised money in wei
  uint256 public weiRaised;

  // current amount of purchased tokens for pre-sale and main sale
  uint256 public preSaleTotalSupply;
  uint256 public mainSaleTotalSupply;
  // current amount of minted tokens for bounty
  uint256 public bountyTotalSupply;
  uint256 private mainSaleTokensExtra;

  event WalletAddressChanged(address _wallet);          // Triggered upon owner changing the wallet address
  event AmountRaised(address beneficiary, uint amountRaised); // Triggered upon crowdfund being finalized
  event Mint(address indexed to, uint256 amount);
  /**
   * event for token purchase logging
   * @param purchaser who paid for the tokens
   * @param beneficiary who got the tokens
   * @param value weis paid for purchase
   * @param amount amount of tokens purchased
   */
  event TokenPurchase(address indexed purchaser, address indexed beneficiary, uint256 value, uint256 amount);

  modifier nonZeroAddress(address _to) {               // Ensures an address is provided
    require(_to != 0x0);
    _;
  }

  modifier nonZeroAmount(uint _amount) {               // Ensures a non-zero amount
    require(_amount > 0);
    _;
  }

  modifier nonZeroValue() {                            // Ensures a non-zero value is passed
    require(msg.value > 0);
    _;
  }

  modifier crowdsaleIsActive() {                       // Ensures the crowdfund is ongoing
    require(!isFinalised && (isInPreSale() || isInMainSale()));
    _;
  }

  function YRXCrowdsale(address _wallet, address _token) public {

    /* default start-end periods dates
      _preSaleStartTime =     '29 Nov 2017 00:00:00 GMT' 1511913600
      _preSaleEndTime =       '10 Jan 2018 23:59:59 GMT' 1515628799
      _mainSaleStartTime =    '11 Jan 2018 00:00:00 GMT' 1515628800
      _mainSaleEndTime =      '11 May 2018 00:00:00 GMT' 1525996800
    */

    // check dates
    require(mainSaleStartTime() >= now);               // can't start main sale in the past
    require(preSaleEndTime() < mainSaleStartTime());      // can't start main sale before the end of pre-sale
    require(preSaleStartTime() < preSaleEndTime());       // the end of pre-sale can't happen before it's start
    require(mainSaleStartTime() < mainSaleEndTime());     // the end of main sale can't happen before it's start

    // create token contract
    YRX = YRXToken(_token);
    wallet = _wallet;
    isPreSaleFinalised = false;
    isFinalised = false;

    // current amount of purchased tokens for pre-sale and main sale
    preSaleTotalSupply = 0;
```

```
  mainSaleTotalSupply = 0;
  bountyTotalSupply = 0;
  mainSaleTokensExtra = 0;
}

// -----------------------------------------------
// Changes main contribution wallet
// -----------------------------------------------
function changeWalletAddress(address _wallet) external onlyOwner {
  wallet = _wallet;
  WalletAddressChanged(_wallet);
}


function maxTokens() public pure returns(uint256) {
  return 510000000 * 1 ether;
}

function preSaleMaxTokens() public pure returns(uint256) {
  return 51000000 * 1 ether;
}

function mainSaleMaxTokens() public view returns(uint256) {
  return 433500000  * 1 ether + mainSaleTokensExtra;
}

function bountyMaxTokens() public pure returns(uint256) {
  return 25500000 * 1 ether;
}

function preSaleStartTime() public pure returns(uint256) {
  return 1511913600;
}

function preSaleEndTime() public pure returns(uint256) {
  return 1515628799;
}

function mainSaleStartTime() public pure returns(uint256) {
  return 1515628800;
}

function mainSaleEndTime() public pure returns(uint256) {
  return 1525996800;
}

function rate() public pure returns(uint256) {
  return 540;
}

function discountRate() public pure returns(uint256) {
  return 1350;
}

function discountICO() public pure returns(uint256) {
  return 60;
}

function isInPreSale() public constant returns(bool){
  return now >= preSaleStartTime() && now <= preSaleEndTime();
}

function isInMainSale() public constant returns(bool){
  return now >= mainSaleStartTime() && now <= mainSaleEndTime();
}

function totalSupply() public view returns(uint256){
  return YRX.totalSupply();
}

// fallback function can be used to buy tokens
function () public payable {
  buyTokens(msg.sender);
}

// low level token purchase function
function buyTokens(address beneficiary) public crowdsaleIsActive nonZeroAddress(beneficiary) nonZeroValue payable {
  uint256 weiAmount = msg.value;

  // calculate token amount to be created
  uint256 tokens = weiAmount * rate();
  if (isInPreSale()) {
    require(!isPreSaleFinalised);
    tokens = weiAmount * discountRate();
    require(tokens <= preSaleTokenLeft());
  }
```

```
    if (isInMainSale()) {
      // tokens with discount
      tokens = weiAmount * discountRate();
      require(mainSaleTotalSupply + tokens <= mainSaleMaxTokens());
    }

    // update state
    weiRaised = weiRaised.add(weiAmount);
    if (isInPreSale())
      preSaleTotalSupply += tokens;
    if (isInMainSale())
      mainSaleTotalSupply += tokens;

    forwardFunds();
    if (!YRX.transferFromCrowdsale(beneficiary, tokens)) {revert();}
    TokenPurchase(msg.sender, beneficiary, weiAmount, tokens);
  }

  // send ether to the fund collection wallet
  function forwardFunds() internal {
    wallet.transfer(msg.value);
  }

  // mint tokens to beneficiary
  function mint(address beneficiary, uint256 amount) public onlyOwner crowdsaleIsActive nonZeroAddress(beneficiary)
nonZeroAmount(amount) {
    // check period
    bool withinPreSalePeriod = isInPreSale();
    bool withinMainSalePeriod = isInMainSale();
    if (withinPreSalePeriod) {
      require(!isPreSaleFinalised);
      require(amount <= preSaleTokenLeft());
    }
    if (withinMainSalePeriod) {
      require(amount <= (mainSaleMaxTokens() - mainSaleTotalSupply));
    }

    if (withinPreSalePeriod)
      preSaleTotalSupply += amount;
    if (withinMainSalePeriod)
      mainSaleTotalSupply += amount;

    if (!YRX.transferFromCrowdsale(beneficiary, amount)) {revert();}
    Mint(beneficiary, amount);
  }

  function preSaleTokenLeft() public constant returns(uint256){
    return preSaleMaxTokens() - preSaleTotalSupply;
  }

  // finalise presale
  function finalisePreSale() public onlyOwner {
    require(!isFinalised);
    require(!isPreSaleFinalised);
    require(now >= preSaleStartTime()); // can finalise presale only after it starts

    if (preSaleTokenLeft() > 0) {
      mainSaleTokensExtra = preSaleTokenLeft();
    }

    isPreSaleFinalised = true;
  }

  // finalase crowdsale (mainsale)
  function finalise() public onlyOwner returns(bool success){
    require(!isFinalised);
    require(now >= mainSaleStartTime()); // can finalise mainsale (crowdsale) only after it starts
    AmountRaised(wallet, weiRaised);
    isFinalised = true;
    return true;
  }

  // mint bounty tokens to beneficiary
  function mintBounty(address beneficiary, uint256 amount) public onlyOwner crowdsaleIsActive nonZeroAddress(beneficiary)
nonZeroAmount(amount) {
    require(amount <= (bountyMaxTokens() - bountyTotalSupply));

    bountyTotalSupply += amount;
    if (!YRX.transferFromCrowdsale(beneficiary, amount)) {revert();}
    Mint(beneficiary, amount);
  }

}
```