PriceBalancer

Contract address
0xde16646549C2680C14c671e90bf7Df6FA69647e6

The PriceBalancer contract (or Main Pool) is a multi-functional smart contract designed to manage tokenized investment packages with integrated support for the ApexiaToken (APX). It handles package creation, user investment tracking, token minting, and price updates while ensuring a transparent and efficient ecosystem. This contract acts as the backbone of the Apexia system, responsible for managing token operations and pool balances. Its core functionalities include:

- Token Minting and Burning:
  - Tokens are minted when a user purchases a package, with the minted amount determined by the current token price.
  - Tokens are burned when users exchange their APX tokens for USDT, with the equivalent USDT amount transferred to the user's wallet.
- Package Purchase:
  - The contract maintains multiple investment packages with varying prices.
  - Users can purchase these packages with USDT, receive APX tokens, and become eligible to participate in other reward pools.
  - The APX tokens minted during the purchase are directly transferred to the user's wallet, and the balances of the associated pools are updated.
  - After each package purchase, the token price is recalculated, affecting all future buying and selling transactions.
- Selling APX Tokens:
  - Users can sell their APX tokens back to the this pool for USDT.
  - The contract calculates the equivalent USDT amount based on the current token price, burns the APX tokens, and transfers USDT to the user's wallet.
  - After the transaction, the token price is recalculated.

This pool supports three main pools: MatchingBonusPool, OwnersPool, and BalancerPool and manages funds across three sub-pools.

Libraries and Interfaces

Role-Based Access Control:
The contract employs OpenZeppelin's AccessControl to manage permissions for specific operations:

- DEFAULT_ADMIN_ROLE: Grants full control over the contract.
- OPERATOR_ROLE: Enables operational tasks like adding packages and updating data.
- POOL_ROLE: Allows withdrawal and interaction with pool-related functions.

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

IERC20APX Interface:
This interface defines functions specific to the Apexia Token, which implements an extended ERC20 functionality for minting and burning tokens.This enables the contract to mint Apexia tokens during package purchases, and also burns Apexia tokens when users sell their tokens back to the system.

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.
POOL_ROLE	A custom role, likely used to manage specific pool-related actions or permissions within the contract.
addingOldUsers	A flag representing the old-user data is imported to the contract after migration.
apexiaToken()	Returns the address of the ApexiaToken contract.
currentPrice()	Returns the current price of the token which is calculated dynamically.
getAllUsers()	Fetches a list of all user addresses who have participated in investments.
getBalancerPoolAddress()	Returns the address of the Balancer Pool.
getMatchingBonusPoolAddress()	Returns the address of the Matching Bonus Pool.
getOwnerPoolAddress()	Fetches the address of the Owner Pool.
getPackageDetails(uint256 packageId)	Returns the details of a specific package.
getPackagesCount()	Returns the total number of investment packages.
getRoleAdmin(role bytes32)	Returns the admin role responsible for managing a specific role, such as POOL_ROLE.
getUserHistory(address _user)	Fetches the purchase history for a specific user.
hasRole(role bytes32, account address)	Checks if a specific address has a given role (e.g., POOL_ROLE).
initialPrice()	Returns the starting price of the APX token.
poolAddresses(uint256)	Returns the address of specific pool.
supportsInterface(interfaceId bytes4)	Checks if the contract implements a specific interface (e.g., ERC20 or AccessControl).

Function	Description
tetherToken()	Returns the address of the Tether (USDT) token contract used in the system.

Executable Functions

Function	Description
addOldUsers(_userData tuple[], _userAddress address[])	Migrates legacy user data into the contract. Adds a list of pre-existing users to the contract,
addPackage(_name string, _tetherAmount uint256, _ownerPurchaseWage uint256, _tokenPlanWage uint256, _networkPlanWage uint256, _tokenOwnerWage uint256, _tokenIncreaseRate uint256)	Creates a new investment package with specific attributes. _name: name of the package, _tetherAmount: the value of the package (USDT), _ownerPurchaseWage: the owner pool wage, _tokenPlanWage: the percentage of the Token plan, _networkPlanWage: the percentage of Network plan, _tokenOwnerWage: the owner pool wage of the network plan, _tokenIncreaseRate: the percentage for raising the token price.
buy(_packageId uint256, _txId uint256)	Allows users to invest in a package using USDT. This functions: • Distributes funds to predefined pools. • Mints and transfers ApexiaTokens to the user based on the package's token allocation rate. • Updates the token price to reflect the new supply dynamics.
changePackageStatus(_packageId uint256, _status bool)	Modifies the status of a package (e.g., enable, disable).
endOfUsersDataMigration()	Signals the end of a user data migration process, ensuring the system transitions to a stable state.
grantRole(role bytes32, account address)	Assigns a specific role to an account, giving it permissions to perform certain actions.
poolWithdrawal(receiver address, _amount uint256)	Enables withdrawal of funds or tokens from a specific pool within the contract.
renounceRole(role bytes32, callerConfirmation address)	Allows an account to relinquish a role it holds, removing its permissions.
revokeRole(role bytes32, account address)	Used by an admin to remove a role from an account, revoking associated permissions.
sell(uint256 _amount)	Enables users to sell ApexiaTokens for USDT. A small fee is deducted, and the token price is updated to reflect the reduced supply
setPoolAddresses(address[] memory _poolAddresses)	Sets addresses for the three predefined pools.
updatePoolAddress(Pools _poolNumber, address _poolAddress)	Modifies a specific pool's address, ensuring it points to the correct contract.

Function	Description
updateTokenPrice(uint256 _amount)	Updates the token price based on additional investment.
updateTokenPriceOnlyPools(uint256 _amount)	Specifically updates token prices for the reward pools.

Codes

Function	**Code**
DEFAULT_ADMIN_ROLE	`Public variable`
OPERATOR_ROLE	`Public variable`
POOL_ROLE	`Public variable`
addingOldUsers	`Public variable`
apexiaToken()	`Public variable`
currentPrice()	`Public variable`

getAllUsers()	`function getAllUsers() public view returns (address[] memory) {`
	`return (userAddresses);`
	`}`

getBalancerPoolAddress()	`function getBalancerPoolAddress() public view returns(address balancerPool) {`
	`return poolAddresses[uint256(Pools.balancerPool)];`
	`}`

getMatchingBonusPoolAddress()	`function getMatchingBonusPoolAddress() public view returns(address matchingBonusPool) {`
	`return poolAddresses[uint256(Pools.matchingBonusPool)];`
	`}`

getOwnerPoolAddress()	`function getOwnerPoolAddress() public view returns(address ownerPool) {`
	`return poolAddresses[uint256(Pools.ownersPool)];`
	`}`

getPackageDetails(uint256 packageId)	`function getPackageDetails(uint256 packageId) public view returns (Package memory) {`
	`require(packageId < packages.length, "Invalid package");`
	`return packages[packageId];`
	`}`

getPackagesCount()	`function getPackagesCount() public view returns (uint256) {`
	`return packages.length;`
	`}`

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

getUserHistory(address _user)	`function getUserHistory(address _user) public view returns (UserData memory) {`
	`return userPurchaseHistory[_user];`
	`}`

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

Function	**Code**
initialPrice()	`Public variable`
poolAddresses(uint256)	`Public variable`

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

tetherToken()	`Public variable`

addOldUsers(_userData tuple[], _userAddress address[])	`function addOldUsers(UserData[] memory _userData, address[] memory _userAddress) public onlyRole(OPERATOR_ROLE) {`
	`        require(_userData.length ==  _userAddress.length, "Wrong length");`
	`        require(addingOldUsers, "Adding old users is disabled");`
	
	`        for (uint i = 0; i < _userData.length; i++) {`
	`            UserData memory userData = _userData[i];`
	`            if (userPurchaseHistory[_userAddress[i]].totalPurchase == 0 ){`
	`                userAddresses.push(_userAddress[i]);`
	`            }`
	`            userPurchaseHistory[_userAddress[i]].packageIds = userData.packageIds;`
	`            userPurchaseHistory[_userAddress[i]].purchaseTime = userData.purchaseTime;`
	`            userPurchaseHistory[_userAddress[i]].totalPurchase = userData.totalPurchase;`
	`        }`
	`        emit OldUsersAdded(_userAddress);`
	`    }`

addPackage(
_name string,
_tetherAmount uint256,
_ownerPurchaseWage uint256,
_tokenPlanWage uint256,
_networkPlanWage uint256,
_tokenOwnerWage uint256,
_tokenIncreaseRate uint256)
	`function addPackage(string memory _name, uint256 _tetherAmount, uint256 _ownerPurchaseWage,uint256 _tokenPlanWage, uint256 _networkPlanWage, uint256 _tokenOwnerWage, uint256 _tokenIncreaseRate) external onlyRole(OPERATOR_ROLE) {`
	`        require((_tokenPlanWage + _networkPlanWage) == 100000, "Total must be 1");`
	`        packages.push(Package({`
	`            name: _name,`
	`            tetherAmount: _tetherAmount,`
	`            ownerPurchaseWage: _ownerPurchaseWage,`
	`            tokenPlanWage: _tokenPlanWage,`
	`            networkPlanWage: _networkPlanWage,`
	`            tokenOwnerWage: _tokenOwnerWage,`
	`            tokenIncreaseRate: _tokenIncreaseRate,`
	`            status: true`
	`        }));`
	`        emit PackageAdded(packages.length - 1, _name, _tetherAmount);`
	`    }`

Function	**Code**

buy(_packageId uint256, _txId uint256)	function buy(uint256 _packageId, uint256 _txId) public {
	require(_packageId < packages.length, "Invalid package");
	Package storage _package = packages[_packageId];
	require(_package.status, "Package is not active.");
	
	uint256 mintAmount = 0;
	uint256 _tetherAmount = _package.tetherAmount;
	uint256 ownerAmount = (_tetherAmount * _package.ownerPurchaseWage) / 100000;
	uint256 recivedAmount = _tetherAmount + ownerAmount;
	
	require(tetherToken.allowance(msg.sender, address(this)) >= recivedAmount, "Token allowance not enough");
	require(tetherToken.balanceOf(msg.sender) >= recivedAmount, "Insufficient token balance");
	tetherToken.transferFrom(msg.sender, address(this), recivedAmount);
	
	if (_package.networkPlanWage != 0) {
	// calculating network amount
	uint256 networkPlaneAmount = (_tetherAmount * _package.networkPlanWage) / 100000;
	uint256 matchingBonusAmount = (networkPlaneAmount * 40000) / 100000;
	uint256 balancerAmount =(networkPlaneAmount * 50000) / 100000;
	ownerAmount += (networkPlaneAmount * 10000) / 100000;
	
	// transfer tether to the matchingBonusPool and balancerPool
	
	tetherToken.transfer(poolAddresses[uint256(Pools.matchingBonusPool)], matchingBonusAmount);
	emit poolBalanceUpdated(poolAddresses[uint256(Pools.matchingBonusPool)], address(this), matchingBonusAmount);
	
	tetherToken.transfer(poolAddresses[uint256(Pools.balancerPool)], balancerAmount);
	emit poolBalanceUpdated(poolAddresses[uint256(Pools.balancerPool)], address(this), balancerAmount);
	
	}
	if (_package.tokenPlanWage != 0) {
	// calculating mint amount
	uint256 tokenPlaneAmount = (_tetherAmount * _package.tokenPlanWage) / 100000;
	uint256 tokenOwnerAmount = (tokenPlaneAmount * _package.tokenOwnerWage) / 100000;
	uint256 tokenIncreasevalue = (tokenPlaneAmount * _package.tokenIncreaseRate) / 100000;
	uint256 userTokenPlaneAmount = tokenPlaneAmount - tokenOwnerAmount - tokenIncreasevalue;
	ownerAmount += tokenOwnerAmount;
	
	_mintToken(msg.sender, userTokenPlaneAmount);
	_updatePrice(tokenIncreasevalue);
	}
	
	tetherToken.transfer(poolAddresses[uint256(Pools.ownersPool)], ownerAmount);
	emit poolBalanceUpdated(poolAddresses[uint256(Pools.ownersPool)], address(this), ownerAmount);
	if (userPurchaseHistory[msg.sender].totalPurchase == 0){
	userAddresses.push(msg.sender);
	}

Function	**Code**

changePackageStatus(_packageId uint256, _status bool)	`function changePackageStatus(uint256 _packageId, bool _status) external onlyRole(DEFAULT_ADMIN_ROLE) returns (bool) {`
	`Package storage _package = packages[_packageId];`
	`if (_status) {`
	`_package.status = _status;`
	`} else {`
	`_package.status = _status;`
	`}`
	`emit PackageStatusUpdated(_packageId, _package.name, _package.status);`
	`return true;`
	`}`

endOfUsersDataMigration()	`function endOfUsersDataMigration() external onlyRole(DEFAULT_ADMIN_ROLE) {`
	`require(addingOldUsers, "Already disabled");`
	`addingOldUsers = false;`
	`emit EndOfUsersDataMigration(block.timestamp, addingOldUsers);`
	`}`

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

poolWithdrawal(receiver address, _amount uint256)	`function poolWithdrawal(address receiver, uint256 _amount) public onlyRole(POOL_ROLE) returns (bool) {`
	`uint256 _tokenIncreasevalue = (_amount * 2)/100;`
	`_mintToken(receiver, (_amount- _tokenIncreasevalue));`
	`_updatePrice(_tokenIncreasevalue);`
	`return (true);`
	`}`

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);`
	`}`

sell(uint256 _amount)	`function sell(uint256 _amount) public {`
	`require(apexiaToken.balanceOf(msg.sender) >= _amount, "Insufficient token balance");`
	`apexiaToken.burn(msg.sender, _amount);`
	
	`uint256 tetherAmount = (currentPrice * _amount) / (10 ** uint256(30));`
	`uint256 sellWage = (tetherAmount * 2) / 100;`
	
	`_updatePrice(sellWage/2);`
	
	`tetherToken.transfer(poolAddresses[uint256(Pools.ownersPool)], (sellWage/2));`
	`emit poolBalanceUpdated(poolAddresses[uint256(Pools.ownersPool)], address(this), (sellWage/2));`
	`tetherToken.transfer(msg.sender, (tetherAmount - sellWage));`
	`emit Withdrawal(address(this), msg.sender, _amount);`
	`}`

Function	**Code**

setPoolAddresses(address[] memory _poolAddresses)	`function setPoolAddresses(address[] memory _poolAddresses) external onlyRole(DEFAULT_ADMIN_ROLE) {`
	`require(_poolAddresses.length == 3, "Must provide exactly 3 addresses");`
	
	`for (uint i = 0; i < _poolAddresses.length; i++) {`
	`require(_poolAddresses[i] != address(0), "Invalid address provided");`
	`poolAddresses[i] = _poolAddresses[i];`
	`grantRole(POOL_ROLE, _poolAddresses[i]);`
	`emit PoolAddressUpdated(Pools(i), _poolAddresses[i]);`
	`}`
	`}`

updatePoolAddress(
Pools _poolNumber,
address _poolAddress)
	`function updatePoolAddress(Pools _poolNumber, address _poolAddress) external onlyRole(DEFAULT_ADMIN_ROLE) {`
	`require(_poolAddress != address(0), "Invalid address");`
	`revokeRole(POOL_ROLE,poolAddresses[uint(_poolNumber)]);`
	`poolAddresses[uint(_poolNumber)] = _poolAddress;`
	`grantRole(POOL_ROLE, _poolAddress);`
	`emit PoolAddressUpdated(_poolNumber, _poolAddress);`
	`}`

updateTokenPrice(uint256 _amount)	`function updateTokenPrice(uint256 _amount) public returns (bool) {`
	`require(tetherToken.allowance(msg.sender, address(this)) >= _amount, "Token allowance not enough");`
	`require(tetherToken.balanceOf(msg.sender) >= _amount, "Insufficient token balance");`
	`tetherToken.transferFrom(msg.sender, address(this), _amount);`
	
	`_updatePrice(_amount);`
	`emit TokenPriceUpdatedByUser(msg.sender, _amount, currentPrice);`
	`return (true);`
	`}`

updateTokenPriceOnlyPools(
uint256 _amount)
	`function updateTokenPriceOnlyPools(uint256 _amount) external onlyRole(POOL_ROLE) returns (bool) {`
	`_updatePrice(_amount);`
	`return (true);`
	`}`