/\*\*

\*Submitted for verification at BscScan.com on 2022-01-24

\*/

/\*\*

██████╗██╗ ███████╗ █████╗ ██████╗ ██╗ ██╗ █████╗ ████████╗███████╗██████╗

██╔════╝██║ ██╔════╝██╔══██╗██╔══██╗ ██║ ██║██╔══██╗╚══██╔══╝██╔════╝██╔══██╗

██║ ██║ █████╗ ███████║██████╔╝ ██║ █╗ ██║███████║ ██║ █████╗ ██████╔╝

██║ ██║ ██╔══╝ ██╔══██║██╔══██╗ ██║███╗██║██╔══██║ ██║ ██╔══╝ ██╔══██╗

╚██████╗███████╗███████╗██║ ██║██║ ██║ ╚███╔███╔╝██║ ██║ ██║ ███████╗██║ ██║

╚═════╝╚══════╝╚══════╝╚═╝ ╚═╝╚═╝ ╚═╝ ╚══╝╚══╝ ╚═╝ ╚═╝ ╚═╝ ╚══════╝╚═╝ ╚═╝

\*/

// SPDX-License-Identifier: MIT

pragma solidity 0.8.10;

interface IUniswapV2Router01 {

function factory() external pure returns (address);

function WETH() external pure returns (address);

function addLiquidity(

address tokenA,

address tokenB,

uint amountADesired,

uint amountBDesired,

uint amountAMin,

uint amountBMin,

address to,

uint deadline

) external returns (uint amountA, uint amountB, uint liquidity);

function addLiquidityETH(

address token,

uint amountTokenDesired,

uint amountTokenMin,

uint amountETHMin,

address to,

uint deadline

) external payable returns (uint amountToken, uint amountETH, uint liquidity);

function removeLiquidity(

address tokenA,

address tokenB,

uint liquidity,

uint amountAMin,

uint amountBMin,

address to,

uint deadline

) external returns (uint amountA, uint amountB);

function removeLiquidityETH(

address token,

uint liquidity,

uint amountTokenMin,

uint amountETHMin,

address to,

uint deadline

) external returns (uint amountToken, uint amountETH);

function removeLiquidityWithPermit(

address tokenA,

address tokenB,

uint liquidity,

uint amountAMin,

uint amountBMin,

address to,

uint deadline,

bool approveMax, uint8 v, bytes32 r, bytes32 s

) external returns (uint amountA, uint amountB);

function removeLiquidityETHWithPermit(

address token,

uint liquidity,

uint amountTokenMin,

uint amountETHMin,

address to,

uint deadline,

bool approveMax, uint8 v, bytes32 r, bytes32 s

) external returns (uint amountToken, uint amountETH);

function swapExactTokensForTokens(

uint amountIn,

uint amountOutMin,

address[] calldata path,

address to,

uint deadline

) external returns (uint[] memory amounts);

function swapTokensForExactTokens(

uint amountOut,

uint amountInMax,

address[] calldata path,

address to,

uint deadline

) external returns (uint[] memory amounts);

function swapExactETHForTokens(uint amountOutMin, address[] calldata path, address to, uint deadline)

external

payable

returns (uint[] memory amounts);

function swapTokensForExactETH(uint amountOut, uint amountInMax, address[] calldata path, address to, uint deadline)

external

returns (uint[] memory amounts);

function swapExactTokensForETH(uint amountIn, uint amountOutMin, address[] calldata path, address to, uint deadline)

external

returns (uint[] memory amounts);

function swapETHForExactTokens(uint amountOut, address[] calldata path, address to, uint deadline)

external

payable

returns (uint[] memory amounts);

function quote(uint amountA, uint reserveA, uint reserveB) external pure returns (uint amountB);

function getAmountOut(uint amountIn, uint reserveIn, uint reserveOut) external pure returns (uint amountOut);

function getAmountIn(uint amountOut, uint reserveIn, uint reserveOut) external pure returns (uint amountIn);

function getAmountsOut(uint amountIn, address[] calldata path) external view returns (uint[] memory amounts);

function getAmountsIn(uint amountOut, address[] calldata path) external view returns (uint[] memory amounts);

}

interface IUniswapV2Router02 is IUniswapV2Router01 {

function removeLiquidityETHSupportingFeeOnTransferTokens(

address token,

uint liquidity,

uint amountTokenMin,

uint amountETHMin,

address to,

uint deadline

) external returns (uint amountETH);

function removeLiquidityETHWithPermitSupportingFeeOnTransferTokens(

address token,

uint liquidity,

uint amountTokenMin,

uint amountETHMin,

address to,

uint deadline,

bool approveMax, uint8 v, bytes32 r, bytes32 s

) external returns (uint amountETH);

function swapExactTokensForTokensSupportingFeeOnTransferTokens(

uint amountIn,

uint amountOutMin,

address[] calldata path,

address to,

uint deadline

) external;

function swapExactETHForTokensSupportingFeeOnTransferTokens(

uint amountOutMin,

address[] calldata path,

address to,

uint deadline

) external payable;

function swapExactTokensForETHSupportingFeeOnTransferTokens(

uint amountIn,

uint amountOutMin,

address[] calldata path,

address to,

uint deadline

) external;

}

interface IUniswapV2Factory {

event PairCreated(address indexed token0, address indexed token1, address pair, uint);

function feeTo() external view returns (address);

function feeToSetter() external view returns (address);

function getPair(address tokenA, address tokenB) external view returns (address pair);

function allPairs(uint) external view returns (address pair);

function allPairsLength() external view returns (uint);

function createPair(address tokenA, address tokenB) external returns (address pair);

function setFeeTo(address) external;

function setFeeToSetter(address) external;

}

/\*\*

\* @dev Wrappers over Solidity's arithmetic operations.

\*

\* NOTE: `SignedSafeMath` is no longer needed starting with Solidity 0.8. The compiler

\* now has built in overflow checking.

\*/

library SignedSafeMath {

/\*\*

\* @dev Returns the multiplication of two signed integers, reverting on

\* overflow.

\*

\* Counterpart to Solidity's `\*` operator.

\*

\* Requirements:

\*

\* - Multiplication cannot overflow.

\*/

function mul(int256 a, int256 b) internal pure returns (int256) {

return a \* b;

}

/\*\*

\* @dev Returns the integer division of two signed integers. Reverts on

\* division by zero. The result is rounded towards zero.

\*

\* Counterpart to Solidity's `/` operator.

\*

\* Requirements:

\*

\* - The divisor cannot be zero.

\*/

function div(int256 a, int256 b) internal pure returns (int256) {

return a / b;

}

/\*\*

\* @dev Returns the subtraction of two signed integers, reverting on

\* overflow.

\*

\* Counterpart to Solidity's `-` operator.

\*

\* Requirements:

\*

\* - Subtraction cannot overflow.

\*/

function sub(int256 a, int256 b) internal pure returns (int256) {

return a - b;

}

/\*\*

\* @dev Returns the addition of two signed integers, reverting on

\* overflow.

\*

\* Counterpart to Solidity's `+` operator.

\*

\* Requirements:

\*

\* - Addition cannot overflow.

\*/

function add(int256 a, int256 b) internal pure returns (int256) {

return a + b;

}

}

// CAUTION

// This version of SafeMath should only be used with Solidity 0.8 or later,

// because it relies on the compiler's built in overflow checks.

/\*\*

\* @dev Wrappers over Solidity's arithmetic operations.

\*

\* NOTE: `SafeMath` is no longer needed starting with Solidity 0.8. The compiler

\* now has built in overflow checking.

\*/

library SafeMath {

/\*\*

\* @dev Returns the addition of two unsigned integers, with an overflow flag.

\*

\* \_Available since v3.4.\_

\*/

function tryAdd(uint256 a, uint256 b) internal pure returns (bool, uint256) {

unchecked {

uint256 c = a + b;

if (c < a) return (false, 0);

return (true, c);

}

}

/\*\*

\* @dev Returns the substraction of two unsigned integers, with an overflow flag.

\*

\* \_Available since v3.4.\_

\*/

function trySub(uint256 a, uint256 b) internal pure returns (bool, uint256) {

unchecked {

if (b > a) return (false, 0);

return (true, a - b);

}

}

/\*\*

\* @dev Returns the multiplication of two unsigned integers, with an overflow flag.

\*

\* \_Available since v3.4.\_

\*/

function tryMul(uint256 a, uint256 b) internal pure returns (bool, uint256) {

unchecked {

// 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-contracts/pull/522

if (a == 0) return (true, 0);

uint256 c = a \* b;

if (c / a != b) return (false, 0);

return (true, c);

}

}

/\*\*

\* @dev Returns the division of two unsigned integers, with a division by zero flag.

\*

\* \_Available since v3.4.\_

\*/

function tryDiv(uint256 a, uint256 b) internal pure returns (bool, uint256) {

unchecked {

if (b == 0) return (false, 0);

return (true, a / b);

}

}

/\*\*

\* @dev Returns the remainder of dividing two unsigned integers, with a division by zero flag.

\*

\* \_Available since v3.4.\_

\*/

function tryMod(uint256 a, uint256 b) internal pure returns (bool, uint256) {

unchecked {

if (b == 0) return (false, 0);

return (true, a % b);

}

}

/\*\*

\* @dev Returns the addition of two unsigned integers, reverting on

\* overflow.

\*

\* Counterpart to Solidity's `+` operator.

\*

\* Requirements:

\*

\* - Addition cannot overflow.

\*/

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

return a + b;

}

/\*\*

\* @dev Returns the subtraction of two unsigned integers, reverting on

\* overflow (when the result is negative).

\*

\* Counterpart to Solidity's `-` operator.

\*

\* Requirements:

\*

\* - Subtraction cannot overflow.

\*/

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

return a - b;

}

/\*\*

\* @dev Returns the multiplication of two unsigned integers, reverting on

\* overflow.

\*

\* Counterpart to Solidity's `\*` operator.

\*

\* Requirements:

\*

\* - Multiplication cannot overflow.

\*/

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

return a \* b;

}

/\*\*

\* @dev Returns the integer division of two unsigned integers, reverting on

\* division by zero. The result is rounded towards zero.

\*

\* Counterpart to Solidity's `/` operator.

\*

\* Requirements:

\*

\* - The divisor cannot be zero.

\*/

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

return a / b;

}

/\*\*

\* @dev Returns the remainder of dividing two unsigned integers. (unsigned integer modulo),

\* reverting when dividing by zero.

\*

\* Counterpart to Solidity's `%` operator. This function uses a `revert`

\* opcode (which leaves remaining gas untouched) while Solidity uses an

\* invalid opcode to revert (consuming all remaining gas).

\*

\* Requirements:

\*

\* - The divisor cannot be zero.

\*/

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

return a % b;

}

/\*\*

\* @dev Returns the subtraction of two unsigned integers, reverting with custom message on

\* overflow (when the result is negative).

\*

\* CAUTION: This function is deprecated because it requires allocating memory for the error

\* message unnecessarily. For custom revert reasons use {trySub}.

\*

\* Counterpart to Solidity's `-` operator.

\*

\* Requirements:

\*

\* - Subtraction cannot overflow.

\*/

function sub(

uint256 a,

uint256 b,

string memory errorMessage

) internal pure returns (uint256) {

unchecked {

require(b <= a, errorMessage);

return a - b;

}

}

/\*\*

\* @dev Returns the integer division of two unsigned integers, reverting with custom message on

\* division by zero. The result is rounded towards zero.

\*

\* Counterpart to Solidity's `/` operator. Note: this function uses a

\* `revert` opcode (which leaves remaining gas untouched) while Solidity

\* uses an invalid opcode to revert (consuming all remaining gas).

\*

\* Requirements:

\*

\* - The divisor cannot be zero.

\*/

function div(

uint256 a,

uint256 b,

string memory errorMessage

) internal pure returns (uint256) {

unchecked {

require(b > 0, errorMessage);

return a / b;

}

}

/\*\*

\* @dev Returns the remainder of dividing two unsigned integers. (unsigned integer modulo),

\* reverting with custom message when dividing by zero.

\*

\* CAUTION: This function is deprecated because it requires allocating memory for the error

\* message unnecessarily. For custom revert reasons use {tryMod}.

\*

\* Counterpart to Solidity's `%` operator. This function uses a `revert`

\* opcode (which leaves remaining gas untouched) while Solidity uses an

\* invalid opcode to revert (consuming all remaining gas).

\*

\* Requirements:

\*

\* - The divisor cannot be zero.

\*/

function mod(

uint256 a,

uint256 b,

string memory errorMessage

) internal pure returns (uint256) {

unchecked {

require(b > 0, errorMessage);

return a % b;

}

}

}

/\*\*

\* @dev Wrappers over Solidity's uintXX/intXX casting operators with added overflow

\* checks.

\*

\* Downcasting from uint256/int256 in Solidity does not revert on overflow. This can

\* easily result in undesired exploitation or bugs, since developers usually

\* assume that overflows raise errors. `SafeCast` restores this intuition by

\* reverting the transaction when such an operation overflows.

\*

\* Using this library instead of the unchecked operations eliminates an entire

\* class of bugs, so it's recommended to use it always.

\*

\* Can be combined with {SafeMath} and {SignedSafeMath} to extend it to smaller types, by performing

\* all math on `uint256` and `int256` and then downcasting.

\*/

library SafeCast {

/\*\*

\* @dev Returns the downcasted uint224 from uint256, reverting on

\* overflow (when the input is greater than largest uint224).

\*

\* Counterpart to Solidity's `uint224` operator.

\*

\* Requirements:

\*

\* - input must fit into 224 bits

\*/

function toUint224(uint256 value) internal pure returns (uint224) {

require(value <= type(uint224).max, "SafeCast: value doesn't fit in 224 bits");

return uint224(value);

}

/\*\*

\* @dev Returns the downcasted uint128 from uint256, reverting on

\* overflow (when the input is greater than largest uint128).

\*

\* Counterpart to Solidity's `uint128` operator.

\*

\* Requirements:

\*

\* - input must fit into 128 bits

\*/

function toUint128(uint256 value) internal pure returns (uint128) {

require(value <= type(uint128).max, "SafeCast: value doesn't fit in 128 bits");

return uint128(value);

}

/\*\*

\* @dev Returns the downcasted uint96 from uint256, reverting on

\* overflow (when the input is greater than largest uint96).

\*

\* Counterpart to Solidity's `uint96` operator.

\*

\* Requirements:

\*

\* - input must fit into 96 bits

\*/

function toUint96(uint256 value) internal pure returns (uint96) {

require(value <= type(uint96).max, "SafeCast: value doesn't fit in 96 bits");

return uint96(value);

}

/\*\*

\* @dev Returns the downcasted uint64 from uint256, reverting on

\* overflow (when the input is greater than largest uint64).

\*

\* Counterpart to Solidity's `uint64` operator.

\*

\* Requirements:

\*

\* - input must fit into 64 bits

\*/

function toUint64(uint256 value) internal pure returns (uint64) {

require(value <= type(uint64).max, "SafeCast: value doesn't fit in 64 bits");

return uint64(value);

}

/\*\*

\* @dev Returns the downcasted uint32 from uint256, reverting on

\* overflow (when the input is greater than largest uint32).

\*

\* Counterpart to Solidity's `uint32` operator.

\*

\* Requirements:

\*

\* - input must fit into 32 bits

\*/

function toUint32(uint256 value) internal pure returns (uint32) {

require(value <= type(uint32).max, "SafeCast: value doesn't fit in 32 bits");

return uint32(value);

}

/\*\*

\* @dev Returns the downcasted uint16 from uint256, reverting on

\* overflow (when the input is greater than largest uint16).

\*

\* Counterpart to Solidity's `uint16` operator.

\*

\* Requirements:

\*

\* - input must fit into 16 bits

\*/

function toUint16(uint256 value) internal pure returns (uint16) {

require(value <= type(uint16).max, "SafeCast: value doesn't fit in 16 bits");

return uint16(value);

}

/\*\*

\* @dev Returns the downcasted uint8 from uint256, reverting on

\* overflow (when the input is greater than largest uint8).

\*

\* Counterpart to Solidity's `uint8` operator.

\*

\* Requirements:

\*

\* - input must fit into 8 bits.

\*/

function toUint8(uint256 value) internal pure returns (uint8) {

require(value <= type(uint8).max, "SafeCast: value doesn't fit in 8 bits");

return uint8(value);

}

/\*\*

\* @dev Converts a signed int256 into an unsigned uint256.

\*

\* Requirements:

\*

\* - input must be greater than or equal to 0.

\*/

function toUint256(int256 value) internal pure returns (uint256) {

require(value >= 0, "SafeCast: value must be positive");

return uint256(value);

}

/\*\*

\* @dev Returns the downcasted int128 from int256, reverting on

\* overflow (when the input is less than smallest int128 or

\* greater than largest int128).

\*

\* Counterpart to Solidity's `int128` operator.

\*

\* Requirements:

\*

\* - input must fit into 128 bits

\*

\* \_Available since v3.1.\_

\*/

function toInt128(int256 value) internal pure returns (int128) {

require(value >= type(int128).min && value <= type(int128).max, "SafeCast: value doesn't fit in 128 bits");

return int128(value);

}

/\*\*

\* @dev Returns the downcasted int64 from int256, reverting on

\* overflow (when the input is less than smallest int64 or

\* greater than largest int64).

\*

\* Counterpart to Solidity's `int64` operator.

\*

\* Requirements:

\*

\* - input must fit into 64 bits

\*

\* \_Available since v3.1.\_

\*/

function toInt64(int256 value) internal pure returns (int64) {

require(value >= type(int64).min && value <= type(int64).max, "SafeCast: value doesn't fit in 64 bits");

return int64(value);

}

/\*\*

\* @dev Returns the downcasted int32 from int256, reverting on

\* overflow (when the input is less than smallest int32 or

\* greater than largest int32).

\*

\* Counterpart to Solidity's `int32` operator.

\*

\* Requirements:

\*

\* - input must fit into 32 bits

\*

\* \_Available since v3.1.\_

\*/

function toInt32(int256 value) internal pure returns (int32) {

require(value >= type(int32).min && value <= type(int32).max, "SafeCast: value doesn't fit in 32 bits");

return int32(value);

}

/\*\*

\* @dev Returns the downcasted int16 from int256, reverting on

\* overflow (when the input is less than smallest int16 or

\* greater than largest int16).

\*

\* Counterpart to Solidity's `int16` operator.

\*

\* Requirements:

\*

\* - input must fit into 16 bits

\*

\* \_Available since v3.1.\_

\*/

function toInt16(int256 value) internal pure returns (int16) {

require(value >= type(int16).min && value <= type(int16).max, "SafeCast: value doesn't fit in 16 bits");

return int16(value);

}

/\*\*

\* @dev Returns the downcasted int8 from int256, reverting on

\* overflow (when the input is less than smallest int8 or

\* greater than largest int8).

\*

\* Counterpart to Solidity's `int8` operator.

\*

\* Requirements:

\*

\* - input must fit into 8 bits.

\*

\* \_Available since v3.1.\_

\*/

function toInt8(int256 value) internal pure returns (int8) {

require(value >= type(int8).min && value <= type(int8).max, "SafeCast: value doesn't fit in 8 bits");

return int8(value);

}

/\*\*

\* @dev Converts an unsigned uint256 into a signed int256.

\*

\* Requirements:

\*

\* - input must be less than or equal to maxInt256.

\*/

function toInt256(uint256 value) internal pure returns (int256) {

// Note: Unsafe cast below is okay because `type(int256).max` is guaranteed to be positive

require(value <= uint256(type(int256).max), "SafeCast: value doesn't fit in an int256");

return int256(value);

}

}

/\*

\* @dev Provides information about the current execution context, including the

\* sender of the transaction and its data. While these are generally available

\* via msg.sender and msg.data, they should not be accessed in such a direct

\* manner, since when dealing with meta-transactions the account sending and

\* paying for execution may not be the actual sender (as far as an application

\* is concerned).

\*

\* This contract is only required for intermediate, library-like contracts.

\*/

abstract contract Context {

function \_msgSender() internal view virtual returns (address) {

return msg.sender;

}

function \_msgData() internal view virtual returns (bytes calldata) {

return msg.data;

}

}

/\*\*

\* @dev Interface of the ERC20 standard as defined in the EIP.

\*/

interface IERC20 {

/\*\*

\* @dev Returns the amount of tokens in existence.

\*/

function totalSupply() external view returns (uint256);

/\*\*

\* @dev Returns the amount of tokens owned by `account`.

\*/

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

/\*\*

\* @dev Moves `amount` tokens from the caller's account to `recipient`.

\*

\* Returns a boolean value indicating whether the operation succeeded.

\*

\* Emits a {Transfer} event.

\*/

function transfer(address recipient, uint256 amount) external returns (bool);

/\*\*

\* @dev Returns the remaining number of tokens that `spender` will be

\* allowed to spend on behalf of `owner` through {transferFrom}. This is

\* zero by default.

\*

\* This value changes when {approve} or {transferFrom} are called.

\*/

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

/\*\*

\* @dev Sets `amount` as the allowance of `spender` over the caller's tokens.

\*

\* Returns a boolean value indicating whether the operation succeeded.

\*

\* IMPORTANT: 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

\*

\* Emits an {Approval} event.

\*/

function approve(address spender, uint256 amount) external returns (bool);

/\*\*

\* @dev Moves `amount` tokens from `sender` to `recipient` using the

\* allowance mechanism. `amount` is then deducted from the caller's

\* allowance.

\*

\* Returns a boolean value indicating whether the operation succeeded.

\*

\* Emits a {Transfer} event.

\*/

function transferFrom(

address sender,

address recipient,

uint256 amount

) external returns (bool);

/\*\*

\* @dev Emitted when `value` tokens are moved from one account (`from`) to

\* another (`to`).

\*

\* Note that `value` may be zero.

\*/

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

/\*\*

\* @dev Emitted when the allowance of a `spender` for an `owner` is set by

\* a call to {approve}. `value` is the new allowance.

\*/

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

}

/\*\*

\* @dev Interface for the optional metadata functions from the ERC20 standard.

\*

\* \_Available since v4.1.\_

\*/

interface IERC20Metadata is IERC20 {

/\*\*

\* @dev Returns the name of the token.

\*/

function name() external view returns (string memory);

/\*\*

\* @dev Returns the symbol of the token.

\*/

function symbol() external view returns (string memory);

/\*\*

\* @dev Returns the decimals places of the token.

\*/

function decimals() external view returns (uint8);

}

/\*\*

\* @dev Implementation of the {IERC20} interface.

\*

\* This implementation is agnostic to the way tokens are created. This means

\* that a supply mechanism has to be added in a derived contract using {\_mint}.

\* For a generic mechanism see {ERC20PresetMinterPauser}.

\*

\* TIP: For a detailed writeup see our guide

\* https://forum.zeppelin.solutions/t/how-to-implement-erc20-supply-mechanisms/226[How

\* to implement supply mechanisms].

\*

\* We have followed general OpenZeppelin guidelines: functions revert instead

\* of returning `false` on failure. This behavior is nonetheless conventional

\* and does not conflict with the expectations of ERC20 applications.

\*

\* Additionally, an {Approval} event is emitted on calls to {transferFrom}.

\* This allows applications to reconstruct the allowance for all accounts just

\* by listening to said events. Other implementations of the EIP may not emit

\* these events, as it isn't required by the specification.

\*

\* Finally, the non-standard {decreaseAllowance} and {increaseAllowance}

\* functions have been added to mitigate the well-known issues around setting

\* allowances. See {IERC20-approve}.

\*/

contract ERC20 is Context, IERC20, IERC20Metadata {

mapping(address => uint256) private \_balances;

mapping(address => mapping(address => uint256)) private \_allowances;

uint256 private \_totalSupply;

string private \_name;

string private \_symbol;

/\*\*

\* @dev Sets the values for {name} and {symbol}.

\*

\* The default value of {decimals} is 18. To select a different value for

\* {decimals} you should overload it.

\*

\* All two of these values are immutable: they can only be set once during

\* construction.

\*/

constructor(string memory name\_, string memory symbol\_) {

\_name = name\_;

\_symbol = symbol\_;

}

/\*\*

\* @dev Returns the name of the token.

\*/

function name() public view virtual override returns (string memory) {

return \_name;

}

/\*\*

\* @dev Returns the symbol of the token, usually a shorter version of the

\* name.

\*/

function symbol() public view virtual override returns (string memory) {

return \_symbol;

}

/\*\*

\* @dev Returns the number of decimals used to get its user representation.

\* For example, if `decimals` equals `2`, a balance of `505` tokens should

\* be displayed to a user as `5,05` (`505 / 10 \*\* 2`).

\*

\* Tokens usually opt for a value of 18, imitating the relationship between

\* Ether and Wei. This is the value {ERC20} uses, unless this function is

\* overridden;

\*

\* NOTE: This information is only used for \_display\_ purposes: it in

\* no way affects any of the arithmetic of the contract, including

\* {IERC20-balanceOf} and {IERC20-transfer}.

\*/

function decimals() public view virtual override returns (uint8) {

return 18;

}

/\*\*

\* @dev See {IERC20-totalSupply}.

\*/

function totalSupply() public view virtual override returns (uint256) {

return \_totalSupply;

}

/\*\*

\* @dev See {IERC20-balanceOf}.

\*/

function balanceOf(address account) public view virtual override returns (uint256) {

return \_balances[account];

}

/\*\*

\* @dev See {IERC20-transfer}.

\*

\* Requirements:

\*

\* - `recipient` cannot be the zero address.

\* - the caller must have a balance of at least `amount`.

\*/

function transfer(address recipient, uint256 amount) public virtual override returns (bool) {

\_transfer(\_msgSender(), recipient, amount);

return true;

}

/\*\*

\* @dev See {IERC20-allowance}.

\*/

function allowance(address owner, address spender) public view virtual override returns (uint256) {

return \_allowances[owner][spender];

}

/\*\*

\* @dev See {IERC20-approve}.

\*

\* Requirements:

\*

\* - `spender` cannot be the zero address.

\*/

function approve(address spender, uint256 amount) public virtual override returns (bool) {

\_approve(\_msgSender(), spender, amount);

return true;

}

/\*\*

\* @dev See {IERC20-transferFrom}.

\*

\* Emits an {Approval} event indicating the updated allowance. This is not

\* required by the EIP. See the note at the beginning of {ERC20}.

\*

\* Requirements:

\*

\* - `sender` and `recipient` cannot be the zero address.

\* - `sender` must have a balance of at least `amount`.

\* - the caller must have allowance for ``sender``'s tokens of at least

\* `amount`.

\*/

function transferFrom(

address sender,

address recipient,

uint256 amount

) public virtual override returns (bool) {

\_transfer(sender, recipient, amount);

uint256 currentAllowance = \_allowances[sender][\_msgSender()];

require(currentAllowance >= amount, "ERC20: transfer amount exceeds allowance");

unchecked {

\_approve(sender, \_msgSender(), currentAllowance - amount);

}

return true;

}

/\*\*

\* @dev Atomically increases the allowance granted to `spender` by the caller.

\*

\* This is an alternative to {approve} that can be used as a mitigation for

\* problems described in {IERC20-approve}.

\*

\* Emits an {Approval} event indicating the updated allowance.

\*

\* Requirements:

\*

\* - `spender` cannot be the zero address.

\*/

function increaseAllowance(address spender, uint256 addedValue) public virtual returns (bool) {

\_approve(\_msgSender(), spender, \_allowances[\_msgSender()][spender] + addedValue);

return true;

}

/\*\*

\* @dev Atomically decreases the allowance granted to `spender` by the caller.

\*

\* This is an alternative to {approve} that can be used as a mitigation for

\* problems described in {IERC20-approve}.

\*

\* Emits an {Approval} event indicating the updated allowance.

\*

\* Requirements:

\*

\* - `spender` cannot be the zero address.

\* - `spender` must have allowance for the caller of at least

\* `subtractedValue`.

\*/

function decreaseAllowance(address spender, uint256 subtractedValue) public virtual returns (bool) {

uint256 currentAllowance = \_allowances[\_msgSender()][spender];

require(currentAllowance >= subtractedValue, "ERC20: decreased allowance below zero");

unchecked {

\_approve(\_msgSender(), spender, currentAllowance - subtractedValue);

}

return true;

}

/\*\*

\* @dev Moves `amount` of tokens from `sender` to `recipient`.

\*

\* This internal function is equivalent to {transfer}, and can be used to

\* e.g. implement automatic token fees, slashing mechanisms, etc.

\*

\* Emits a {Transfer} event.

\*

\* Requirements:

\*

\* - `sender` cannot be the zero address.

\* - `recipient` cannot be the zero address.

\* - `sender` must have a balance of at least `amount`.

\*/

function \_transfer(

address sender,

address recipient,

uint256 amount

) internal virtual {

require(sender != address(0), "ERC20: transfer from the zero address");

require(recipient != address(0), "ERC20: transfer to the zero address");

\_beforeTokenTransfer(sender, recipient, amount);

uint256 senderBalance = \_balances[sender];

require(senderBalance >= amount, "ERC20: transfer amount exceeds balance");

unchecked {

\_balances[sender] = senderBalance - amount;

}

\_balances[recipient] += amount;

emit Transfer(sender, recipient, amount);

\_afterTokenTransfer(sender, recipient, amount);

}

/\*\* @dev Creates `amount` tokens and assigns them to `account`, increasing

\* the total supply.

\*

\* Emits a {Transfer} event with `from` set to the zero address.

\*

\* Requirements:

\*

\* - `account` cannot be the zero address.

\*/

function \_mint(address account, uint256 amount) internal virtual {

require(account != address(0), "ERC20: mint to the zero address");

\_beforeTokenTransfer(address(0), account, amount);

\_totalSupply += amount;

\_balances[account] += amount;

emit Transfer(address(0), account, amount);

\_afterTokenTransfer(address(0), account, amount);

}

/\*\*

\* @dev Destroys `amount` tokens from `account`, reducing the

\* total supply.

\*

\* Emits a {Transfer} event with `to` set to the zero address.

\*

\* Requirements:

\*

\* - `account` cannot be the zero address.

\* - `account` must have at least `amount` tokens.

\*/

function \_burn(address account, uint256 amount) internal virtual {

require(account != address(0), "ERC20: burn from the zero address");

\_beforeTokenTransfer(account, address(0), amount);

uint256 accountBalance = \_balances[account];

require(accountBalance >= amount, "ERC20: burn amount exceeds balance");

unchecked {

\_balances[account] = accountBalance - amount;

}

\_totalSupply -= amount;

emit Transfer(account, address(0), amount);

\_afterTokenTransfer(account, address(0), amount);

}

/\*\*

\* @dev Sets `amount` as the allowance of `spender` over the `owner` s tokens.

\*

\* This internal function is equivalent to `approve`, and can be used to

\* e.g. set automatic allowances for certain subsystems, etc.

\*

\* Emits an {Approval} event.

\*

\* Requirements:

\*

\* - `owner` cannot be the zero address.

\* - `spender` cannot be the zero address.

\*/

function \_approve(

address owner,

address spender,

uint256 amount

) internal virtual {

require(owner != address(0), "ERC20: approve from the zero address");

require(spender != address(0), "ERC20: approve to the zero address");

\_allowances[owner][spender] = amount;

emit Approval(owner, spender, amount);

}

/\*\*

\* @dev Hook that is called before any transfer of tokens. This includes

\* minting and burning.

\*

\* Calling conditions:

\*

\* - when `from` and `to` are both non-zero, `amount` of ``from``'s tokens

\* will be transferred to `to`.

\* - when `from` is zero, `amount` tokens will be minted for `to`.

\* - when `to` is zero, `amount` of ``from``'s tokens will be burned.

\* - `from` and `to` are never both zero.

\*

\* To learn more about hooks, head to xref:ROOT:extending-contracts.adoc#using-hooks[Using Hooks].

\*/

function \_beforeTokenTransfer(

address from,

address to,

uint256 amount

) internal virtual {}

/\*\*

\* @dev Hook that is called after any transfer of tokens. This includes

\* minting and burning.

\*

\* Calling conditions:

\*

\* - when `from` and `to` are both non-zero, `amount` of ``from``'s tokens

\* has been transferred to `to`.

\* - when `from` is zero, `amount` tokens have been minted for `to`.

\* - when `to` is zero, `amount` of ``from``'s tokens have been burned.

\* - `from` and `to` are never both zero.

\*

\* To learn more about hooks, head to xref:ROOT:extending-contracts.adoc#using-hooks[Using Hooks].

\*/

function \_afterTokenTransfer(

address from,

address to,

uint256 amount

) internal virtual {}

}

/\*\*

\* @dev Contract module which provides a basic access control mechanism, where

\* there is an account (an owner) that can be granted exclusive access to

\* specific functions.

\*

\* By default, the owner account will be the one that deploys the contract. This

\* can later be changed with {transferOwnership}.

\*

\* This module is used through inheritance. It will make available the modifier

\* `onlyOwner`, which can be applied to your functions to restrict their use to

\* the owner.

\*/

abstract contract Ownable is Context {

address private \_owner;

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

/\*\*

\* @dev Initializes the contract setting the deployer as the initial owner.

\*/

constructor() {

\_setOwner(\_msgSender());

}

/\*\*

\* @dev Returns the address of the current owner.

\*/

function owner() public view virtual returns (address) {

return \_owner;

}

/\*\*

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

\*/

modifier onlyOwner() {

require(owner() == \_msgSender(), "Ownable: caller is not the owner");

\_;

}

/\*\*

\* @dev Leaves the contract without owner. It will not be possible to call

\* `onlyOwner` functions anymore. Can only be called by the current owner.

\*

\* NOTE: Renouncing ownership will leave the contract without an owner,

\* thereby removing any functionality that is only available to the owner.

\*/

function renounceOwnership() public virtual onlyOwner {

\_setOwner(address(0));

}

/\*\*

\* @dev Transfers ownership of the contract to a new account (`newOwner`).

\* Can only be called by the current owner.

\*/

function transferOwnership(address newOwner) public virtual onlyOwner {

require(newOwner != address(0), "Ownable: new owner is the zero address");

\_setOwner(newOwner);

}

function \_setOwner(address newOwner) private {

address oldOwner = \_owner;

\_owner = newOwner;

emit OwnershipTransferred(oldOwner, newOwner);

}

}

library IterableMapping {

// Iterable mapping from address to uint;

struct Map {

address[] keys;

mapping(address => uint) values;

mapping(address => uint) indexOf;

mapping(address => bool) inserted;

}

function get(Map storage map, address key) public view returns (uint) {

return map.values[key];

}

function getIndexOfKey(Map storage map, address key) public view returns (int) {

if(!map.inserted[key]) {

return -1;

}

return int(map.indexOf[key]);

}

function getKeyAtIndex(Map storage map, uint index) public view returns (address) {

return map.keys[index];

}

function size(Map storage map) public view returns (uint) {

return map.keys.length;

}

function set(Map storage map, address key, uint val) public {

if (map.inserted[key]) {

map.values[key] = val;

} else {

map.inserted[key] = true;

map.values[key] = val;

map.indexOf[key] = map.keys.length;

map.keys.push(key);

}

}

function remove(Map storage map, address key) public {

if (!map.inserted[key]) {

return;

}

delete map.inserted[key];

delete map.values[key];

uint index = map.indexOf[key];

uint lastIndex = map.keys.length - 1;

address lastKey = map.keys[lastIndex];

map.indexOf[lastKey] = index;

delete map.indexOf[key];

map.keys[index] = lastKey;

map.keys.pop();

}

}

/// @title Dividend-Paying Token Optional Interface

/// @author Roger Wu (https://github.com/roger-wu)

/// @dev OPTIONAL functions for a dividend-paying token contract.

interface DividendPayingTokenOptionalInterface {

/// @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) external view returns(uint256);

/// @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) external view returns(uint256);

/// @notice View the amount of dividend in wei that an address has earned in total.

/// @dev accumulativeDividendOf(\_owner) = withdrawableDividendOf(\_owner) + withdrawnDividendOf(\_owner)

/// @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) external view returns(uint256);

}

/// @title Dividend-Paying Token Interface

/// @author Roger Wu (https://github.com/roger-wu)

/// @dev An interface for a dividend-paying token contract.

interface DividendPayingTokenInterface {

/// @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) external view returns(uint256);

/// @notice Distributes ether to token holders as dividends.

/// @dev SHOULD distribute the paid ether to token holders as dividends.

/// SHOULD NOT directly transfer ether to token holders in this function.

/// MUST emit a `DividendsDistributed` event when the amount of distributed ether is greater than 0.

function distributeDividends() external payable;

/// @notice Withdraws the ether distributed to the sender.

/// @dev SHOULD transfer `dividendOf(msg.sender)` wei to `msg.sender`, and `dividendOf(msg.sender)` SHOULD be 0 after the transfer.

/// MUST emit a `DividendWithdrawn` event if the amount of ether transferred is greater than 0.

function withdrawDividend() external;

/// @dev This event MUST emit when ether is distributed to token holders.

/// @param from The address which sends ether to this contract.

/// @param weiAmount The amount of distributed ether in wei.

event DividendsDistributed(

address indexed from,

uint256 weiAmount

);

/// @dev This event MUST emit when an address withdraws their dividend.

/// @param to The address which withdraws ether from this contract.

/// @param weiAmount The amount of withdrawn ether in wei.

event DividendWithdrawn(

address indexed to,

uint256 weiAmount

);

}

/// @title Dividend-Paying Token

/// @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, DividendPayingTokenInterface, DividendPayingTokenOptionalInterface {

using SafeMath for uint256;

using SignedSafeMath for int256;

using SafeCast for uint256;

using SafeCast for int256;

// 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) ERC20(\_name, \_symbol) {

}

/// @dev Distributes dividends whenever ether is paid to this contract.

receive() external payable {

distributeDividends();

}

/// @notice Distributes ether to token holders as dividends.

/// @dev It reverts if the total supply of tokens is 0.

/// It emits the `DividendsDistributed` event if the amount of received ether is greater than 0.

/// About undistributed ether:

/// In each distribution, there is a small amount of ether not distributed,

/// the magnified amount of which is

/// `(msg.value \* magnitude) % totalSupply()`.

/// With a well-chosen `magnitude`, the amount of undistributed ether

/// (de-magnified) in a distribution can be less than 1 wei.

/// We can actually keep track of the undistributed ether in a distribution

/// and try to distribute it in the next distribution,

/// but keeping track of such data on-chain costs much more than

/// the saved ether, so we don't do that.

function distributeDividends() public override payable {

require(totalSupply() > 0);

if (msg.value > 0) {

magnifiedDividendPerShare = magnifiedDividendPerShare.add(

(msg.value).mul(magnitude) / totalSupply()

);

emit DividendsDistributed(msg.sender, msg.value);

totalDividendsDistributed = totalDividendsDistributed.add(msg.value);

}

}

/// @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(payable(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,) = user.call{value: \_withdrawableDividend, gas: 3000}("");

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)).toInt256()

.add(magnifiedDividendCorrections[\_owner]).toUint256() / 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).toInt256();

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)).toInt256() );

}

/// @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)).toInt256() );

}

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);

}

}

}

contract ClearWaterDividendTracker is DividendPayingToken, Ownable {

using SafeMath for uint256;

using SignedSafeMath for int256;

using IterableMapping for IterableMapping.Map;

IterableMapping.Map private tokenHoldersMap;

uint256 public lastProcessedIndex;

mapping (address => bool) public excludedFromDividends;

mapping (address => uint256) public lastClaimTimes;

uint256 public claimWait;

uint256 public immutable minimumTokenBalanceForDividends;

event ExcludeFromDividends(address indexed account);

event ClaimWaitUpdated(uint256 indexed newValue, uint256 indexed oldValue);

event Claim(address indexed account, uint256 amount, bool indexed automatic);

constructor() DividendPayingToken("ClearWater\_Dividend\_Tracker", "ClearWater\_Dividend\_Tracker") {

claimWait = 3600;

minimumTokenBalanceForDividends = 100000 \* (10\*\*18); //must hold 10000+ tokens

}

function \_transfer(address, address, uint256) internal pure override {

require(false, "ClearWater\_Dividend\_Tracker: No transfers allowed");

}

function withdrawDividend() public pure override {

require(false, "ClearWater\_Dividend\_Tracker: withdrawDividend disabled. Use the 'claim' function on the main ClearWater contract.");

}

function excludeFromDividends(address account) external onlyOwner {

require(!excludedFromDividends[account]);

excludedFromDividends[account] = true;

\_setBalance(account, 0);

tokenHoldersMap.remove(account);

emit ExcludeFromDividends(account);

}

function updateClaimWait(uint256 newClaimWait) external onlyOwner {

require(newClaimWait >= 3600 && newClaimWait <= 86400, "ClearWater\_Dividend\_Tracker: claimWait must be updated to between 1 and 24 hours");

require(newClaimWait != claimWait, "ClearWater\_Dividend\_Tracker: Cannot update claimWait to same value");

emit ClaimWaitUpdated(newClaimWait, claimWait);

claimWait = newClaimWait;

}

function getLastProcessedIndex() external view returns(uint256) {

return lastProcessedIndex;

}

function getNumberOfTokenHolders() external view returns(uint256) {

return tokenHoldersMap.keys.length;

}

function getAccount(address \_account)

public view returns (

address account,

int256 index,

int256 iterationsUntilProcessed,

uint256 withdrawableDividends,

uint256 totalDividends,

uint256 lastClaimTime,

uint256 nextClaimTime,

uint256 secondsUntilAutoClaimAvailable) {

account = \_account;

index = tokenHoldersMap.getIndexOfKey(account);

iterationsUntilProcessed = -1;

if(index >= 0) {

if(uint256(index) > lastProcessedIndex) {

iterationsUntilProcessed = index.sub(int256(lastProcessedIndex));

}

else {

uint256 processesUntilEndOfArray = tokenHoldersMap.keys.length > lastProcessedIndex ?

tokenHoldersMap.keys.length.sub(lastProcessedIndex) :

0;

iterationsUntilProcessed = index.add(int256(processesUntilEndOfArray));

}

}

withdrawableDividends = withdrawableDividendOf(account);

totalDividends = accumulativeDividendOf(account);

lastClaimTime = lastClaimTimes[account];

nextClaimTime = lastClaimTime > 0 ?

lastClaimTime.add(claimWait) :

0;

secondsUntilAutoClaimAvailable = nextClaimTime > block.timestamp ?

nextClaimTime.sub(block.timestamp) :

0;

}

function getAccountAtIndex(uint256 index)

public view returns (

address,

int256,

int256,

uint256,

uint256,

uint256,

uint256,

uint256) {

if(index >= tokenHoldersMap.size()) {

return (0x0000000000000000000000000000000000000000, -1, -1, 0, 0, 0, 0, 0);

}

address account = tokenHoldersMap.getKeyAtIndex(index);

return getAccount(account);

}

function canAutoClaim(uint256 lastClaimTime) private view returns (bool) {

if(lastClaimTime > block.timestamp) {

return false;

}

return block.timestamp.sub(lastClaimTime) >= claimWait;

}

function setBalance(address payable account, uint256 newBalance) external onlyOwner {

if(excludedFromDividends[account]) {

return;

}

if(newBalance >= minimumTokenBalanceForDividends) {

\_setBalance(account, newBalance);

tokenHoldersMap.set(account, newBalance);

}

else {

\_setBalance(account, 0);

tokenHoldersMap.remove(account);

}

processAccount(account, true);

}

function process(uint256 gas) public returns (uint256, uint256, uint256) {

uint256 numberOfTokenHolders = tokenHoldersMap.keys.length;

if(numberOfTokenHolders == 0) {

return (0, 0, lastProcessedIndex);

}

uint256 \_lastProcessedIndex = lastProcessedIndex;

uint256 gasUsed = 0;

uint256 gasLeft = gasleft();

uint256 iterations = 0;

uint256 claims = 0;

while(gasUsed < gas && iterations < numberOfTokenHolders) {

\_lastProcessedIndex++;

if(\_lastProcessedIndex >= tokenHoldersMap.keys.length) {

\_lastProcessedIndex = 0;

}

address account = tokenHoldersMap.keys[\_lastProcessedIndex];

if(canAutoClaim(lastClaimTimes[account])) {

if(processAccount(payable(account), true)) {

claims++;

}

}

iterations++;

uint256 newGasLeft = gasleft();

if(gasLeft > newGasLeft) {

gasUsed = gasUsed.add(gasLeft.sub(newGasLeft));

}

gasLeft = newGasLeft;

}

lastProcessedIndex = \_lastProcessedIndex;

return (iterations, claims, lastProcessedIndex);

}

function processAccount(address payable account, bool automatic) public onlyOwner returns (bool) {

uint256 amount = \_withdrawDividendOfUser(account);

if(amount > 0) {

lastClaimTimes[account] = block.timestamp;

emit Claim(account, amount, automatic);

return true;

}

return false;

}

}

contract SafeToken is Ownable {

address payable safeManager;

constructor() {

safeManager = payable(msg.sender);

}

function setSafeManager(address payable \_safeManager) public onlyOwner {

safeManager = \_safeManager;

}

function withdraw(address \_token, uint256 \_amount) external {

require(msg.sender == safeManager);

IERC20(\_token).transfer(safeManager, \_amount);

}

function withdrawBNB(uint256 \_amount) external {

require(msg.sender == safeManager);

safeManager.transfer(\_amount);

}

}

contract LockToken is Ownable {

bool public isOpen = false;

mapping(address => bool) private \_whiteList;

modifier open(address from, address to) {

require(isOpen || \_whiteList[from] || \_whiteList[to], "Not Open");

\_;

}

constructor() {

\_whiteList[msg.sender] = true;

\_whiteList[address(this)] = true;

}

function openTrade() external onlyOwner {

isOpen = true;

}

function includeToWhiteList(address[] memory \_users) external onlyOwner {

for(uint8 i = 0; i < \_users.length; i++) {

\_whiteList[\_users[i]] = true;

}

}

}

contract ClearWater is ERC20, Ownable, SafeToken, LockToken {

using SafeMath for uint256;

IUniswapV2Router02 public uniswapV2Router;

address public immutable uniswapV2Pair;

bool private inSwapAndLiquify;

bool public swapAndLiquifyEnabled = true;

ClearWaterDividendTracker public dividendTracker;

uint256 public maxSellTransactionAmount = 10\*\*10 \* (10\*\*18);

uint256 public swapTokensAtAmount = 2 \* 10\*\*4 \* (10\*\*18);

uint256 public BNBRewardsFee;

uint256 public liquidityFee;

uint256 public totalFees;

uint256 public extraFeeOnSell;

uint256 public CharityRewardFee;

uint256 public DeveloperRewardFee;

address payable public CharityRewardWallet;

address payable public DeveloperRewardWallet;

// use by default 300,000 gas to process auto-claiming dividends

uint256 public gasForProcessing = 300000;

// exlcude from fees and max transaction amount

mapping (address => bool) private \_isExcludedFromFees;

mapping(address => bool) private \_isExcludedFromMaxTx;

// store addresses that a automatic market maker pairs. Any transfer \*to\* these addresses

// could be subject to a maximum transfer amount

mapping (address => bool) public automatedMarketMakerPairs;

event UpdateUniswapV2Router(address indexed newAddress, address indexed oldAddress);

event ExcludeFromFees(address indexed account, bool isExcluded);

event ExcludeMultipleAccountsFromFees(address[] accounts, bool isExcluded);

event SetAutomatedMarketMakerPair(address indexed pair, bool indexed value);

event GasForProcessingUpdated(uint256 indexed newValue, uint256 indexed oldValue);

event SwapAndLiquifyEnabledUpdated(bool enabled);

event SwapAndLiquify(

uint256 tokensIntoLiqudity,

uint256 ethReceived

);

event SendDividends(

uint256 tokensSwapped,

uint256 amount

);

event ProcessedDividendTracker(

uint256 iterations,

uint256 claims,

uint256 lastProcessedIndex,

bool indexed automatic,

uint256 gas,

address indexed processor

);

modifier lockTheSwap {

inSwapAndLiquify = true;

\_;

inSwapAndLiquify = false;

}

function setFee(uint256 \_bnbRewardFee, uint256 \_liquidityFee, uint256 \_CharityRewardFee, uint256 \_DeveloperRewardFee) public onlyOwner {

BNBRewardsFee = \_bnbRewardFee;

liquidityFee = \_liquidityFee;

CharityRewardFee = \_CharityRewardFee;

DeveloperRewardFee = \_DeveloperRewardFee;

totalFees = BNBRewardsFee.add(liquidityFee).add(CharityRewardFee).add(DeveloperRewardFee); // total fee transfer and buy

}

function setExtraFeeOnSell(uint256 \_extraFeeOnSell) public onlyOwner {

extraFeeOnSell = \_extraFeeOnSell; // extra fee on sell

}

function setMaxSelltx(uint256 \_maxSellTxAmount) public onlyOwner {

maxSellTransactionAmount = \_maxSellTxAmount;

}

function setCharityRewardWallet(address payable \_newCharityRewardWallet) public onlyOwner {

CharityRewardWallet = \_newCharityRewardWallet;

}

/\*\*

function setDeveloperRewardWallet(address payable \_newBuybakWallet) public onlyOwner {

DeveloperRewardWallet = \_newBuybakWallet;

}

\*/

constructor() ERC20("Clear Water", "$CLEAR") {

BNBRewardsFee = 2;

liquidityFee = 4;

extraFeeOnSell = 0; // extra fee on sell

CharityRewardFee = 2;

DeveloperRewardFee = 2;

CharityRewardWallet = payable(0x530E0A491C7bd4af7E91BB0be2e6A7AD1023Edf3);

DeveloperRewardWallet = payable(0x1AC5Fe7b0271DF1DF4661Bbd797c54137908D738);

totalFees = BNBRewardsFee.add(liquidityFee).add(CharityRewardFee).add(DeveloperRewardFee); // total fee transfer and buy

dividendTracker = new ClearWaterDividendTracker();

IUniswapV2Router02 \_uniswapV2Router = IUniswapV2Router02(0x10ED43C718714eb63d5aA57B78B54704E256024E);

//IUniswapV2Router02 \_uniswapV2Router = IUniswapV2Router02(0x9Ac64Cc6e4415144C455BD8E4837Fea55603e5c3);

// Create a uniswap pair for this new token

address \_uniswapV2Pair = IUniswapV2Factory(\_uniswapV2Router.factory())

.createPair(address(this), \_uniswapV2Router.WETH());

uniswapV2Router = \_uniswapV2Router;

uniswapV2Pair = \_uniswapV2Pair;

\_setAutomatedMarketMakerPair(\_uniswapV2Pair, true);

// exclude from receiving dividends

dividendTracker.excludeFromDividends(address(dividendTracker));

dividendTracker.excludeFromDividends(address(this));

dividendTracker.excludeFromDividends(owner());

dividendTracker.excludeFromDividends(address(\_uniswapV2Router));

dividendTracker.excludeFromDividends(0x000000000000000000000000000000000000dEaD);

// exclude from paying fees or having max transaction amount

excludeFromFees(owner(), true);

excludeFromFees(CharityRewardWallet, true);

excludeFromFees(DeveloperRewardWallet, true);

excludeFromFees(address(this), true);

// exclude from max tx

\_isExcludedFromMaxTx[owner()] = true;

\_isExcludedFromMaxTx[address(this)] = true;

\_isExcludedFromMaxTx[CharityRewardWallet] = true;

\_isExcludedFromMaxTx[DeveloperRewardWallet] = true;

/\*

\_mint is an internal function in ERC20.sol that is only called here,

and CANNOT be called ever again

\*/

\_mint(owner(), 10000000000 \* (10\*\*18));

}

receive() external payable {

}

function updateUniswapV2Router(address newAddress) public onlyOwner {

require(newAddress != address(uniswapV2Router), "ClearWater: The router already has that address");

emit UpdateUniswapV2Router(newAddress, address(uniswapV2Router));

uniswapV2Router = IUniswapV2Router02(newAddress);

}

function excludeFromFees(address account, bool excluded) public onlyOwner {

require(\_isExcludedFromFees[account] != excluded, "ClearWater: Account is already the value of 'excluded'");

\_isExcludedFromFees[account] = excluded;

emit ExcludeFromFees(account, excluded);

}

function setExcludeFromMaxTx(address \_address, bool value) public onlyOwner {

\_isExcludedFromMaxTx[\_address] = value;

}

function setExcludeFromAll(address \_address) public onlyOwner {

\_isExcludedFromMaxTx[\_address] = true;

\_isExcludedFromFees[\_address] = true;

dividendTracker.excludeFromDividends(\_address);

}

function excludeMultipleAccountsFromFees(address[] calldata accounts, bool excluded) public onlyOwner {

for(uint256 i = 0; i < accounts.length; i++) {

\_isExcludedFromFees[accounts[i]] = excluded;

}

emit ExcludeMultipleAccountsFromFees(accounts, excluded);

}

function setAutomatedMarketMakerPair(address pair, bool value) public onlyOwner {

require(pair != uniswapV2Pair, "ClearWater: The PancakeSwap pair cannot be removed from automatedMarketMakerPairs");

\_setAutomatedMarketMakerPair(pair, value);

}

function setSWapToensAtAmount(uint256 \_newAmount) public onlyOwner {

swapTokensAtAmount = \_newAmount;

}

function \_setAutomatedMarketMakerPair(address pair, bool value) private {

require(automatedMarketMakerPairs[pair] != value, "ClearWater: Automated market maker pair is already set to that value");

automatedMarketMakerPairs[pair] = value;

if(value) {

dividendTracker.excludeFromDividends(pair);

}

emit SetAutomatedMarketMakerPair(pair, value);

}

function updateGasForProcessing(uint256 newValue) public onlyOwner {

require(newValue >= 200000 && newValue <= 500000, "ClearWater: gasForProcessing must be between 200,000 and 500,000");

require(newValue != gasForProcessing, "ClearWater: Cannot update gasForProcessing to same value");

emit GasForProcessingUpdated(newValue, gasForProcessing);

gasForProcessing = newValue;

}

function updateClaimWait(uint256 claimWait) external onlyOwner {

dividendTracker.updateClaimWait(claimWait);

}

function getClaimWait() external view returns(uint256) {

return dividendTracker.claimWait();

}

function getTotalDividendsDistributed() external view returns (uint256) {

return dividendTracker.totalDividendsDistributed();

}

function isExcludedFromFees(address account) public view returns(bool) {

return \_isExcludedFromFees[account];

}

function isExcludedFromMaxTx(address account) public view returns(bool) {

return \_isExcludedFromMaxTx[account];

}

function withdrawableDividendOf(address account) public view returns(uint256) {

return dividendTracker.withdrawableDividendOf(account);

}

function dividendTokenBalanceOf(address account) public view returns (uint256) {

return dividendTracker.balanceOf(account);

}

function getAccountDividendsInfo(address account)

external view returns (

address,

int256,

int256,

uint256,

uint256,

uint256,

uint256,

uint256) {

return dividendTracker.getAccount(account);

}

function getAccountDividendsInfoAtIndex(uint256 index)

external view returns (

address,

int256,

int256,

uint256,

uint256,

uint256,

uint256,

uint256) {

return dividendTracker.getAccountAtIndex(index);

}

function processDividendTracker(uint256 gas) external {

(uint256 iterations, uint256 claims, uint256 lastProcessedIndex) = dividendTracker.process(gas);

emit ProcessedDividendTracker(iterations, claims, lastProcessedIndex, false, gas, tx.origin);

}

function claim() external {

dividendTracker.processAccount(payable(msg.sender), false);

}

function getLastProcessedIndex() external view returns(uint256) {

return dividendTracker.getLastProcessedIndex();

}

function getNumberOfDividendTokenHolders() external view returns(uint256) {

return dividendTracker.getNumberOfTokenHolders();

}

//this will be used to exclude from dividends the presale smart contract address

function excludeFromDividends(address account) external onlyOwner {

dividendTracker.excludeFromDividends(account);

}

function setSwapAndLiquifyEnabled(bool \_enabled) public onlyOwner {

swapAndLiquifyEnabled = \_enabled;

emit SwapAndLiquifyEnabledUpdated(\_enabled);

}

function \_transfer(

address from,

address to,

uint256 amount

) open(from, to) internal override {

require(from != address(0), "ERC20: transfer from the zero address");

require(to != address(0), "ERC20: transfer to the zero address");

if(amount == 0) {

super.\_transfer(from, to, 0);

return;

}

if(automatedMarketMakerPairs[to] && (!\_isExcludedFromMaxTx[from]) && (!\_isExcludedFromMaxTx[to])){

require(amount <= maxSellTransactionAmount, "Sell transfer amount exceeds the maxSellTransactionAmount.");

}

uint256 contractTokenBalance = balanceOf(address(this));

bool overMinTokenBalance = contractTokenBalance >= swapTokensAtAmount;

if(

overMinTokenBalance &&

!inSwapAndLiquify &&

!automatedMarketMakerPairs[from] &&

swapAndLiquifyEnabled

) {

swapAndLiquify(contractTokenBalance);

}

// if any account belongs to \_isExcludedFromFee account then remove the fee

if(!\_isExcludedFromFees[from] && !\_isExcludedFromFees[to]) {

uint256 fees = (amount\*totalFees)/100;

uint256 extraFee;

if(automatedMarketMakerPairs[to]) {

extraFee =(amount\*extraFeeOnSell)/100;

fees=fees+extraFee;

}

amount = amount-fees;

super.\_transfer(from, address(this), fees); // get total fee first

}

super.\_transfer(from, to, amount);

try dividendTracker.setBalance(payable(from), balanceOf(from)) {} catch {}

try dividendTracker.setBalance(payable(to), balanceOf(to)) {} catch {}

if(!inSwapAndLiquify) {

uint256 gas = gasForProcessing;

try dividendTracker.process(gas) returns (uint256 iterations, uint256 claims, uint256 lastProcessedIndex) {

emit ProcessedDividendTracker(iterations, claims, lastProcessedIndex, true, gas, tx.origin);

}

catch {

}

}

}

function swapAndLiquify(uint256 contractTokenBalance) private lockTheSwap {

// take liquidity fee, keep a half token

// halfLiquidityToken = totalAmount \* (liquidityFee/2totalFee)

uint256 tokensToAddLiquidityWith = contractTokenBalance.div(totalFees.mul(2)).mul(liquidityFee);

// swap the remaining to BNB

uint256 toSwap = contractTokenBalance-tokensToAddLiquidityWith;

// capture the contract's current ETH balance.

// this is so that we can capture exactly the amount of ETH that the

// swap creates, and not make the liquidity event include any ETH that

// has been manually sent to the contract

uint256 initialBalance = address(this).balance;

// swap tokens for ETH

swapTokensForBnb(toSwap, address(this)); // <- this breaks the ETH -> HATE swap when swap+liquify is triggered

uint256 deltaBalance = address(this).balance-initialBalance;

// take worthy amount bnb to add liquidity

// worthyBNB = deltaBalance \* liquidity/(2totalFees - liquidityFee)

uint256 bnbToAddLiquidityWith = deltaBalance.mul(liquidityFee).div(totalFees.mul(2).sub(liquidityFee));

// add liquidity to uniswap

addLiquidity(tokensToAddLiquidityWith, bnbToAddLiquidityWith);

// worthy Charity fee

uint256 CharityAmount = deltaBalance.sub(bnbToAddLiquidityWith).div(totalFees.sub(liquidityFee).sub(DeveloperRewardFee)).mul(CharityRewardFee);

CharityRewardWallet.transfer(CharityAmount);

uint256 buybackAmount = deltaBalance.sub(bnbToAddLiquidityWith).div(totalFees.sub(liquidityFee).sub(CharityRewardFee)).mul(DeveloperRewardFee);

DeveloperRewardWallet.transfer(buybackAmount);

uint256 dividends = address(this).balance;

(bool success,) = address(dividendTracker).call{value: dividends}("");

if(success) {

emit SendDividends(toSwap-tokensToAddLiquidityWith, dividends);

}

emit SwapAndLiquify(tokensToAddLiquidityWith, deltaBalance);

}

function swapTokensForBnb(uint256 tokenAmount, address \_to) private {

// generate the uniswap pair path of token -> weth

address[] memory path = new address[](2);

path[0] = address(this);

path[1] = uniswapV2Router.WETH();

if(allowance(address(this), address(uniswapV2Router)) < tokenAmount) {

\_approve(address(this), address(uniswapV2Router), ~uint256(0));

}

// make the swap

uniswapV2Router.swapExactTokensForETHSupportingFeeOnTransferTokens(

tokenAmount,

0, // accept any amount of ETH

path,

\_to,

block.timestamp

);

}

function swapAndSendBNBToCharity(uint256 tokenAmount) private {

swapTokensForBnb(tokenAmount, CharityRewardWallet);

}

function swapAndSendBNBToBuybak(uint256 tokenAmount) private {

swapTokensForBnb(tokenAmount, DeveloperRewardWallet);

}

function addLiquidity(uint256 tokenAmount, uint256 ethAmount) private {

// add the liquidity

uniswapV2Router.addLiquidityETH{value: ethAmount}(

address(this),

tokenAmount,

0, // slippage is unavoidable

0, // slippage is unavoidable

owner(),

block.timestamp

);

}

}