Name : Syed Muhammad Abdullah

Problem no 1

- Develop a smart contract on the Ethereum blockchain platform to create a new token called DOLA, which mirrors DAI.
- DOLA's value will be pegged to our upcoming token, ROI, with BDOLA as collateral.
- Ensure the smart contract maintains stability and accurately pegs its value to the specified currency.

Description:

The **DolaToken** contract is designed to create a stable token, DOLA, which is pegged to an upcoming token, ROI, while utilizing BDOLA as collateral. This contract allows users to mint and burn DOLA tokens, with minting being secured by the provision of BDOLA tokens.

Key Components of the Contract

1. **Token Metadata**:
    - **Name**: `"DOLA Token"` - This is the display name of the token.
    - **Symbol**: `"DOLA"` - This is the abbreviated name used to represent the token on exchanges and wallets.
    - **Decimals**: `18` - Specifies the decimal precision for the token, allowing for smaller units to be represented (similar to Ether).
2. **Token References**:
    - **`IERC20 public roiToken`**: This variable stores a reference to the ROI token, which will serve as the underlying asset to which DOLA is pegged.
    - **`IERC20 public bdolaToken`**: This variable references the BDOLA token, which serves as collateral when minting DOLA.
3. **State Variables**:
    - **`mapping(address => uint256) public balances`**: This mapping tracks the DOLA balance for each user, allowing users to check their holdings.
    - **`uint256 public totalSupply`**: This variable keeps track of the total supply of DOLA tokens currently in circulation.
4. **Events**:
    - **`event Mint(address indexed user, uint256 amount)`**: This event is emitted whenever a user mints DOLA tokens, recording the user's address and the amount minted.
    - **`event Burn(address indexed user, uint256 amount)`**: This event is emitted when a user burns DOLA tokens, capturing the user's address and the amount burned.
5. **Constructor**:

- The constructor initializes the contract by setting the addresses for the ROI and BDOLA tokens.

```
solidity
Copy code
constructor(address _roiToken, address _bdolaToken) {
    roiToken = IERC20(_roiToken);     // Set the ROI token address
    bdolaToken = IERC20(_bdolaToken); // Set the BDOLA token address
}
```

Core Functions

1. **Minting DOLA**:

```
solidity
Copy code
function mintDola(uint256 amount) external {
    require(amount > 0, "Amount must be greater than zero");
    require(bdolaToken.transferFrom(msg.sender, address(this), amount),
"Transfer of BDOLA failed");
    balances[msg.sender] += amount;
    totalSupply += amount;
    emit Mint(msg.sender, amount);
}
```

   - **Purpose**: Allows users to mint DOLA tokens by providing BDOLA as collateral.
   - **Logic**:
     - The function checks that the provided amount is greater than zero.
     - It transfers BDOLA from the user to the contract as collateral.
     - The user's DOLA balance is updated, and the total supply of DOLA is incremented.
     - Finally, the Mint event is emitted to log the action.
2. **Burning DOLA**:

```
solidity
Copy code
function burnDola(uint256 amount) external {
    require(amount > 0, "Amount must be greater than zero");
    require(balances[msg.sender] >= amount, "Insufficient DOLA balance");
    balances[msg.sender] -= amount;
    totalSupply -= amount;
    require(bdolaToken.transfer(msg.sender, amount), "Transfer of BDOLA
failed");
    emit Burn(msg.sender, amount);
}
```

   - **Purpose**: Allows users to burn DOLA tokens in exchange for their equivalent value in BDOLA.
   - **Logic**:
     - The function ensures that the amount to be burned is greater than zero.
     - It checks that the user has enough DOLA to burn.
     - The user's balance and the total supply of DOLA are updated accordingly.
     - The equivalent amount of BDOLA is transferred back to the user.
     - The Burn event is emitted to log the action.
3. **Balance Query**:

```
solidity
Copy code
function balanceOf(address account) external view returns (uint256) {
    return balances[account];
}
```

- **Purpose**: Returns the DOLA balance of a specified address.
- **Usage**: This function allows users and other contracts to check the balance of DOLA tokens held by any address.

4. **Total Supply Query**:

```
solidity
Copy code
function getTotalSupply() external view returns (uint256) {
    return totalSupply;
}
```

- **Purpose**: Returns the total supply of DOLA tokens in circulation.
- **Usage**: Useful for understanding the overall issuance of DOLA and its market status.

Use Case and Importance

The DolaToken contract serves a crucial role in the DeFi ecosystem by providing a stablecoin-like asset (DOLA) that is pegged to another token (ROI) and backed by collateral (BDOLA). This structure offers users a way to gain liquidity and stability while allowing them to retain the ability to recover their collateral at any time through the burn mechanism.

Conclusion

The DolaToken contract is a well-structured ERC20 token implementation that integrates collateralization features, ensuring that DOLA maintains its value by being pegged to the ROI token. This contract exemplifies how DeFi projects can leverage smart contracts to create stable, user-friendly financial products in a decentralized manner.

Problem no 2

- Develop a smart contract on the Ethereum blockchain platform to create a new token called DOLA, which mirrors DAI.
- DOLA's value will be pegged to our upcoming token, ROI, with BDOLA as collateral.
- Ensure the smart contract maintains stability and accurately pegs its value to the specified
currency.

Description:

The **Wrapped Empress Token (WEMP)** contract is designed to facilitate the wrapping and unwrapping of the native **Empress Token (EMP)** on the Ethereum blockchain. This contract allows users to convert EMP into WEMP, creating a

wrapped version of the token that can be utilized within various decentralized finance (DeFi) applications, while also enabling users to revert back to the original token as needed.

Key Features

1. **Token Metadata**:
    - **Name**: Wrapped Empress Token
    - **Symbol**: WEMP
    - These attributes define the token and make it easily identifiable on the blockchain.
2. **Core Components**:
    - **ERC20 Compliance**: The contract adheres to the ERC20 standard for token interactions, ensuring compatibility with other decentralized applications and wallets.
3. **State Variables**:
    - `IERC20 public empressToken`: A reference to the underlying Empress Token contract, allowing interactions with EMP.
    - `mapping(address => uint256) public wrappedBalances`: A mapping that keeps track of how much WEMP each user holds, enabling users to see their balance and manage their tokens.
    - `uint256 public totalWrappedSupply`: A variable that tracks the total supply of WEMP currently in circulation, which helps in understanding the overall distribution of the wrapped token.
4. **Events**:
    - `event Wrapped(address indexed user, uint256 amount)`: This event is emitted whenever a user successfully wraps EMP into WEMP. It records the user's address and the amount wrapped.
    - `event Unwrapped(address indexed user, uint256 amount)`: This event is emitted when a user successfully unwraps WEMP back into EMP, capturing the user's address and the amount unwrapped. These events improve transparency and allow for better tracking of token movements.
5. **Functions**:
    - **Constructor**:

      ```
      solidity
      Copy code
      constructor(address _empressTokenAddress) {
          empressToken = IERC20(_empressTokenAddress);
      }
      ```

      Initializes the contract with the address of the EMP token, establishing a connection for future transactions.

    - `wrap(uint256 amount)`:
        - **Purpose**: Converts a specified amount of EMP into WEMP.
        - **Logic**:
            1. Checks that the amount to be wrapped is greater than zero.

2. Verifies that the user has sufficient EMP balance.
3. Transfers EMP from the user's wallet to the contract.
4. Updates the user's WEMP balance and the total supply.
5. Emits the `Wrapped` event.

- `unwrap(uint256 amount)`:
  - **Purpose**: Converts a specified amount of WEMP back into EMP.
  - **Logic**:
    1. Checks that the amount to be unwrapped is greater than zero.
    2. Verifies that the user has enough WEMP.
    3. Updates the user's WEMP balance and total supply accordingly.
    4. Transfers the equivalent amount of EMP back to the user.
    5. Emits the `Unwrapped` event.
- `balanceOf(address account)`:
  - **Purpose**: Returns the WEMP balance of a specified address.
  - **Usage**: Allows users and applications to query the WEMP balance of any address.
- `totalSupply()`:
  - **Purpose**: Returns the total supply of WEMP tokens currently issued.
  - **Usage**: Provides insights into the overall circulation of the wrapped token.

Use Cases

- **Decentralized Finance (DeFi)**: Users can wrap EMP tokens to use them in DeFi applications, providing liquidity, earning rewards, or participating in governance while retaining the ability to convert back to the original token.
- **Tokenized Assets**: The WEMP token can serve as a representation of EMP in various ecosystems, enhancing the token's utility across different platforms.

Conclusion

The Wrapped Empress Token (WEMP) contract provides a seamless way to manage the conversion between EMP and its wrapped version, enhancing its usability within the growing DeFi ecosystem. By enabling wrapping and unwrapping functionalities, the contract increases the flexibility and accessibility of the Empress Token for users and developers alike.