/\*\*

\*Submitted for verification at Etherscan.io on 2018-10-27

\*/

pragma solidity ^0.4.25;

/\*\*

\* XGN is token issued by Golden Currency.

\* XGN can be exchanged for Golden cash, each Golden is backed by 0.025 grams of gold.

\* Terms of such exchange are available at https://goldencurrency.money

\*/

library SafeMath {

function mul(uint256 a, uint256 b) internal pure returns (uint256) {

uint256 c = a \* b;

assert(a == 0 || 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;

}

function max64(uint64 a, uint64 b) internal pure returns (uint64) {

return a >= b ? a : b;

}

function min64(uint64 a, uint64 b) internal pure returns (uint64) {

return a < b ? a : b;

}

function max256(uint256 a, uint256 b) internal pure returns (uint256) {

return a >= b ? a : b;

}

function min256(uint256 a, uint256 b) internal pure returns (uint256) {

return a < b ? a : b;

}

}

contract ERC20Basic {

uint256 public totalSupply;

bool public transfersEnabled;

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);

}

contract ERC20 {

uint256 public totalSupply;

bool public transfersEnabled;

function balanceOf(address \_owner) public constant returns (uint256 balance);

function transfer(address \_to, uint256 \_value) public returns (bool success);

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

function approve(address \_spender, uint256 \_value) public returns (bool success);

function allowance(address \_owner, address \_spender) public constant returns (uint256 remaining);

event Transfer(address indexed \_from, address indexed \_to, uint256 \_value);

event Approval(address indexed \_owner, address indexed \_spender, uint256 \_value);

}

contract BasicToken is ERC20Basic {

using SafeMath for uint256;

mapping (address => uint256) balances;

/\*\*

\* Protection against short address attack

\*/

modifier onlyPayloadSize(uint numwords) {

assert(msg.data.length == numwords \* 32 + 4);

\_;

}

/\*\*

\* @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 onlyPayloadSize(2) returns (bool) {

require(\_to != address(0));

require(\_value <= balances[msg.sender]);

require(transfersEnabled);

// SafeMath.sub will throw if there is not enough balance.

balances[msg.sender] = balances[msg.sender].sub(\_value);

balances[\_to] = balances[\_to].add(\_value);

emit 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 constant returns (uint256 balance) {

return balances[\_owner];

}

}

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 onlyPayloadSize(3) returns (bool) {

require(\_to != address(0));

require(\_value <= balances[\_from]);

require(\_value <= allowed[\_from][msg.sender]);

require(transfersEnabled);

balances[\_from] = balances[\_from].sub(\_value);

balances[\_to] = balances[\_to].add(\_value);

allowed[\_from][msg.sender] = allowed[\_from][msg.sender].sub(\_value);

emit 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;

emit 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 onlyPayloadSize(2) constant returns (uint256 remaining) {

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 success) {

allowed[msg.sender][\_spender] = allowed[msg.sender][\_spender].add(\_addedValue);

emit Approval(msg.sender, \_spender, allowed[msg.sender][\_spender]);

return true;

}

function decreaseApproval(address \_spender, uint \_subtractedValue) public returns (bool success) {

uint oldValue = allowed[msg.sender][\_spender];

if (\_subtractedValue > oldValue) {

allowed[msg.sender][\_spender] = 0;

}

else {

allowed[msg.sender][\_spender] = oldValue.sub(\_subtractedValue);

}

emit 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;

address public administratorOne;

address public administratorTwo;

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

event AdministratorChanged(uint8 numberAdmin, address indexed previousAddress, address indexed newAddress);

/\*\*

\* @dev Throws if called by any account other than the owner.

\*/

modifier onlyOwner() {

require(msg.sender == owner);

\_;

}

/\*\*

\* @dev Throws if called by any account other than the owner or administrators.

\*/

modifier onlyOwnerOrAnyAdmin() {

require(msg.sender == owner || msg.sender == administratorOne || msg.sender == administratorTwo);

\_;

}

/\*\*

\* @dev Allows the current owner to transfer control of the contract to a newOwner.

\* @param \_newOwner The address to transfer ownership to.

\*/

function changeOwner(address \_newOwner) onlyOwner public {

require(\_newOwner != address(0));

emit OwnerChanged(owner, \_newOwner);

owner = \_newOwner;

}

/\*\*

\* @dev Changing the address of the contract administrator wallet.

\* @param \_numberAdmin Admin number

\* @param \_newAddress New admin address

\*/

function changeAdmin(uint8 \_numberAdmin, address \_newAddress) onlyOwner public {

require(\_newAddress != address(0));

address oldAddress;

if (\_numberAdmin == 1) {

oldAddress = administratorOne;

administratorOne = \_newAddress;

}

if (\_numberAdmin == 2) {

oldAddress = administratorTwo;

administratorTwo = \_newAddress;

}

emit AdministratorChanged(\_numberAdmin, oldAddress, \_newAddress);

}

}

/\*\*

\* @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 {

string public constant name = "Golden";

string public constant symbol = "XGN";

uint8 public constant decimals = 18;

bool public mintingFinished;

event Mint(address indexed to, uint256 amount);

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, address \_owner) canMint internal returns (bool) {

require(\_amount <= balances[\_owner]);

balances[\_to] = balances[\_to].add(\_amount);

balances[\_owner] = balances[\_owner].sub(\_amount);

emit Mint(\_to, \_amount);

emit Transfer(\_owner, \_to, \_amount);

return true;

}

/\*\*

\* Peterson's Law Protection

\* Claim tokens

\*/

function claimTokens(address \_token) public onlyOwner {

if (\_token == 0x0) {

owner.transfer(address(this).balance);

return;

}

MintableToken token = MintableToken(\_token);

uint256 balance = token.balanceOf(this);

token.transfer(owner, balance);

emit Transfer(\_token, owner, balance);

}

}

/\*\*

\* @title TwoAdmins

\* @dev TwoAdmins is a base contract for managing a token.

\* Start and end time for sales are set by contract administrators.

\* Investors can make tokens when authorized by administrators.

\* Funds collected are forwarded to a administratorOne as they arrive.

\*/

contract TwoAdmins is Ownable, MintableToken {

using SafeMath for uint256;

/\*\*

\* Price: 1 ETH = 204 token

\* https://www.coingecko.com/en/coins/ethereum

\* October, 23, 2018

\*/

uint256 public priceToken = 204;

bool saleOfTokens = false;

// amount of raised money in wei

uint256 public weiRaised;

uint256 public constant INITIAL\_SUPPLY = 10\*\*12 \* (10 \*\* uint256(decimals));

event TokenPurchase(address indexed beneficiary, uint256 value, uint256 amount);

event TokenLimitReached(address indexed sender, uint256 purchasedToken);

event ChangePriceToken(address indexed owner, uint256 newValue, uint256 oldValue);

event Burn(address indexed burner, uint256 value);

event DisableTransfer(address indexed admin);

event EnableTransfer(address indexed admin);

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

constructor(address \_owner) public

{

require(\_owner != address(0));

owner = \_owner;

//owner = msg.sender; // for test's

transfersEnabled = true;

mintingFinished = false;

totalSupply = INITIAL\_SUPPLY;

balances[owner] = balances[owner].add(INITIAL\_SUPPLY);

}

// fallback function can be used to buy tokens

function() payable public {

buyTokens(msg.sender);

}

function buyTokens(address \_investor) public payable returns (uint256){

require(\_investor != address(0));

uint256 weiAmount = msg.value;

if (saleOfTokens == false) {

weiRaised = weiRaised.add(weiAmount);

administratorOne.transfer(weiAmount);

return 0;

}

uint256 tokens = validPurchaseTokens(weiAmount);

if (tokens == 0) {revert();}

weiRaised = weiRaised.add(weiAmount);

mint(\_investor, tokens, owner);

emit TokenPurchase(\_investor, weiAmount, tokens);

administratorOne.transfer(weiAmount);

return tokens;

}

function getTotalAmountOfTokens(uint256 \_weiAmount) internal view returns (uint256) {

uint256 amountOfTokens = 0;

amountOfTokens = \_weiAmount.mul(priceToken);

return amountOfTokens;

}

function sendTokens(address \_walletTo, uint256 \_tokens) public onlyOwnerOrAnyAdmin returns (bool \_result) {

\_result = false;

require(\_walletTo != address(0));

mint(\_walletTo, \_tokens, owner);

emit SendToken(msg.sender, \_walletTo, \_tokens);

\_result = true;

}

function validPurchaseTokens(uint256 \_weiAmount) public returns (uint256) {

uint256 addTokens = getTotalAmountOfTokens(\_weiAmount);

if (addTokens > balances[owner]) {

emit TokenLimitReached(msg.sender, addTokens);

return 0;

}

return addTokens;

}

/\*\*

\* @dev owner burn Token.

\* @param \_value amount of burnt tokens

\*/

function burnToken(uint \_value) public onlyOwnerOrAnyAdmin {

require(\_value > 0);

require(\_value <= balances[owner]);

require(\_value <= totalSupply);

balances[owner] = balances[owner].sub(\_value);

totalSupply = totalSupply.sub(\_value);

emit Burn(msg.sender, \_value);

}

/\*\*

\* @dev owner or Admins disable transfer tokens

\*/

function disableTransfer() public onlyOwnerOrAnyAdmin {

transfersEnabled = false;

emit DisableTransfer(msg.sender);

}

/\*\*

\* @dev owner or Admins enable transfer tokens

\*/

function enableTransfer() public onlyOwnerOrAnyAdmin {

transfersEnabled = true;

emit EnableTransfer(msg.sender);

}

/\*\*

\* @dev owner or any Admin change price of tokens

\* @param \_newPriceToken new price

\*/

function setPriceToken(uint256 \_newPriceToken) external onlyOwnerOrAnyAdmin {

require(\_newPriceToken > 0);

uint256 \_oldPriceToken = priceToken;

priceToken = \_newPriceToken;

emit ChangePriceToken(msg.sender, \_newPriceToken, \_oldPriceToken);

}

function startSale() public onlyOwnerOrAnyAdmin {

saleOfTokens = true;

}

function stopSale() public onlyOwnerOrAnyAdmin {

saleOfTokens = false;

}

}