RewardPool20

Contract address
0x58Dd54C2c5C2A06259A3631CA8A7a4C268021980

The RewardPool20 smart contract is designed to manage reward tokens for a decentralized system, offering functionality to allocate, withdraw, and distribute tokens while maintaining strict role-based access control. The contract interacts with external contracts (IERC20_USDT and IMAIN_POOL) for token management and pool-related operations.
This is one of the Matching bonus reward pools. The contract acts similarly to the Matching Bonus Pool, with the key difference being that any surplus in this pool is transferred back to the Matching Bonus Pool rather than being allocated elsewhere. Similar to the Matching bonus pool allocates its balance to users, and when users withdraw their tokens, the USDT will be transferred to the user's wallet and a portion directed to the main pool to adjust the token price in that pool. Like the Matching Bonus Pool, these pools follow a time-sensitive allocation mechanism. If the allocated balance is not fully distributed within the specified timeframe, the remaining balance is transferred back to the Matching Bonus Pool. This creates a continuous cycle of redistribution.

Libraries and Interfaces

Role-Based Access Control:
Utilizes OpenZeppelin's AccessControl for defining roles.

DEFAULT_ADMIN_ROLE:
- Grants full administrative privileges, including assigning or revoking roles.
- Controls the overall configuration of the contract.

OPERATOR_ROLE:
- Allows operators to perform specific actions, such as:
- Adding user reward balances.
- Managing critical contract variables, including updating the main pool address.
- Initiating surplus transfers to designated pools.

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.
getMatchingBonusPoolAddress()	Returns the MatchingBonusPool 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.
totalAllocated()	Returns the portion of contract balance that is allocated to the users.

Executable Functions

Function	Description
addUser(_user address, _amount uint256)	Allocates a specified reward amount to a 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.
transferSurplus ()	Transfers any surplus tokens (not allocated to users) to the matching bonus pool.
withdrawal(_amount uint256)	Allows users to withdraw their reward balances. The withdrawn amount is divided as follows: - Owner Share: 2.5% of the withdrawal amount is transferred to the owner pool. - Token Increase Share: 2.5% of the withdrawal amount is transferred to the main pool. - User Share: The remaining amount is transferred to the user.

Codes

Function	Code
DEFAULT_ADMIN_ROLE	Public variable
OPERATOR_ROLE	Public variable

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

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

getMatchingBonusPoolAddress()	function getMatchingBonusPoolAddress() external view returns (address) {
	return IMAIN_POOL(mainPoolAddress).getMatchingBonusPoolAddress();
	}

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

Function	**Code**

changeMainPool(newAddress address)	function changeMainPool(address newAddress) external onlyRole(OPERATOR_ROLE) {
	require(newAddress != address(0), "Invalid address");
	mainPoolAddress = newAddress;
	emit MainPoolAddressUpdated(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);
	}

transferSurplus ()	function transferSurplus() external onlyRole(OPERATOR_ROLE) {
	uint256 remainBalance = tetherToken.balanceOf(address(this)) - totalAllocated;
	require(remainBalance > 0, "Nothing to distribute.");
	
	address matchingBonusPool = IMAIN_POOL(mainPoolAddress).getMatchingBonusPoolAddress();
	
	IERC20_USDT(address(tetherToken)).transfer(matchingBonusPool, remainBalance);
	emit SurplusTransferred(matchingBonusPool, remainBalance);
	}

Function	**Code**

withdrawal(_amount uint256)	function withdrawal(uint256 _amount) external {
	uint256 balance = userRewardBalances[msg.sender];
	require(balance >= _amount, "No balance to withdraw");
	userRewardBalances[msg.sender] -= _amount;
	totalAllocated -= _amount;
	
	uint256 ownerShare = (_amount * 25)/1000;
	uint256 tokenIncreaseShare = (_amount * 25)/1000;
	uint256 userShare = _amount - ownerShare - tokenIncreaseShare;
	
	// transfer token to owner pool
	address ownerPool = IMAIN_POOL(mainPoolAddress).getOwnerPoolAddress();
	tetherToken.transfer(ownerPool, ownerShare);
	
	// transfer token to main pool
	tetherToken.transfer(mainPoolAddress, tokenIncreaseShare);
	
	require(IMAIN_POOL(mainPoolAddress).updateTokenPriceOnlyPools(tokenIncreaseShare));
	
	// transfer token to user wallet
	tetherToken.transfer(msg.sender, userShare);
	
	emit poolBalanceUpdated(ownerPool, address(this), ownerShare);
	emit poolBalanceUpdated(mainPoolAddress, address(this), tokenIncreaseShare);
	emit Withdrawal(address(this), msg.sender, _amount);
	}