MatchingBonusPool

Contract address
0x31DC822976b227273E81cB2cA5f3CE5A8aA1Ab7c

The MatchingBonusPool is a smart contract that manages and distributes matching bonus rewards to users in a decentralized application. It integrates with a main pool contract and uses USDT tokens to handle user rewards and withdrawals. The contract also allows role-based management, ensuring secure and efficient operations.
The MatchingBonusPool allocates its balance to users, and when users withdraw their tokens, new tokens are minted for them, and also updates the token price. If the pool balance is not fully allocated within a specific time frame, any remaining balance will be divided equally among the four reward pools. Admins have the ability to manage the allocation of reward pool addresses, and the system ensures that the pool's balance is always distributed efficiently to users and reward pools.
If the allocated balance is not fully distributed within a set timeframe, the surplus is redistributed equally among four reward pools:

- RewardPool10
- RewardPool20
- RewardPool30
- RewardPool40

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, and update critical contract variables.

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.

IMAIN_POOL Interface:
Represents the price balancer contract and its core functions, enabling seamless integration with the reward system.

- Ensures that the reward system operates in conjunction with the main pool's logic.
- Verifies user eligibility (getUserHistory) before allocating rewards.
- Routes token shares to the appropriate pools during withdrawal operations.

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.
getRewardPoolAddress(_poolNumber uint8)	Returns the address of a specific reward pool.
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.
totalAllocated()	Returns the portion of contract balance that is allocated to the users.

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.
setRewardPoolAddresses(_poolAddresses address[])	Sets all reward pool addresses. Must provide exactly 4 valid addresses.
update()	Distributes unallocated funds equally among the four reward pools.
updateRewardPoolAddress(_poolNumber uint8, _poolAddress address)	Updates a specific reward pool address.
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 userMatchingBonusBalances[_user];
	    }

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

getRewardPoolAddress(_poolNumber uint8)	function getRewardPoolAddress(RewardPools _poolNumber) external view returns (address) {
	        return rewardPools[uint(_poolNumber)];
	    }

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
totalAllocated()	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");
	        userMatchingBonusBalances[_user] += _amount;
	        totalAllocated += _amount;
	        emit userBalanceIncreased(_user, _amount);
	    }

Function	**Code**

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);
	}

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);
	}

setRewardPoolAddresses(_poolAddresses address[])	function setRewardPoolAddresses(address[] calldata _poolAddresses) external onlyRole(OPERATOR_ROLE){
	require(_poolAddresses.length == 4, "Must provide exactly 4 addresses");
	
	for (uint i = 0; i < _poolAddresses.length; i++) {
	require(_poolAddresses[i] != address(0), "Invalid address provided");
	rewardPools[i] = _poolAddresses[i];
	emit RewardPoolAddressUpdated(RewardPools(i), _poolAddresses[i]);
	}
	}

update()	function update() external onlyRole(OPERATOR_ROLE) {
	uint256 remainBalance = tetherToken.balanceOf(address(this)) - totalAllocated;
	require(remainBalance > 0, "Nothing to distribute.");
	uint256 rewardAmount = remainBalance / 4;
	
	for (uint i = 0; i < rewardPools.length; i++) {
	require(rewardPools[i] != address(0), "Invalid address provided");
	
	IERC20_USDT(address(tetherToken)).transfer(rewardPools[i], rewardAmount);
	emit RewardPoolBalanceUpdated(rewardPools[i], rewardAmount);
	}
	}

updateRewardPoolAddress(_poolNumber uint8, _poolAddress address)	function updateRewardPoolAddress(RewardPools _poolNumber, address _poolAddress) external onlyRole(OPERATOR_ROLE){
	require(_poolAddress != address(0), "Invalid address");
	rewardPools[uint(_poolNumber)] = _poolAddress;
	emit RewardPoolAddressUpdated(_poolNumber, _poolAddress);
	}

Function	**Code**

withdrawal(_amount uint256)	function withdrawal(uint256 _amount) external {
	require(_amount > 2000000, "No balance to withdraw");
	uint256 balance = userMatchingBonusBalances[msg.sender];
	require(_amount <= balance, "Insufficient balance");
	
	userMatchingBonusBalances[msg.sender] -= _amount;
	totalAllocated -= _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);
	}