BalancerPool Contract

Contract address:
0x110B1EE769ba9D0a1A2FAeCe4be3924D1F30Ed8C

The BalancerPool contract is a decentralized pool management system designed to interact with Tether (USDT) and a Main Pool (PriceBalancer) contract. It uses robust role-based access controls to ensure secure and efficient allocation, tracking, and withdrawal of user token balances. Built with OpenZeppelin's AccessControl library, this contract ensures a modular and secure design while providing transparency for all stakeholders.
The Admins can allocate tokens to users, ensuring they exist in the main pool. Users can withdraw their tokens, which transfers a portion to the main pool and mint the users new APX tokens.

Libraries and Interfaces

Role-Based Access Control:
Utilizes OpenZeppelin's AccessControl for defining roles.
- DEFAULT_ADMIN_ROLE: Grants administrative privileges, such as assigning roles.
- OPERATOR_ROLE: Allows operators to add users, manage balances.

Only addresses with the OPERATOR_ROLE can add user balances and update the main pool address.Withdrawal operations are limited to users with sufficient balances.

IERC20_USDT Interface:
Defines key ERC20 functions (transferFrom, transfer, balanceOf) for interaction with the Tether (USDT) token contract.

Key Features

User Balance Management:
- Supports adding token balances for users verified through the Main Pool contract.
- Enables secure withdrawals with automatic allocation to related pools, ensuring a consistent fund distribution mechanism.

Integration with Main Pool:
- Interfaces with the Main Pool to verify user history and handle fund withdrawals effectively.
- Retrieving addresses of related pools (Owner, Matching Bonus, Balancer).
- Delegating fund withdrawals to the Main Pool.

Transparent Events:
- Logs significant actions such as user additions, withdrawals, balance updates, and pool address changes, enhancing traceability.

Contract Functions

Readable functions

Function	Description
DEFAULT_ADMIN_ROLE	A constant that defines the default administrative role in the smart contract. Used to manage access control.
OPERATOR_ROLE	A constant that defines the role for operators who may have specific permissions within the contract.
checkBalance(_user address)	Returns the current balance of a _user in this pool.
getMainPoolAddress()	Returns the PriceBalancer address.
getRoleAdmin(role bytes32)	A function to retrieve the administrator responsible for managing a specific role within the contract.
hasRole(role bytes32, account address)	A function that checks whether a specific account holds a particular role in the contract.
supportsInterface(interfaceId bytes4)	Checks if the contract implements a specific interface (useful for compatibility and standards like ERC165).
tetherToken()	Returns the address of a linked Tether (USDT) token.

Executable Functions

Function	Description
addUser(_user address, _amount uint256)	Increase the balance of a verified user. This function is accessible by the OPERATOR_ROLE only. The user's address must be valid and already registered in the Main Pool. The added balance must be greater than zero.
changeMainPool(newAddress address)	Updates the address of the Main Pool (Price Balancer) contract and is restricted to the OPERATOR_ROLE only.
grantRole(role bytes32, account address)	Assigns a specified role to an account, giving it specific permissions or access within the smart contract.
renounceRole(role bytes32, callerConfirmation address)	Allows an account to voluntarily relinquish a role it holds, removing its associated permissions.
revokeRole(role bytes32, account address)	Used by an administrator to remove a specific role from an account, effectively revoking its permissions.
withdrawal(_amount uint256)	Allows users to withdraw their allocated token balance. Users must have sufficient balance and the minimum withdrawal amount is 2 USDT units. One USDT units are transferred to the Owner Pool and the remaining balance is transferred to the Main Pool and mint the user new APX tokens. It also updates the APX token price.

Codes

Function	**Code**
DEFAULT_ADMIN_ROLE	`Public variable`
OPERATOR_ROLE	`Public variable`

checkBalance(_user address)	`function checkBalance(address _user) external view returns (uint256) {`
	`        return userBalancerBalances[_user];`
	`    }`

getMainPoolAddress()	`function getMainPoolAddress() external view returns (address) {`
	`        return mainPoolAddress;`
	`    }`

getRoleAdmin(role bytes32)	`function getRoleAdmin(bytes32 role) public view virtual returns (bytes32) {`
	`        return _roles[role].adminRole;`
	`    }`

hasRole(role bytes32, account address)	`function hasRole(bytes32 role, address account) public view virtual returns (bool) {`
	`        return _roles[role].hasRole[account];`
	`    }`

supportsInterface(interfaceId bytes4)	`function supportsInterface(bytes4 interfaceId) public view virtual returns (bool) {`
	`        return interfaceId == type(IERC165).interfaceId;`
	`    }`

tetherToken()	`Public variable`

addUser(_user address, _amount uint256)	`function addUser(address _user, uint256 _amount) external onlyRole(OPERATOR_ROLE){`
	`        require(_user != address(0), "Invalid address");`
	`        require(_amount > 0, "Amount must be greater than zero");`
	`        IMAIN_POOL.UserData memory userHistory = IMAIN_POOL(mainPoolAddress).getUserHistory(_user);`
	`        require(userHistory.totalPurchase != 0, "You are not user");`
	
	`        userBalancerBalances[_user] += _amount;`
	`        emit NewUserAdded(_user, _amount);`
	`    }`

changeMainPool(newAddress address)	`function changeMainPool(address newAddress) external onlyRole(OPERATOR_ROLE) {`
	`        mainPoolAddress = newAddress;`
	`        emit MainPoolUpdated(newAddress);`
	`    }`

grantRole(role bytes32, account address)	`function grantRole(bytes32 role, address account) public virtual onlyRole(getRoleAdmin(role)) {`
	`        _grantRole(role, account);`
	`    }`

Function	**Code**

renounceRole(role bytes32, callerConfirmation address)	`function renounceRole(bytes32 role, address callerConfirmation) public virtual {`
	`if (callerConfirmation != _msgSender()) {`
	`revert AccessControlBadConfirmation();`
	`}`
	
	`_revokeRole(role, callerConfirmation);`
	`}`

revokeRole(role bytes32, account address)	`function revokeRole(bytes32 role, address account) public virtual onlyRole(getRoleAdmin(role)) {`
	`_revokeRole(role, account);`
	`}`

withdrawal(_amount uint256)	`function withdrawal(uint256 _amount) external {`
	`require(_amount > 2000000, "No balance to withdraw");`
	`uint256 balance = userBalancerBalances[msg.sender];`
	`require(_amount <= balance, "Insufficient balance");`
	
	`userBalancerBalances[msg.sender] -= _amount;`
	`address ownerPool = IMAIN_POOL(mainPoolAddress).getOwnerPoolAddress();`
	`tetherToken.transfer(mainPoolAddress, _amount - 1000000);`
	`tetherToken.transfer(ownerPool, 1000000);`
	
	`require(IMAIN_POOL(mainPoolAddress).poolWithdrawal(msg.sender, _amount - 1000000));`
	`emit poolBalanceUpdated(ownerPool, address(this), 1000000);`
	`emit poolBalanceUpdated(mainPoolAddress, address(this), _amount - 1000000);`
	`emit Withdrawal(address(this), msg.sender, _amount);`
	`}`