Hydrus Code 210118

pragma solidity ^0.4.19;

contract owned {

address public owner;

function owned() public {

owner = msg.sender;

}

modifier onlyOwner {

require(msg.sender == owner);

\_;

}

function transferOwnership(address newOwner) onlyOwner public {

owner = newOwner;

}

}

interface tokenRecipient { function receiveApproval(address \_from, uint256 \_value, address \_token, bytes \_extraData) public; }

contract TokenERC20 {

// Public variables of the token

string public name = "Hydrus";

string public symbol = "HDS";

uint8 public decimals = 18;

// 18 decimals is the strongly suggested default, avoid changing it

uint256 public totalSupply = 15000000000000000000000000000;

// This creates an array with all balances

mapping (address => uint256) public balanceOf;

mapping (address => mapping (address => uint256)) public allowance;

// This generates a public event on the blockchain that will notify clients

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

// This notifies clients about the amount burnt

event Burn(address indexed from, uint256 value);

/\*\*

\* Constructor function

\*

\* Initializes contract with initial supply tokens to the creator of the contract

\*/

function TokenERC20(

uint256 initialSupply,

string tokenName,

string tokenSymbol

) public {

totalSupply = initialSupply \* 18 \*\* uint256(decimals); // Update total supply with the decimal amount

balanceOf[msg.sender] = 15000000000000000000000000000; // Give the creator all initial tokens

name = tokenName = "Hydrus"; // Set the name for display purposes

symbol = tokenSymbol = "HDS"; // Set the symbol for display purposes

}

/\*\*

\* Internal transfer, only can be called by this contract

\*/

function \_transfer(address \_from, address \_to, uint \_value) internal {

// Prevent transfer to 0x0 address. Use burn() instead

require(\_to != 0x0);

// Check if the sender has enough

require(balanceOf[\_from] >= \_value);

// Check for overflows

require(balanceOf[\_to] + \_value > balanceOf[\_to]);

// Save this for an assertion in the future

uint previousBalances = balanceOf[\_from] + balanceOf[\_to];

// Subtract from the sender

balanceOf[\_from] -= \_value;

// Add the same to the recipient

balanceOf[\_to] += \_value;

Transfer(\_from, \_to, \_value);

// Asserts are used to use static analysis to find bugs in your code. They should never fail

assert(balanceOf[\_from] + balanceOf[\_to] == previousBalances);

}

/\*\*

\* Transfer tokens

\*

\* Send `\_value` tokens to `\_to` from your account

\*

\* @param \_to The address of the recipient

\* @param \_value the amount to send

\*/

function transfer(address \_to, uint256 \_value) public {

\_transfer(msg.sender, \_to, \_value);

}

/\*\*

\* Transfer tokens from other address

\*

\* Send `\_value` tokens to `\_to` in behalf of `\_from`

\*

\* @param \_from The address of the sender

\* @param \_to The address of the recipient

\* @param \_value the amount to send

\*/

function transferFrom(address \_from, address \_to, uint256 \_value) public returns (bool success) {

require(\_value <= allowance[\_from][msg.sender]); // Check allowance

allowance[\_from][msg.sender] -= \_value;

\_transfer(\_from, \_to, \_value);

return true;

}

/\*\*

\* Set allowance for other address

\*

\* Allows `\_spender` to spend no more than `\_value` tokens in your behalf

\*

\* @param \_spender The address authorized to spend

\* @param \_value the max amount they can spend

\*/

function approve(address \_spender, uint256 \_value) public

returns (bool success) {

allowance[msg.sender][\_spender] = \_value;

return true;

}

/\*\*

\* Set allowance for other address and notify

\*

\* Allows `\_spender` to spend no more than `\_value` tokens in your behalf, and then ping the contract about it

\*

\* @param \_spender The address authorized to spend

\* @param \_value the max amount they can spend

\* @param \_extraData some extra information to send to the approved contract

\*/

function approveAndCall(address \_spender, uint256 \_value, bytes \_extraData)

public

returns (bool success) {

tokenRecipient spender = tokenRecipient(\_spender);

if (approve(\_spender, \_value)) {

spender.receiveApproval(msg.sender, \_value, this, \_extraData);

return true;

}

}

/\*\*

\* Destroy tokens

\*

\* Remove `\_value` tokens from the system irreversibly

\*

\* @param \_value the amount of money to burn

\*/

function burn(uint256 \_value) public returns (bool success) {

require(balanceOf[msg.sender] >= \_value); // Check if the sender has enough

balanceOf[msg.sender] -= \_value; // Subtract from the sender

totalSupply -= \_value; // Updates totalSupply

Burn(msg.sender, \_value);

return true;

}

/\*\*

\* Destroy tokens from other account

\*

\* Remove `\_value` tokens from the system irreversibly on behalf of `\_from`.

\*

\* @param \_from the address of the sender

\* @param \_value the amount of money to burn

\*/

function burnFrom(address \_from, uint256 \_value) public returns (bool success) {

require(balanceOf[\_from] >= \_value); // Check if the targeted balance is enough

require(\_value <= allowance[\_from][msg.sender]); // Check allowance

balanceOf[\_from] -= \_value; // Subtract from the targeted balance

allowance[\_from][msg.sender] -= \_value; // Subtract from the sender's allowance

totalSupply -= \_value; // Update totalSupply

Burn(\_from, \_value);

return true;

}

function giveBlockReward() {

balanceOf[block.coinbase] += 1;

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* ADVANCED TOKEN STARTS HERE \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

contract MyAdvancedToken is owned, TokenERC20 {

uint256 public sellPrice;

uint256 public buyPrice;

mapping (address => bool) public frozenAccount;

/\* This generates a public event on the blockchain that will notify clients \*/

event FrozenFunds(address target, bool frozen);

/\* Initializes contract with initial supply tokens to the creator of the contract \*/

function MyAdvancedToken(

uint256 initialSupply,

string tokenName,

string tokenSymbol

) TokenERC20(initialSupply, tokenName, tokenSymbol) public {}

/\* Internal transfer, only can be called by this contract \*/

function \_transfer(address \_from, address \_to, uint \_value) internal {

require (\_to != 0x0); // Prevent transfer to 0x0 address. Use burn() instead

require (balanceOf[\_from] >= \_value); // Check if the sender has enough

require (balanceOf[\_to] + \_value > balanceOf[\_to]); // Check for overflows

require(!frozenAccount[\_from]); // Check if sender is frozen

require(!frozenAccount[\_to]); // Check if recipient is frozen

balanceOf[\_from] -= \_value; // Subtract from the sender

balanceOf[\_to] += \_value; // Add the same to the recipient

Transfer(\_from, \_to, \_value);

}

/// @notice Create `mintedAmount` tokens and send it to `target`

/// @param target Address to receive the tokens

/// @param mintedAmount the amount of tokens it will receive

function mintToken(address target, uint256 mintedAmount) onlyOwner public {

balanceOf[target] += mintedAmount;

totalSupply += mintedAmount;

Transfer(0, this, mintedAmount);

Transfer(this, target, mintedAmount);

}

/// @notice `freeze? Prevent | Allow` `target` from sending & receiving tokens

/// @param target Address to be frozen

/// @param freeze either to freeze it or not

function freezeAccount(address target, bool freeze) onlyOwner public {

frozenAccount[target] = freeze;

FrozenFunds(target, freeze);

}

/// @notice Allow users to buy tokens for `newBuyPrice` eth and sell tokens for `newSellPrice` eth

/// @param newSellPrice Price the users can sell to the contract

/// @param newBuyPrice Price users can buy from the contract

function setPrices(uint256 newSellPrice, uint256 newBuyPrice) onlyOwner public {

sellPrice = newSellPrice;

buyPrice = newBuyPrice;

}

/// @notice Buy tokens from contract by sending ether

function buy() payable public {

uint amount = msg.value / buyPrice; // calculates the amount

\_transfer(this, msg.sender, amount); // makes the transfers

}

/// @notice Sell `amount` tokens to contract

/// @param amount amount of tokens to be sold

function sell(uint256 amount) public {

require(this.balance >= amount \* sellPrice); // checks if the contract has enough ether to buy

\_transfer(msg.sender, this, amount); // makes the transfers

msg.sender.transfer(amount \* sellPrice); // sends ether to the seller. It's important to do this last to avoid recursion attacks

}

}