```
// SPDX-License-Identifier: MIT

pragma solidity ^0.8.6;

import "./DividendPayingToken.sol";
import "./IterableMapping.sol";
import "./Ownable.sol";
import "./IDex.sol";
import "./IERC20.sol";


library Address{
    function sendValue(address payable recipient, uint256 amount) internal {
        require(address(this).balance >= amount, "Address: insufficient balance");

        (bool success, ) = recipient.call{value: amount}("");
        require(success, "Address: unable to send value, recipient may have reverted");
    }
}


contract FLOKIVADER is ERC20, Ownable {
    using Address for address payable;

    IRouter public router;
    address public  pair;

    bool private swapping;
    bool public swapEnabled = true;

    FLOKIVADERDividendTracker public dividendTracker;
```

```
address public marketingWallet = 0x91146d6D4bF42A12ccE6e97467F2E9F48f42a974 ;
address public charityWallet = 0x850ccE9010f6f4991172B6E5A6c5A5052E44aDe2;
address public autoBoostWallet = 0xC9e212BE0C405a298CC8B4abb624662F65DF52Ad;

uint256 public swapTokensAtAmount = 200_000_000 * 10**9;


      ///////////////
      //   Fees    //
     ///////////////

struct Taxes {
   uint256 rewards;
   uint256 marketing;
   uint256 autoBoost;
   uint256 charity;
}

Taxes public buyTaxes = Taxes(2,4,5,1);
Taxes public sellTaxes = Taxes(2,6,6,1);
Taxes public transferTaxes = Taxes(0,5,0,0);

uint256 public totalBuyTax = 12;
uint256 public totalSellTax = 15;
uint256 public totalTransferTax = 5;

// use by default 300,000 gas to process auto-claiming dividends
uint256 public gasForProcessing = 300000;

mapping (address => bool) private _isExcludedFromFees;
```

```
    mapping (address => bool) public automatedMarketMakerPairs;

     ///////////////
     //  Events  //
     ///////////////

    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 SendDividends(uint256 tokensSwapped,uint256 amount);
    event ProcessedDividendTracker(uint256 iterations,uint256 claims,uint256
lastProcessedIndex,bool indexed automatic,uint256 gas,address indexed processor);


    constructor() ERC20("Floki Vader", "FLOKIVADER") {

          dividendTracker = new FLOKIVADERDividendTracker();

          IRouter _router = IRouter(0x10ED43C718714eb63d5aA57B78B54704E256024E);
        address _pair = IFactory(_router.factory()).createPair(address(this), _router.WETH());

        router = _router;
        pair = _pair;

        _setAutomatedMarketMakerPair(_pair, true);


        // exclude from receiving dividends
        dividendTracker.excludeFromDividends(address(dividendTracker), true);
        dividendTracker.excludeFromDividends(address(this), true);
```

```
        dividendTracker.excludeFromDividends(owner(), true);
        dividendTracker.excludeFromDividends(address(0xdead), true);
        dividendTracker.excludeFromDividends(address(_router), true);

        // exclude from paying fees or having max transaction amount
        excludeFromFees(owner(), true);
        excludeFromFees(address(this), true);
        excludeFromFees(marketingWallet, true);
        excludeFromFees(charityWallet, true);
        excludeFromFees(autoBoostWallet, true);

        /*
            _mint is an internal function in ERC20.sol that is only called here,
            and CANNOT be called ever again
        */
        _mint(owner(), 1e15* (10**9));
    }

    receive() external payable {}
    function updateDividendTracker(address newAddress) public onlyOwner {
        FLOKIVADERDividendTracker newDividendTracker =
FLOKIVADERDividendTracker(payable(newAddress));

        newDividendTracker.excludeFromDividends(address(newDividendTracker), true);
        newDividendTracker.excludeFromDividends(address(this), true);
        newDividendTracker.excludeFromDividends(owner(), true);
        newDividendTracker.excludeFromDividends(address(router), true);
        dividendTracker = newDividendTracker;
    }

    function processDividendTracker(uint256 gas) external {
```

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

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

    }


    /// @notice Manual claim the dividends after claimWait is passed

    ///    This can be useful during low volume days.

    function claim() external {

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

    }


    /// @notice Withdraw tokens sent by mistake.

    /// @param tokenAddress The address of the token to withdraw

    function rescueBEP20Tokens(address tokenAddress) external onlyOwner{

        IERC20(tokenAddress).transfer(msg.sender,
IERC20(tokenAddress).balanceOf(address(this)));

    }


    /// @notice Send remaining BNB to marketingWallet

    /// @dev It will send all BNB to marketingWallet

    function forceSend() external {

        uint256 BNBbalance = address(this).balance;

        payable(marketingWallet).sendValue(BNBbalance);

    }


    function updateRouter(address newRouter) external onlyOwner{

        router = IRouter(newRouter);

    }



     ////////////////////////////////
```

```
    // Exclude / Include functions //
    ////////////////////////////////

    function excludeFromFees(address account, bool excluded) public onlyOwner {
        require(_isExcludedFromFees[account] != excluded, "FLOKIVADER: Account is already the value of 'excluded'");
        _isExcludedFromFees[account] = excluded;

        emit ExcludeFromFees(account, excluded);
    }

    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);
    }

    /// @dev "true" to exlcude, "false" to include
    function excludeFromDividends(address account, bool value) external onlyOwner{
                dividendTracker.excludeFromDividends(account, value);
            }


     ////////////////////////
    //  Setter Functions //
    ////////////////////////

    function setMarketingWallet(address newWallet) external onlyOwner{
        marketingWallet = newWallet;
```

```
    }

    function setCharityWallet(address newWallet) external onlyOwner{
        charityWallet = newWallet;
    }

    function setAutoBoostWallet(address newWallet) external onlyOwner{
        autoBoostWallet = newWallet;
    }

    /// @notice Update the threshold to swap tokens for liquidity,
    ///   marketing and dividends.
    function setSwapTokensAtAmount(uint256 amount) external onlyOwner{
        swapTokensAtAmount = amount * 10**9;
    }

    function setBuyTaxes(uint256 _rewards, uint256 _marketing, uint256 _autoBoost, uint256
_charity) external onlyOwner{
        buyTaxes = Taxes(_rewards, _marketing, _autoBoost, _charity);
        totalBuyTax = _rewards + _marketing + _autoBoost +_charity;
    }

    function setTransferTaxes(uint256 _rewards, uint256 _marketing, uint256 _autoBoost,
uint256 _charity) external onlyOwner{
        transferTaxes = Taxes(_rewards, _marketing, _autoBoost, _charity);
        totalTransferTax = _rewards + _marketing + _autoBoost +_charity;
    }

    function setSellTaxes(uint256 _rewards, uint256 _marketing, uint256 _autoBoost,uint256
_charity) external onlyOwner{
        sellTaxes = Taxes(_rewards, _marketing, _autoBoost, _charity);
        totalSellTax = _rewards + _marketing + _autoBoost +_charity;
```

```
    }

    /// @notice Enable or disable internal swaps
    /// @dev Set "true" to enable internal swaps for liquidity, marketing and dividends
    function setSwapEnabled(bool _enabled) external onlyOwner{
        swapEnabled = _enabled;
    }



    /// @dev Set new pairs created due to listing in new DEX
    function setAutomatedMarketMakerPair(address newPair, bool value) external onlyOwner {
        _setAutomatedMarketMakerPair(newPair, value);
    }

    function setMinBalanceForDividends(uint256 amount) external onlyOwner{
        dividendTracker.setMinBalanceForDividends(amount);
    }

    function _setAutomatedMarketMakerPair(address newPair, bool value) private {
        require(automatedMarketMakerPairs[newPair] != value, "FLOKIVADER: Automated
market maker pair is already set to that value");
        automatedMarketMakerPairs[newPair] = value;

        if(value) {
            dividendTracker.excludeFromDividends(newPair, true);
        }

        emit SetAutomatedMarketMakerPair(newPair, value);
    }

    /// @notice Update the gasForProcessing needed to auto-distribute rewards
```

```
    /// @param newValue The new amount of gas needed
    /// @dev The amount should not be greater than 500k to avoid expensive transactions
    function setGasForProcessing(uint256 newValue) external onlyOwner {
        require(newValue >= 200000 && newValue <= 500000, "FLOKIVADER: gasForProcessing must be between 200,000 and 500,000");
        require(newValue != gasForProcessing, "FLOKIVADER: Cannot update gasForProcessing to same value");
        emit GasForProcessingUpdated(newValue, gasForProcessing);
        gasForProcessing = newValue;
    }

    /// @dev Update the dividendTracker claimWait
    function setClaimWait(uint256 claimWait) external onlyOwner {
        dividendTracker.updateClaimWait(claimWait);
    }

     ////////////////////
    // Getter Functions //
   ////////////////////

    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 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 getLastProcessedIndex() external view returns(uint256) {
        return dividendTracker.getLastProcessedIndex();
    }

    function getNumberOfDividendTokenHolders() external view returns(uint256) {
        return dividendTracker.getNumberOfTokenHolders();
    }

     ////////////////////////
    // Transfer Functions //
    ////////////////////////

    function airdropTokens(address[] memory accounts, uint256[] memory amounts) external
onlyOwner{
        require(accounts.length == amounts.length, "Arrays must have same size");
        for(uint256 i; i< accounts.length; i++){
            super._transfer(msg.sender, accounts[i], amounts[i]);
        }
    }

    function _transfer(address from, address to, uint256 amount) 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;
        }
```

```
                uint256 contractTokenBalance = balanceOf(address(this));

        bool canSwap = contractTokenBalance >= swapTokensAtAmount;


        if( canSwap && !swapping && swapEnabled && !automatedMarketMakerPairs[from] &&
!_isExcludedFromFees[from] && !_isExcludedFromFees[to]) {
            swapping = true;


            bool isSell;
            if(automatedMarketMakerPairs[to]){ isSell = true;}


            if(isSell && totalSellTax > 0) swapAndLiquify(swapTokensAtAmount, true);
            else if(!isSell && totalTransferTax > 0) swapAndLiquify(swapTokensAtAmount, false);


            swapping = false;
        }


        bool takeFee = !swapping;


        // if any account belongs to _isExcludedFromFee account then remove the fee
        if(_isExcludedFromFees[from] || _isExcludedFromFees[to]) {
            takeFee = false;
        }


        if(takeFee) {
            uint256 feeAmt;


            if(automatedMarketMakerPairs[to]) feeAmt = amount * totalSellTax / 100;
            else if(automatedMarketMakerPairs[from]) feeAmt = amount * totalBuyTax / 100;
            else feeAmt = amount * totalTransferTax / 100;
```

```
            amount = amount - feeAmt;
            super._transfer(from, address(this), feeAmt);
        }
        super._transfer(from, to, amount);

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

        if(!swapping) {
                        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 tokens, bool isSell) private {
        uint256 initialBalance = address(this).balance;
        swapTokensForBNB(tokens);
        uint256 deltaBalance = address(this).balance - initialBalance;

        Taxes memory temp;
        uint256 totalTax;
        if(isSell) { temp = sellTaxes; totalTax = totalSellTax; }
        else  { temp = transferTaxes; totalTax = totalTransferTax; }

        // Send BNB to marketingWallet
        uint256 marketingWalletAmt = deltaBalance * temp.marketing / totalTax;
```

```
    if(marketingWalletAmt > 0){
        payable(marketingWallet).sendValue(marketingWalletAmt);
    }

    // Send BNB to charity
    uint256 charityAmt = deltaBalance * temp.charity / totalTax;
    if(charityAmt > 0){
        payable(charityWallet).sendValue(charityAmt);
    }

    // Send BNB to autoboost
    uint256 autoBoostAmt = deltaBalance * temp.autoBoost / totalTax;
    if(autoBoostAmt > 0){
        payable(autoBoostWallet).sendValue(autoBoostAmt);
    }

    // Send BNB to rewards
    uint256 dividends = deltaBalance * temp.rewards / totalTax;
    if(dividends > 0){
        (bool success,) = address(dividendTracker).call{value: dividends}("");
        if(success)emit SendDividends(tokens, dividends);
    }


}

function swapTokensForBNB(uint256 tokenAmount) private {
    address[] memory path = new address[](2);
    path[0] = address(this);
    path[1] = router.WETH();
```

```
        _approve(address(this), address(router), tokenAmount);

        // make the swap
        router.swapExactTokensForETHSupportingFeeOnTransferTokens(
            tokenAmount,
            0, // accept any amount of ETH
            path,
            address(this),
            block.timestamp
        );

    }

    function addLiquidity(uint256 tokenAmount, uint256 ethAmount) private {

        // approve token transfer to cover all possible scenarios
        _approve(address(this), address(router), tokenAmount);

        // add the liquidity
        router.addLiquidityETH{value: ethAmount}(
            address(this),
            tokenAmount,
            0, // slippage is unavoidable
            0, // slippage is unavoidable
            marketingWallet,
            block.timestamp
        );

    }

}
```

```
contract FLOKIVADERDividendTracker is Ownable, DividendPayingToken {
    using SafeMath for uint256;
    using SafeMathInt 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 minimumTokenBalanceForDividends;

    event ExcludeFromDividends(address indexed account, bool value);
    event ClaimWaitUpdated(uint256 indexed newValue, uint256 indexed oldValue);

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

    constructor()  DividendPayingToken("FLOKIVADER_Dividen_Tracker",
"FLOKIVADER_Dividend_Tracker") {
            claimWait = 1 days;
            minimumTokenBalanceForDividends = 200_000_000_000 * (10**9);
    }

    function _transfer(address, address, uint256) internal pure override {
        require(false, "FLOKIVADER_Dividend_Tracker: No transfers allowed");
    }
```

```
    function setMinBalanceForDividends(uint256 amount) external onlyOwner{
        minimumTokenBalanceForDividends = amount * 10**9;
    }

    function excludeFromDividends(address account, bool value) external onlyOwner {
            require(excludedFromDividends[account] != value);
            excludedFromDividends[account] = value;
      if(value == true){
        _setBalance(account, 0);
        tokenHoldersMap.remove(account);
      }
      else{
        _setBalance(account, balanceOf(account));
        tokenHoldersMap.set(account, balanceOf(account));
      }
      emit ExcludeFromDividends(account, value);

    }

    function updateClaimWait(uint256 newClaimWait) external onlyOwner {
        require(newClaimWait >= 3600 && newClaimWait <= 86400,
"FLOKIVADER_Dividend_Tracker: claimWait must be updated to between 1 and 24 hours");
        require(newClaimWait != claimWait, "FLOKIVADER_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 + (int256(processesUntilEndOfArray));
        }
    }


    withdrawableDividends = withdrawableDividendOf(account);
    totalDividends = accumulativeDividendOf(account);

    lastClaimTime = lastClaimTimes[account];

    nextClaimTime = lastClaimTime > 0 ?
                        lastClaimTime + (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 account, uint256 newBalance) public onlyOwner {
        if(excludedFromDividends[account]) {
                return;
        }

        if(newBalance >= minimumTokenBalanceForDividends) {
      _setBalance(account, newBalance);
                tokenHoldersMap.set(account, newBalance);
        }

        else {
      _setBalance(account, 0);
                tokenHoldersMap.remove(account);
        }
```

```
        processAccount(payable(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 + (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;
    }
```

}