TTC smart contract code

```
/**
 *Submitted for verification at BscScan.com on 2021-02-24
*/

/**
 *Submitted for verification at Etherscan.io on 2019-02-06
*/

pragma solidity ^0.4.24;

// File: node_modules/openzeppelin-solidity/contracts/token/ERC20/IERC20.sol

/**
 * @title ERC20 interface
 * @dev see https://github.com/ethereum/EIPs/issues/20
 */
interface IERC20 {
  function totalSupply() external view returns (uint256);

  function balanceOf(address who) external view returns (uint256);

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

  function transfer(address to, uint256 value) external returns (bool);
  function approve(address spender, uint256 value)
    external returns (bool);

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

  event Transfer(
    address indexed from,
    address indexed to,
    uint256 value
  );

  event Approval(
    address indexed owner,
    address indexed spender,
    uint256 value
  );
}

// File: node_modules/openzeppelin-solidity/contracts/math/SafeMath.sol

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

  /**
  * @dev Multiplies two numbers, reverts on overflow.
  */
  function mul(uint256 a, uint256 b) internal pure returns (uint256) {
    // Gas optimization: this is cheaper than requiring 'a' not being zero, but the
    // benefit is lost if 'b' is also tested.
    // See: https://github.com/OpenZeppelin/openzeppelin-solidity/pull/522
    if (a == 0) {
      return 0;
    }

    uint256 c = a * b;
    require(c / a == b);

    return c;
  }

  /**
  * @dev Integer division of two numbers truncating the quotient, reverts on division by zero.
  */
  function div(uint256 a, uint256 b) internal pure returns (uint256) {
    require(b > 0); // Solidity only automatically asserts 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;
  }

  /**
  * @dev Subtracts two numbers, reverts on overflow (i.e. if subtrahend is greater than minuend).
  */
  function sub(uint256 a, uint256 b) internal pure returns (uint256) {
    require(b <= a);
    uint256 c = a - b;
```

```
        return c;
    }

    /**
    * @dev Adds two numbers, reverts on overflow.
    */
    function add(uint256 a, uint256 b) internal pure returns (uint256) {
        uint256 c = a + b;
        require(c >= a);

        return c;
    }

    /**
    * @dev Divides two numbers and returns the remainder (unsigned integer modulo),
    * reverts when dividing by zero.
    */
    function mod(uint256 a, uint256 b) internal pure returns (uint256) {
        require(b != 0);
        return a % b;
    }
}

// File: node_modules/openzeppelin-solidity/contracts/token/ERC20/ERC20.sol

/**
 * @title Standard ERC20 token
 *
 * @dev Implementation of the basic standard token.
 * https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md
 * Originally based on code by FirstBlood:
https://github.com/Firstbloodio/token/blob/master/smart_contract/FirstBloodToken.sol
```

```
 */
contract ERC20 is IERC20 {
  using SafeMath for uint256;

  mapping (address => uint256) private _balances;

  mapping (address => mapping (address => uint256)) private _allowed;

  uint256 private _totalSupply;

  /**
  * @dev Total number of tokens in existence
  */
  function totalSupply() public view returns (uint256) {
    return _totalSupply;
  }

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

  /**
   * @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];
   }

   /**
   * @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) {
     _transfer(msg.sender, 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) {
  require(spender != address(0));

  _allowed[msg.sender][spender] = value;
  emit Approval(msg.sender, spender, value);
  return true;
}

/**
 * @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(value <= _allowed[from][msg.sender]);

  _allowed[from][msg.sender] = _allowed[from][msg.sender].sub(value);
  _transfer(from, to, value);
  return true;
}

/**
```

```
 * @dev Increase the amount of tokens that an owner allowed to a 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
 * @param spender The address which will spend the funds.
 * @param addedValue The amount of tokens to increase the allowance by.
 */
function increaseAllowance(
  address spender,
  uint256 addedValue
)
  public
  returns (bool)
{
  require(spender != address(0));

  _allowed[msg.sender][spender] = (
    _allowed[msg.sender][spender].add(addedValue));
  emit Approval(msg.sender, spender, _allowed[msg.sender][spender]);
  return true;
}

/**
 * @dev Decrease the amount of tokens that an owner allowed to a spender.
 * approve should be called when allowed_[_spender] == 0. To decrement
 * allowed value is better to use this function to avoid 2 calls (and wait until
 * the first transaction is mined)
 * From MonolithDAO Token.sol
 * @param spender The address which will spend the funds.
 * @param subtractedValue The amount of tokens to decrease the allowance by.
```

```
  */
function decreaseAllowance(
    address spender,
    uint256 subtractedValue
)
    public
    returns (bool)
{
    require(spender != address(0));

    _allowed[msg.sender][spender] = (
        _allowed[msg.sender][spender].sub(subtractedValue));
    emit Approval(msg.sender, spender, _allowed[msg.sender][spender]);
    return true;
}

/**
* @dev Transfer token for a specified addresses
* @param from The address to transfer from.
* @param to The address to transfer to.
* @param value The amount to be transferred.
*/
function _transfer(address from, address to, uint256 value) internal {
    require(value <= _balances[from]);
    require(to != address(0));

    _balances[from] = _balances[from].sub(value);
    _balances[to] = _balances[to].add(value);
    emit Transfer(from, to, value);
}
```

```
/**
 * @dev Internal function that mints an amount of the token and assigns it to
 * an account. This encapsulates the modification of balances such that the
 * proper events are emitted.
 * @param account The account that will receive the created tokens.
 * @param value The amount that will be created.
 */
function _mint(address account, uint256 value) internal {
  require(account != 0);
  _totalSupply = _totalSupply.add(value);
  _balances[account] = _balances[account].add(value);
  emit Transfer(address(0), account, value);
}

/**
 * @dev Internal function that burns an amount of the token of a given
 * account.
 * @param account The account whose tokens will be burnt.
 * @param value The amount that will be burnt.
 */
function _burn(address account, uint256 value) internal {
  require(account != 0);
  require(value <= _balances[account]);

  _totalSupply = _totalSupply.sub(value);
  _balances[account] = _balances[account].sub(value);
  emit Transfer(account, address(0), value);
}

/**
 * @dev Internal function that burns an amount of the token of a given
```

```
   * account, deducting from the sender's allowance for said account. Uses the
   * internal burn function.
   * @param account The account whose tokens will be burnt.
   * @param value The amount that will be burnt.
   */
  function _burnFrom(address account, uint256 value) internal {
    require(value <= _allowed[account][msg.sender]);

    // Should https://github.com/OpenZeppelin/zeppelin-solidity/issues/707 be accepted,
    // this function needs to emit an event with the updated approval.
    _allowed[account][msg.sender] = _allowed[account][msg.sender].sub(
      value);
    _burn(account, value);
  }
}

// File: contracts\ERC20\TokenMintERC20Token.sol

/**
 * @title TokenMintERC20Token
 * @author TokenMint.io
 *
 * @dev Standard ERC20 token with optional functions implemented.
 * For full specification of ERC-20 standard see:
 * https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md
 */
contract TokenMintERC20Token is ERC20 {

    string private _name;
    string private _symbol;
    uint8 private _decimals;
    constructor(string name, string symbol, uint8 decimals, uint256 totalSupply, address feeReceiver,
address tokenOwnerAddress) public payable {
      _name = name;
      _symbol = symbol;
      _decimals = decimals;

      // set tokenOwnerAddress as owner of all tokens
      _mint(tokenOwnerAddress, totalSupply);

      // pay the service fee for contract deployment
      feeReceiver.transfer(msg.value);
    }

    // optional functions from ERC20 stardard

    /**
     * @return the name of the token.
     */
    function name() public view returns (string) {
      return _name;
    }

    /**
     * @return the symbol of the token.
     */
    function symbol() public view returns (string) {
      return _symbol;
    }

    /**
     * @return the number of decimals of the token.
     */
    function decimals() public view returns (uint8) {
      return _decimals;
    }
}
```