**Trading Deployed Smart Contracts**

**24th January 2023**

|  |  |
| --- | --- |
| **Edited By** | Haris Javed |
| **Date** | 24/1/2023 |
| **Document Version** | V0.1 |
| **Template Version** | **V01.00** |

**Table of Contents**

[1 EXECUTIVE SUMMARY 3](file:///C:\Users\Admin\Downloads\TradingWithCrypto.docx#_Toc125463254)

[1.1 Trading Contract 3](file:///C:\Users\Admin\Downloads\TradingWithCrypto.docx#_Toc125463255)

[1.1.1 Trading Contract Functionality 3](file:///C:\Users\Admin\Downloads\TradingWithCrypto.docx#_Toc125463256)

[1.1.2 Function of Trading Contract 4](file:///C:\Users\Admin\Downloads\TradingWithCrypto.docx#_Toc125463257)

# EXECUTIVE SUMMARY

This is a smart contract designed to facilitate trading of assets. It allows the creation and management of trades, and tracking the total number of trades. The contract also includes access control, granting different roles to users such as sellers and buyers. It also includes error messages that can be thrown in case of invalid input or other issues, ensuring the security and control of the trading process. Overall, this contract aims to provide a secure and efficient way to trade assets on the blockchain.

### Trading Contract

Trading contract are used for trade creation, management and also tracking trades.

Before deployment, we must provide following details

* URI – BL Hash base Uri

constructor (string memory uri) {

baseUri=uri;

\_grantRole(DEFAULT\_ADMIN\_ROLE, msg.sender);

\_grantRole(SELLER\_ROLE, msg.sender);

\_grantRole(TRADER\_ROLE, msg.sender);

\_grantRole(BUYER\_ROLE, msg.sender);

}

#### Trading Contract Functionality

* By default, Deployer has an ADMIN and SELLER, TRADER and BUYER role.
* Only ADMIN can **grantRole ()** and **revokeRole ().**
* Any wallet account that holds SELLER, TRADER or BUYER role can **createtrade (), agreeToTrade () , updateLC () , verifyLC (),updateBL (), verifyBL ()** only their tradesand see all trades history using **batchDetailsTrades ()** but any one can see hash of LC and BL using **getAllHash ()**

#### 1.1.2 Function of Trading Contract

1. **grantRole function:**

In grant Role admin can assign roles to new users like SELLER, BUYER, TRADER

function grantRole(bytes32 role, address account) public

virtual override onlyRole(getRoleAdmin(role)) {\_

grantRole(role, account);}

1. **revokeRole function:**

In grant Role admin can revoke the assign roles to users like SELLER, BUYER, TRADER

function revokeRole(bytes32 role, address account) public virtual override onlyRole(getRoleAdmin(role)) {

\_revokeRole(role, account);

}

1. **createTrade function:**

creating an order on a trading platform. The function takes in several parameters, including a trade number, an asset name, a wallet address, a total amount, and a trade type.

The function first checks if the trade number provided has already been used. If it has, the function will revert with an **"InvalidTradeNumber"** error.

Next, the function checks the trade type provided. If the trade type is equal to 1, it will check if the sender of the transaction has the role of "Seller" and if the provided wallet address has the role of "Trader". If either of these checks fails, the function will revert with an **"InvalidSeller"** or **"InvalidTrader"** error. If the trade type is equal to 2, it will check if the sender has the role of "Trader" and if the provided wallet address has the role of "Buyer". If either of these checks fails, the function will revert with an **"InvalidTrader"** or **"InvalidBuyer"** error. If the trade type is equal to 3, it will check if the sender has the role of "Buyer" and if the provided wallet address has the role of "Trader". If either of these checks fails, the function will revert with an **"InvalidBuyer"** or **"InvalidTrader"** error. If none of the above conditions are met, the function will revert with an **"InvalidTradeType"** error.

Assuming all the above checks pass, the function will then call another function **"\_createOrder"** which will create the order with the provided parameters. It also keeps record of the trade number in **"totalTradingNumbers"**

 function createOrder(

        uint256 tradeNumber,

        string memory assetName,

        address walletAddress,

        uint256 totalAmount,

        uint256 tradeType

    )external

    {

        address fromAddress;

        address toAddress;

    if(\_trading[tradeNumber].peelTradeNumber()){revert InvalidTradeNumber();}

     if(tradeType==1){ //Seller To Trader

        if(!hasRole(SELLER\_ROLE,msg.sender)){revert InvalidSeller();}

        if(!hasRole(TRADER\_ROLE,walletAddress)){revert InvalidTrader();}

        fromAddress = msg.sender;

        toAddress = walletAddress;

      }else if(tradeType==2){ //Trader To Buyer

        if(!hasRole(TRADER\_ROLE,msg.sender)){revert InvalidTrader();}

        if(!hasRole(BUYER\_ROLE,walletAddress)){revert InvalidBuyer();}

        fromAddress = msg.sender;

        toAddress = walletAddress;

      }else if(tradeType==3){ //Buyer To Trader

        if(!hasRole(BUYER\_ROLE,msg.sender)){revert InvalidBuyer();}

        if(!hasRole(TRADER\_ROLE,walletAddress)){revert InvalidTrader();}

        fromAddress = walletAddress;

        toAddress = msg.sender;

      }else{

          revert InvalidTradeType();

      }

     \_createOrder(

            tradeNumber,

            assetName,

            fromAddress,

            toAddress,

            totalAmount,

            tradeType

     );

     totalTradingNumbers.add(tradeNumber);

    }

1. **agreeToTrade function:**

The function first checks if the trade number provided is valid by calling the **"peelTradeNumber"** function on the stored trade data. If the trade number is invalid, the function will revert with an **"InvalidTradeNumber"** error.

Next, the function checks if the trade has already been agreed to by calling the **"isAcceptTrading"** function on the stored trade data. If the trade has already been agreed to, the function will revert with an **"AllReadyAgreed"** error.

Then the function checks the trade type, if the trade type is 1 or 2, it will check if the msg.sender is the **'toAddress'** of the trade, if the trade type is 3 it will check if the msg.sender is the **'fromAddress'** of the trade. If the check fails, the function will revert with an **"InvalidCallerAddress"** error. If the trade type is other than 1,2 or 3, it will revert with an **"InvalidTradeType"** error.

Assuming all the above checks pass, the function will then call a function **"setAcceptTrading"** which will mark the trade as agreed.

function agreeToTrade(uint256 tradeNumber)public {

        if(!\_trading[tradeNumber].peelTradeNumber()){revert InvalidTradeNumber();}

        if(\_trading[tradeNumber].isAcceptTrading()){revert AllReadyAgreed();}

        if(\_trading[tradeNumber].tradingType()==1 || \_trading[tradeNumber].tradingType()==2)

        {

            if(\_trading[tradeNumber].tradingToAddress()!=msg.sender){revert InvalidCallerAddress();}

        }else if(\_trading[tradeNumber].tradingType()==3)

        {

            if(\_trading[tradeNumber].tradingFromAddress()!=msg.sender){revert InvalidCallerAddress();}

        }else{

         revert InvalidTradeType();

        }

      \_trading[tradeNumber].setAcceptTrading();

    }

1. **updateLC function:**

The function takes in two parameters: a trade number and a hash.Then function first checks if the trade number provided is valid by calling the **"peelTradeNumber"** function on the stored trade data. If the trade number is invalid, the function will revert with an **"InvalidTradeNumber**" error.

Next, the function checks if the LC hash field is already filled by calling the **"isTradingLCHashEmpty"** function on the stored trade data. If the LC hash field is already filled, the function will revert with an **"AllreadyAdd"** error.

Then the function checks if the trade has been accepted by calling the **"isAcceptTrading"** function on the stored trade data. If the trade has not been accepted, the function will revert with an **"NotAcceptedYet"** error.

Then the function checks if the msg.sender is the **'toAddress'** of the trade. If the check fails, the function will revert with an **"InvalidCallerAddress"** error.

Assuming all the above checks pass, the function will then call a function **"setTradingLC"** which will update the trade with the provided hash as the LC information.

 function updateLC(uint256 tradeNumber,string memory hash)public{

      if(!\_trading[tradeNumber].peelTradeNumber()){revert InvalidTradeNumber();}

      if(!\_trading[tradeNumber].isTradingLCHashEmpty()){revert AllreadyAdd();}

      if(!\_trading[tradeNumber].isAcceptTrading()){revert NotAcceptedYet();}

      if(\_trading[tradeNumber].tradingToAddress()!=msg.sender){revert InvalidCallerAddress();}

      \_trading[tradeNumber].setTradingLC(hash);

    }

1. **verifyLC function:**

This function is verifying the Letter of Credit (LC) information on a trade. The function takes in two parameters: a trade number and a hash.

The function first checks if the trade number provided is valid by calling the **"peelTradeNumber"** function on the stored trade data. If the trade number is invalid, the function will revert with an **"InvalidTradeNumber"** error.

Next, the function checks if the LC verify hash field is already filled by calling the **"isTradingVerifyLCHashEmpty"** function on the stored trade data. If the LC verify hash field is already filled, the function will revert with an **"AllreadyAdd"** error.

Then the function checks if the LC hash field is already filled by calling the **"isTradingLCHashEmpty"** function on the stored trade data. If the LC hash field is not filled, the function will revert with an **"LcNotAddYet"** error.

Then the function checks if the msg.sender is the **'fromAddress'** of the trade. If the check fails, the function will revert with an **"InvalidCallerAddress"** error.

Assuming all the above checks pass, the function will then call a function **"setVerifyTradingLC"** which will update the trade with the provided hash as the verified LC information.

 function verifyLC(uint256 tradeNumber,string memory hash)public{

      if(!\_trading[tradeNumber].peelTradeNumber()){revert InvalidTradeNumber();}

      if(!\_trading[tradeNumber].isTradingVerifyLCHashEmpty()){revert AllreadyAdd();}

      if(\_trading[tradeNumber].isTradingLCHashEmpty()){revert LcNotAddYet();}

      if(\_trading[tradeNumber].tradingFromAddress()!=msg.sender){revert InvalidCallerAddress();}

      \_trading[tradeNumber].setVerifyTradingLC(hash);

    }

1. **updateBL function**

This function is updating the Bill of Lading (BL) information on a trade. The function takes in two parameters: a trade number and a hash.The function first checks if the trade number provided is valid by calling the "peelTradeNumber" function on the stored trade data. If the trade number is invalid, the function will revert with an "InvalidTradeNumber" error.

Next, the function checks if the BL hash field is already filled by calling the "isTradingBlHashEmpty" function on the stored trade data. If the BL hash field is already filled, the function will revert with an "AllreadyAdd" error.Then the function checks if the LC verify hash field is already filled by calling the "isTradingVerifyLCHashEmpty" function on the stored trade data. If the LC verify hash field is not filled, the function will revert with an "LCNotVerifyYet" error.

Then the function checks if the msg.sender is the 'fromAddress' of the trade. If the check fails, the function will revert with an "InvalidCallerAddress" error.

Assuming all the above checks pass, the function will then call a function "setTradingBL" which will update the trade with the provided hash as the BL information.

function updateBL(uint256 tradeNumber,string memory hash)public{

      if(!\_trading[tradeNumber].peelTradeNumber()){revert InvalidTradeNumber();}

      if(!\_trading[tradeNumber].isTradingBlHashEmpty()){revert AllreadyAdd();}

      if(\_trading[tradeNumber].isTradingVerifyLCHashEmpty()){revert LCNotVerifyYet();}

      if(\_trading[tradeNumber].tradingFromAddress()!=msg.sender){revert InvalidCallerAddress();}

      \_trading[tradeNumber].setTradingBL(hash);

    }

1. **verifyBL function**

This functionis used for verifying the Bill of Lading (BL) information on a trade. The function takes in two parameters: a trade number and a hash.The function first checks if the trade number provided is valid by calling the "peelTradeNumber" function on the stored trade data. If the trade number is invalid, the function will revert with an "InvalidTradeNumber" error.

Next, the function checks if the BL verify hash field is already filled by calling the "isTradingVerifyBLHashEmpty" function on the stored trade data. If the BL verify hash field is already filled, the function will revert with an "AllreadyAdd" error.Then the function checks if the BL hash field is already filled by calling the "isTradingBlHashEmpty" function on the stored trade data. If the BL hash field is not filled, the function will revert with an "NotBLAddedYet" error.Then the function checks if the msg.sender is the 'toAddress' of the trade. If the check fails, the function will revert with an "InvalidCallerAddress" error.

Assuming all the above checks pass, the function will then call a function "setVerifyTradingBL" which will update the trade with the provided hash as the verified BL information.

function verifyBL(uint256 tradeNumber,string memory hash)public{

      if(!\_trading[tradeNumber].peelTradeNumber()){revert InvalidTradeNumber();}

      if(!\_trading[tradeNumber].isTradingVerifyBLHashEmpty()){revert AllreadyAdd();}

      if(\_trading[tradeNumber].isTradingBlHashEmpty()){revert NotBLAddedYet();}

      if(\_trading[tradeNumber].tradingToAddress()!=msg.sender){revert InvalidCallerAddress();}

      \_trading[tradeNumber].setVerifyTradingBL(hash);

    }

1. **getAllHash:**

This function is used for getting all the hash values (LC, verified LC, BL, and verified BL) associated with a trade. The function takes in one parameter: a trade number.The function first checks if the trade number provided is valid by calling the **"peelTradeNumber"** function on the stored trade data. If the trade number is invalid, the function will revert with an **"InvalidTradeNumber**" error.

Then it will get the baseURI and all the hashes of LC, verified LC, BL and verified BL of the trade by calling the respective functions on the stored trade data.It will then concatenate the baseURI and the hash values and return them as 4 different strings. If the baseURI is not set, it will return an empty string.

It means this function is getting all the hash values of LC, verified LC, BL and verified BL of a trade and concatenating with the baseURI and returning as 4 different strings.

 function getAllHash(uint256 tradeNumber) public virtual view returns(string memory,string memory,string memory,string memory) {

        if(!\_trading[tradeNumber].peelTradeNumber()){revert InvalidTradeNumber();}

        string memory baseURI = \_baseURI();

        string memory hash = \_trading[tradeNumber].tradingIssueLC();

        string memory hash1 = \_trading[tradeNumber].tradingVerifyLC();

        string memory hash2=\_trading[tradeNumber].tradingBlHash();

        string memory hash3 = \_trading[tradeNumber].tradingVerifyBL();

        return (

            bytes(baseURI).length > 0

            ? string(abi.encodePacked(baseURI,hash))

            : "",

            bytes(baseURI).length > 0

            ? string(abi.encodePacked(baseURI,hash1))

            : "",

            bytes(baseURI).length > 0

            ? string(abi.encodePacked(baseURI,hash2))

            : "",

             bytes(baseURI).length > 0

            ? string(abi.encodePacked(baseURI,hash3))

            : ""

         );

    }

1. **batchDetailsTrades function**

This function is used for fetching the details of multiple trades at once. The function takes in one parameter: an array of trade numbers.The function first declares an array of "**titrasStorage.trading"** struct, which presumably contains all the relevant details of a trade.

Then the function iterates over the array of trade numbers provided, and for each trade number, it retrieves the corresponding trade data from the storage variable "**\_trading**" and assigns it to the corresponding index in the "**detailsTrades**" array.

Finally, the function returns the **"detailsTrades"** array, which contains the details of all the trades specified by the trade numbers in the input array. This function is useful when a user want to fetch details of multiple trades at once instead of fetching them individually.

 function batchDetailsTrades(uint256[] memory tradingNumbers) public view returns(titrasStorage.trading[] memory) {

        titrasStorage.trading[] memory detailsTrades = new titrasStorage.trading[](tradingNumbers.length);

        for (uint256 index = 0; index < tradingNumbers.length; ++index) {

            detailsTrades[index] = \_trading[tradingNumbers[index]];

        }

        return detailsTrades;

    }