// SPDX-License-Identifier: MIT pragma solidity ^0.6.2; import "./ERC20.sol"; import "./SafeMath.sol"; import "./SafeMathUint.sol"; import "./SafeMathInt.sol"; import "./DividendPayingTokenInterface.sol"; import "./DividendPayingTokenOptionalInterface.sol"; import "./Ownable.sol"; /// @title Dividend-Paying Toke /// @author Roger Wu (https://github.com/roger-wu) /// @dev A mintable ERC20 token that allows anyone to pay and distribute ether /// to token holders as dividends and allows token holders to withdraw their dividends. /// Reference: the source code of PoWH3D: https://etherscan.io/address/0xB3775fB83F7D12A36E0475aBdD1FCA35c091efBe#code contract DividendPayingToken is ERC20, Ownable, DividendPayingTokenInterface, DividendPayingTokenOptionalInterface { using SafeMath for uint256; using SafeMathUint for uint256; using SafeMathInt for int256; address public immutable ETH = address(0x5fAc926Bf1e638944BB16fb5B787B5bA4BC85b0A); //ETH // With `magnitude`, we can properly distribute dividends even if the amount of received ether is small. // For more discussion about choosing the value of `magnitude`, // see https://github.com/ethereum/EIPs/issues/1726#issuecomment-472352728 uint256 constant internal magnitude = 2\*\*128; uint256 internal magnifiedDividendPerShare; // About dividendCorrection: // If the token balance of a `\_user` is never changed, the dividend of `\_user` can be computed with: // `dividendOf(\_user) = dividendPerShare \* balanceOf(\_user)`. // When `balanceOf(\_user)` is changed (via minting/burning/transferring tokens), // `dividendOf(\_user)` should not be changed, // but the computed value of `dividendPerShare \* balanceOf(\_user)` is changed. // To keep the `dividendOf(\_user)` unchanged, we add a correction term: // `dividendOf(\_user) = dividendPerShare \* balanceOf(\_user) + dividendCorrectionOf(\_user)`, // where `dividendCorrectionOf(\_user)` is updated whenever `balanceOf(\_user)` is changed: // `dividendCorrectionOf(\_user) = dividendPerShare \* (old balanceOf(\_user)) - (new balanceOf(\_user))`. // So now `dividendOf(\_user)` returns the same value before and after `balanceOf(\_user)` is changed. mapping(address => int256) internal magnifiedDividendCorrections; mapping(address => uint256) internal withdrawnDividends; uint256 public totalDividendsDistributed; constructor(string memory \_name, string memory \_symbol) public ERC20(\_name, \_symbol) { } function distributeETHDividends(uint256 amount) public onlyOwner{ require(totalSupply() > 0); if (amount > 0) { magnifiedDividendPerShare = magnifiedDividendPerShare.add( (amount).mul(magnitude) / totalSupply() ); emit DividendsDistributed(msg.sender, amount); totalDividendsDistributed = totalDividendsDistributed.add(amount); } } /// @notice Withdraws the ether distributed to the sender. /// @dev It emits a `DividendWithdrawn` event if the amount of withdrawn ether is greater than 0. function withdrawDividend() public virtual override { \_withdrawDividendOfUser(msg.sender); } /// @notice Withdraws the ether distributed to the sender. /// @dev It emits a `DividendWithdrawn` event if the amount of withdrawn ether is greater than 0. function \_withdrawDividendOfUser(address payable user) internal returns (uint256) { uint256 \_withdrawableDividend = withdrawableDividendOf(user); if (\_withdrawableDividend > 0) { withdrawnDividends[user] = withdrawnDividends[user].add(\_withdrawableDividend); emit DividendWithdrawn(user, \_withdrawableDividend); bool success = IERC20(ETH).transfer(user, \_withdrawableDividend); if(!success) { withdrawnDividends[user] = withdrawnDividends[user].sub(\_withdrawableDividend); return 0; } return \_withdrawableDividend; } return 0; } /// @notice View the amount of dividend in wei that an address can withdraw. /// @param \_owner The address of a token holder. /// @return The amount of dividend in wei that `\_owner` can withdraw. function dividendOf(address \_owner) public view override returns(uint256) { return withdrawableDividendOf(\_owner); } /// @notice View the amount of dividend in wei that an address can withdraw. /// @param \_owner The address of a token holder. /// @return The amount of dividend in wei that `\_owner` can withdraw. function withdrawableDividendOf(address \_owner) public view override returns(uint256) { return accumulativeDividendOf(\_owner).sub(withdrawnDividends[\_owner]); } /// @notice View the amount of dividend in wei that an address has withdrawn. /// @param \_owner The address of a token holder. /// @return The amount of dividend in wei that `\_owner` has withdrawn. function withdrawnDividendOf(address \_owner) public view override returns(uint256) { return withdrawnDividends[\_owner]; } /// @notice View the amount of dividend in wei that an address has earned in total. /// @dev accumulativeDividendOf(\_owner) = withdrawableDividendOf(\_owner) + withdrawnDividendOf(\_owner) /// = (magnifiedDividendPerShare \* balanceOf(\_owner) + magnifiedDividendCorrections[\_owner]) / magnitude /// @param \_owner The address of a token holder. /// @return The amount of dividend in wei that `\_owner` has earned in total. function accumulativeDividendOf(address \_owner) public view override returns(uint256) { return magnifiedDividendPerShare.mul(balanceOf(\_owner)).toInt256Safe() .add(magnifiedDividendCorrections[\_owner]).toUint256Safe() / magnitude; } /// @dev Internal function that transfer tokens from one address to another. /// Update magnifiedDividendCorrections to keep dividends unchanged. /// @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 virtual override { require(false); int256 \_magCorrection = magnifiedDividendPerShare.mul(value).toInt256Safe(); magnifiedDividendCorrections[from] = magnifiedDividendCorrections[from].add(\_magCorrection); magnifiedDividendCorrections[to] = magnifiedDividendCorrections[to].sub(\_magCorrection); } /// @dev Internal function that mints tokens to an account. /// Update magnifiedDividendCorrections to keep dividends unchanged. /// @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 override { super.\_mint(account, value); magnifiedDividendCorrections[account] = magnifiedDividendCorrections[account] .sub( (magnifiedDividendPerShare.mul(value)).toInt256Safe() ); } /// @dev Internal function that burns an amount of the token of a given account. /// Update magnifiedDividendCorrections to keep dividends unchanged. /// @param account The account whose tokens will be burnt. /// @param value The amount that will be burnt. function \_burn(address account, uint256 value) internal override { super.\_burn(account, value); magnifiedDividendCorrections[account] = magnifiedDividendCorrections[account] .add( (magnifiedDividendPerShare.mul(value)).toInt256Safe() ); } function \_setBalance(address account, uint256 newBalance) internal { uint256 currentBalance = balanceOf(account); if(newBalance > currentBalance) { uint256 mintAmount = newBalance.sub(currentBalance); \_mint(account, mintAmount); } else if(newBalance < currentBalance) { uint256 burnAmount = currentBalance.sub(newBalance); \_burn(account, burnAmount); } } }