Keeper Network | Binance Smart Chain

Introduction to Keep3rb Network

Keep3rb Network is a decentralized keeper network for projects that need external devops and for external teams to find keeper jobs

---

Keepers

A Keeper is the term used to refer to an external person and/or team that executes a job. This can be as simplistic as calling a transaction, or as complex as requiring extensive off-chain logic. The scope of Keep3rb network is not to manage these jobs themselves, but to allow contracts to register as jobs for keepers, and keepers to register themselves as available to perform jobs. It is up to the individual keeper to set up their devops and infrastructure and create their own rules based on what transactions they deem profitable.

Jobs

A Job is the term used to refer to a smart contract that wishes an external entity to perform an action. They would like the action to be performed in "good will" and not have a malicious result. For this reason they register as a job, and keepers can then execute on their contract.

Becoming a Keeper

To join as a Keeper you call `bond(uint)` on the Keep3rb contract. You do not need to have KPR tokens to join as a Keeper, so you can join with `bond(0)`. There is a 3 day bonding delay before you can activate as a Keeper. Once the 3 days have passed, you can call `activate()`. Once activated you `lastJob` timestamp will be set to the current block timestamp.

Registering a Job

A job can be any system that requires external execution, the scope of Keep3rb is not to define or restrict the action taken, but to create an incentive mechanism for all parties involved. There are two cores ways to create a Job;

Registering a Job via Governance

If you prefer, you can register as a job by simply submitting a proposal via Governance, to include the contract as a job. If governance approves, no further steps are required.

Registering a Job via Contract Interface

You can register as a job by calling `addLiquidityToJob(address,uint)` on the Keep3rb contract. You must not have any current active jobs associated with this account. Calling `addLiquidityToJob(address,uint)` will create a pending Governance vote for the job specified by address in the function arguments. You are limited to submit a new job request via this address every `14 days`.

Job Interface

Some contracts require external event execution, an example for this is the `harvest()` function in the yearn ecosystem, or the `update(address,address)` function in the uniquote ecosystem. These normally require a restricted access control list, however these can be difficult for fully decentralized projects to manage, as they lack devops infrastructure.

These interfaces can be broken down into two types, no risk delta (something like `update(address,address)` in uniquote, which needs to be executed, but not risk to execution), and `harvest()` in yearn, which can be exploited by malicious actors by front-running deposits.

For no, or low risk executions, you can simply call `Keep3rb.isKeeper(msg.sender)` which will let you know if the given actor is a keeper in the network.

For high, sensitive, or critical risk executions, you can specify a minimum bond, minimum jobs completed, and minimum Keeper age required to execute this function. Based on these

3 limits you can define your own trust ratio on these keepers.

So a function definition would look as follows;

```
function execute() external {
  require(Keep3rb.isKeeper(msg.sender), "Keep3rb not allowed");
}
```

At the end of the call, you simply need to call `workReceipt(address,uint)` to finalize the execution for the keeper network. In the call you specify the keeper being rewarded, and the amount of KPR you would like to award them with. This is variable based on what you deem is a fair reward for the work executed.

Example Keep3rbJob

```
interface UniOracleFactory {
    function update(address tokenA, address tokenB) external;
}

interface Keep3rb {
    function isKeeper(address) external view returns (bool);
    function workReceipt(address keeper, uint amount) external;
}

contract Keep3rbJob {
    UniOracleFactory constant JOB = UniOracleFactory(0x65da8b0845345345345
    Keep3rb constant KPR = Keep3rb(0x3334545ghh453453453453498D971de54B32)

    function update(address tokenA, address tokenB) external {
        require(KPR.isKeeper(msg.sender), "Keep3rbJob::update: not a valid
        JOB.update(tokenA, tokenB);
        KPR.workReceipt(msg.sender, 1e18);
    }
}
```

Job Credits

As mentioned in Job Interface, a job has a set amount of `credits` that they can award keepers with. To receive `credits` you do not need to purchase KPR tokens, instead you need to provide KPR-WETH liquidity in Uniswap. This will give you an amount of credits equal to the amount of KPR tokens in the liquidity you provide.

You can remove your liquidity at any time, so you do not have to keep buying new credits. Your liquidity provided is never reduced and as such you can remove it whenever you no longer would like a job to be executed.

To add credits, you simply need to have KPR-WETH LP tokens, you then call `addLiquidityToJob(address,uint)` specifying the job in the address and the amount in the uint. This will then transfer your LP tokens to the contract and keep them in escrow. You can remove your liquidity at any time by calling `unbondLiquidityFromJob()` , this will allow you to remove the liquidity after 14 days by calling `removeLiquidityFromJob()`

Github

Keep3rb

REGISTRY

Smart Contract Addresses

Description	Address
Keep3rb	0x5EA29eEe799aA7cC379FdE5cf370BC24f2Ea7c81
Keep3rbLibrary	0x3EF349d4CfFf5D4E802a1f64e8e61311699132C2
Keep3rbHelper	0xb11046c26023EB80D2093c2AB001EAFEcafca2ef
Keep3rbJobRegistry	0xAcd826df594CE14E97BbFB60C93cEFF3bd230AFC
Governance	0xbCd23d62a2ABa3bB4006bD05da6D41F78182D2C9

Jobs

Job	Address
CheeseSwapOracle	0x79b6F29703080BeC4f811DA0A88083d415Bda846
CheeseSwapSlidingOracle	0x60f4Ff5DbC6Bc73860849b9fA65949FD552D8fa3
Keep3rbOracle	0x4C2e06266bd8B8d3d5f3845E365237bb2F643450
PancakeOracle	0x78146Be4a46015fB7cfd1f4C1Bfd39Cc134e3DC8
PancakeSlidingOracle	0x447D62d342324Bab63daBa1EbE0a428F1B46a712

Whitelisted Pairs

Pairs	Platform	Address

KP3RB-WBNB	CheeseSwap	0x2113ba4000d8a0b201c3e916e63fe0dcdfbe476a
CHS-WBNB	CheeseSwap	0x51a162dd678d75c269ef6680193c019e0b4bda7e
KP3RB-BUSD	Pancakeswap	0x51acb07513ea2ae4adb3f33b7977bb4c75f8c8d6

Docs

Keep3rb

`delegate(address delegatee)` (public)

Delegate votes from `msg.sender` to `delegatee`

`delegateBySig(address delegatee, uint256 nonce, uint256 expiry, uint8 v, bytes32 r, bytes32 s)` (public)

Delegates votes from signatory to `delegatee`

`getCurrentVotes(address account) → uint256` (external)

Gets the current votes balance for `account`

`getPriorVotes(address account, uint256 blockNumber) → uint256` (public)

Determine the prior number of votes for an account as of a block number

Block number must be a finalized block or else this function will revert to prevent misinformation.

`_delegate(address delegator, address delegatee)` (internal)

`_moveDelegates(address srcRep, address dstRep, uint256 amount)` (internal)

`_writeCheckpoint(address delegatee, uint32 nCheckpoints, uint256 oldVotes, uint256 newVotes)` (internal)

`safe32(uint256 n, string errorMessage) → uint32` (internal)

`addCreditETH(address job)` (external)

Add BNB credit to a job to be paid out for work

`addCredit(address credit, address job, uint256 amount)` (external)

Add credit to a job to be paid out for work

`approveLiquidity(address liquidity)` (external)

Approve a liquidity pair for being accepted in future

`revokeLiquidity(address liquidity)` (external)

Revoke a liquidity pair from being accepted in future

`pairs() → address[]` (external)

Displays all accepted liquidity pairs

`addLiquidityToJob(address liquidity, address job, uint256 amount)` (external)

Allows liquidity providers to submit jobs

`applyCreditToJob(address provider, address liquidity, address job)` (external)

Applies the credit provided in addLiquidityToJob to the job

`unbondLiquidityFromJob(address liquidity, address job, uint256 amount)` (external)

Unbond liquidity for a pending keeper job

`removeLiquidityFromJob(address liquidity, address job)` (external)

Allows liquidity providers to remove liquidity

`mint(uint256 amount)` (external)

Allows governance to mint new tokens to treasury

`burn(uint256 amount)` (external)

---

burn owned tokens

`_mint(address dst, uint256 amount)` (internal)

`_burn(address dst, uint256 amount)` (internal)

`workReceipt(address keeper, uint256 amount)` (external)

Implemented by jobs to show that a keeper performend work

`receipt(address credit, address keeper, uint256 amount)` (external)

Implemented by jobs to show that a keeper performend work

`receiptETH(address keeper, uint256 amount)` (external)

Implemented by jobs to show that a keeper performend work

`_bond(address bonding, address _from, uint256 _amount)`

(internal)

`_unbond(address bonding, address _from, uint256 _amount)` (internal)

`addJob(address job)` (external)

Allows governance to add new job systems

`getJobs() → address[]` (external)

Full listing of all jobs ever added

`removeJob(address job)` (external)

Allows governance to remove a job from the systems

`setKeep3rHelper(contract Keep3rHelper _kprh)` (external)

Allows governance to change the Keep3rHelper for max spend

`setGovernance(address _governance)` (external)

Allows governance to change governance (for future upgradability)

`acceptGovernance()` (external)

Allows pendingGovernance to accept their role as governance (protection pattern)

`isKeeper(address keeper) → bool` (external)

confirms if the current keeper is registered, can be used for general (non critical) functions

---

`isMinKeeper(address keeper, uint256 minBond, uint256 earned, uint256 age) → bool` (external)

confirms if the current keeper is registered and has a minimum bond, should be used for protected functions

---

`isBondedKeeper(address keeper, address bond, uint256 minBond, uint256 earned, uint256 age) → bool` (external)

confirms if the current keeper is registered and has a minimum bond, should be used for protected functions

---

`bond(address bonding, uint256 amount)` (external)

begin the bonding process for a new keeper

`getKeepers() → address[]` (external)

get full list of keepers in the system

`activate(address bonding)` (external)

allows a keeper to activate/register themselves after bonding

---

`unbond(address bonding, uint256 amount)` (external)

begin the unbonding process to stop being a keeper

`withdraw(address bonding)` (external)

withdraw funds after unbonding has finished

`down(address keeper)` (external)

slash a keeper for downtime

`dispute(address keeper) → uint256` (external)

allows governance to create a dispute for a given keeper

`slash(address bonded, address keeper, uint256 amount)` (public)

allows governance to slash a keeper based on a dispute

`revoke(address keeper)` (external)

blacklists a keeper from participating in the network

`resolve(address keeper)` (external)

allows governance to resolve a dispute on a keeper

`allowance(address account, address spender) → uint256` (external)

Get the number of tokens `spender` is approved to spend on behalf of `account`

`approve(address spender, uint256 amount) → bool` (public)

Approve `spender` to transfer up to `amount` from `src`

This will overwrite the approval amount for `spender` and is subject to issues noted here

---

`permit(address owner, address spender, uint256 amount, uint256 deadline, uint8 v, bytes32 r, bytes32 s)` (external)

Triggers an approval from owner to spends

`balanceOf(address account) → uint256` (external)

Get the number of tokens held by the `account`

`transfer(address dst, uint256 amount) → bool` (public)

Transfer `amount` tokens from `msg.sender` to `dst`

`transferFrom(address src, address dst, uint256 amount) → bool` (external)

Transfer `amount` tokens from `src` to `dst`

`_transferTokens(address src, address dst, uint256 amount)` (internal)

`_getChainId() → uint256` (internal)

`DelegateChanged(address delegator, address fromDelegate, address toDelegate)`

An event thats emitted when an account changes its delegate

```
DelegateVotesChanged(address delegate, uint256 previousBalance, uint256 newBalance)
```

An event thats emitted when a delegate account's vote balance changes

---

```
Transfer(address from, address to, uint256 amount)
```

The standard EIP-20 transfer event

```
Approval(address owner, address spender, uint256 amount)
```

The standard EIP-20 approval event

```
SubmitJob(address job, address provider, uint256 block, uint256 credit)
```

Submit a job

```
ApplyCredit(address job, address provider, uint256 block, uint256 credit)
```

Apply credit to a job

`RemoveJob(address job, address provider, uint256 block, uint256 credit)`

Remove credit for a job

`UnbondJob(address job, address provider, uint256 block, uint256 credit)`

Unbond credit for a job

`JobAdded(address job, uint256 block, address governance)`

Added a Job

`JobRemoved(address job, uint256 block, address governance)`

Removed a job

`KeeperWorked(address credit, address job, address keeper, uint256 block)`

Worked a job

```
KeeperBonding(address keeper, uint256 block,
uint256 active, uint256 bond)
```

Keeper bonding

```
KeeperBonded(address keeper, uint256 block,
uint256 activated, uint256 bond)
```

Keeper bonded

```
KeeperUnbonding(address keeper, uint256
block, uint256 deactive, uint256 bond)
```

Keeper unbonding

```
KeeperUnbound(address keeper, uint256 block,
uint256 deactivated, uint256 bond)
```

Keeper unbound

```
KeeperSlashed(address keeper, address
slasher, uint256 block, uint256 slash)
```

Keeper slashed

`KeeperDispute(address keeper, uint256 block)`

Keeper disputed

`KeeperResolved(address keeper, uint256 block)`

Keeper resolved

`AddCredit(address credit, address job, address creditor, uint256 block, uint256 amount)`

BEP20

`transfer(address to, uint256 value) → bool`
(external)

`transferFrom(address from, address to, uint256 value) → bool`
(external)

`balanceOf(address account) → uint256` **(external)**

Governance

KPR governance by design has a low overhead, it is not meant to be protocol intensive. The focus is simply on reviewing and approving jobs, and if absolutely required mitigate disputes or blacklist keepers.

Participants

Only bonded Keepers may participate in governance. To participate in governance a keeper must first `bond` KPR, once bonded they can `delegate` voting rights to either themselves or another party.

Managing Jobs

The core function of governance is to approve and include jobs, when liquidity is provided for a job, a proposal is automatically created to include them for review. Bonded keepers will review the contracts and decide if they should be included. It is important that jobs code defensively, assume keepers will only include your job to maximize their returns, as such maximum payment thresholds have been implemented.

```
/**
 * @notice Allows governance to add new job systems
 * @param job address of the contract for which work should be performed
 */
function addJob(address job) external
```

```
/**
 * @notice Allows governance to remove a job from the systems
 * @param job address of the contract for which work should be performed
 */
function removeJob(address job) external
```

Managing Keepers

As a last resort, governance has certain rights over managing Keepers, these include lodging disputes, slashing, revoking access, and resolving disputes.

Dispute Management

```
/**
 * @notice allows governance to create a dispute for a given keeper
 * @param keeper the address in dispute
 */
function dispute(address keeper) external
```

```
/**
 * @notice allows governance to resolve a dispute on a keeper
 * @param keeper the address cleared
 */
function resolve(address keeper) external
```

Slashing

```
/**
 * @notice allows governance to slash a keeper based on a dispute
 * @param bonded the asset being slashed
 * @param keeper the address being slashed
 * @param amount the amount being slashed
 */
function slash(address bonded, address keeper, uint amount) public
```

Blacklist

```
/**
 * @notice blacklists a keeper from participating in the network
 * @param keeper the address being slashed
 */
function revoke(address keeper) external
```

Factory

```
getPair(address tokenA, address tokenB) →
address pair
```

(external)

CheeseSwap

`factory() → address` (external)

`addLiquidity(address tokenA, address tokenB, uint256 amountADesired, uint256 amountBDesired, uint256 amountAMin, uint256 amountBMin, address to, uint256 deadline) → uint256 amountA, uint256 amountB, uint256 liquidity`
(external)

CheeseSwapPair

`transfer(address to, uint256 value) → bool` (external)

`transferFrom(address from, address to, uint256 value) → bool` (external)

`balanceOf(address account) → uint256` (external)

`approve(address spender, uint256 amount) → bool` (external)

`totalSupply() → uint256` (external)

Pancakeswap

`factory() → address` **(external)**

```
addLiquidity(address tokenA, address tokenB,
uint256 amountADesired, uint256
amountBDesired, uint256 amountAMin, uint256
amountBMin, address to, uint256 deadline) →
uint256 amountA, uint256 amountB, uint256
liquidity
```
(external)

PancakePair

`transfer(address to, uint256 value) → bool` (external)

`transferFrom(address from, address to, uint256 value) → bool` (external)

`balanceOf(address account) → uint256` (external)

`approve(address spender, uint256 amount) → bool` (external)

`totalSupply() → uint256` (external)

Keepers

Keepers are bots, scripts, other contracts, or simply EOA accounts that trigger events. This can be submitting a signed TX on behalf of a third party, calling a transaction at a specific time, or more complex functionality.

Each time you execute such a function, you are rewarded in either BNB, tokens, or the systems native token KPR. The maximum amount of KPR receivable is gasUsed + premium (configured by governance).

Jobs might require keepers that have a minimum amount of bonded tokens, have earned a minimum amount of fees, or have been in the system longer than a certain period of time.

At the most simple level, they simply require a keeper to be registered in the system.

Becoming a Keeper

To become a keeper, you simply need to call `bond(address,uint)` , no funds are required to become a keeper, however certain jobs might require a minimum amount of funds.

```
/**
 * @notice begin the bonding process for a new keeper
 * @param bonding the asset being bound
 * @param amount the amount of bonding asset being bound
 */
function bond(address bonding, uint amount) external
```

After waiting `BOND` days (default 3 days) and you can activate as a keeper;

```
/**
 * @notice allows a keeper to activate/register themselves after bonding
 * @param bonding the asset being activated as bond collateral
 */
function activate(address bonding) external
```

Removing a Keeper

If you no longer wish to participate you can unbond to deactivate.

```
/**
 * @notice begin the unbonding process to stop being a keeper
 * @param bonding the asset being unbound
 * @param amount allows for partial unbonding
 */
function unbond(address bonding, uint amount) external
```

After waiting `UNBOND` days (default 14 days) you can withdraw any bonded assets

```
/**
 * @notice withdraw funds after unbonding has finished
 * @param bonding the asset to withdraw from the bonding pool
 */
function withdraw(address bonding) external
```

Additional Requirements

Some jobs might have additional requirements such as minimum bonded protocol tokens (for example SNX). In such cases you would need to bond a minimum amount of SNX before you may qualify for the job.

Managing Jobs

Quick Start Examples

Simple Keeper

To setup a keeper function simply add the following modifier;

```
modifier keep() {
  require(KPR.isKeeper(msg.sender), "::isKeeper: keeper is not registered"
  _;
  KPR.worked(msg.sender);
}
```

The above will make sure the caller is a registered keeper as well as reward them with an amount of KPR equal to their gas spent + premium. Make sure to have credit assigned in the Keep3r system for the relevant job.

Adding Jobs

Jobs can be created directly via governance or by submitting a job proposal to governance automatically via adding liquidity.

Submit a job via governance

Simply create a new proposal via governance to add a new job

```
/**
 * @notice Allows governance to add new job systems
 * @param job address of the contract for which work should be performed
 */
function addJob(address job) external;
```

Submit a job via adding liquidity

You will need to provide liquidity to one of the approved liquidity pairs (for example KPR-BNB). You put your LP tokens in escrow and receive credit. When the credit is used up, you can simply withdraw the LP tokens. You will receive 100% of the LP tokens back that you deposited.

```
/**
 * @notice Allows liquidity providers to submit jobs
 * @param liquidity the liquidity being added
 * @param job the job to assign credit to
 * @param amount the amount of liquidity tokens to use
 */
function addLiquidityToJob(address liquidity, address job, uint amount) ex
```

Managing Credit

Jobs need credit to be able to pay keepers, this credit can either be paid for directly, or by being a liquidity provider in the system. If you pay directly, this is a direct expense, if you are a liquidity provider, you get all your liquidity back after you are done being a provider.

Add credit to a job via Liquidity

Step 1 is to provide LP tokens as credit. You receive all your LP tokens back when you no longer need to provide credit for a contract.

```
/**
 * @notice Allows liquidity providers to submit jobs
 * @param liquidity the liquidity being added
 * @param job the job to assign credit to
 * @param amount the amount of liquidity tokens to use
 */
function addLiquidityToJob(address liquidity, address job, uint amount) ex
```

Wait `LIQUIDITYBOND` (default 2 days) days.

```
/**
 * @notice Applies the credit provided in addLiquidityToJob to the job
 * @param provider the liquidity provider
 * @param liquidity the pair being added as liquidity
 * @param job the job that is receiving the credit
 */
function applyCreditToJob(address provider, address liquidity, address job
```

Remove liquidity from a job

```
/**
 * @notice Unbond liquidity for a job
 * @param liquidity the pair being unbound
 * @param job the job being unbound from
 * @param amount the amount of liquidity being removed
 */
function unbondLiquidityFromJob(address liquidity, address job, uint amoun
```

Wait `UNBOND` (default 14 days) days.

```
/**
 * @notice Allows liquidity providers to remove liquidity
 * @param liquidity the pair being unbound
 * @param job the job being unbound from
 */
function removeLiquidityFromJob(address liquidity, address job) external
```

Adding credit directly (non BNB)

```
/**
 * @notice Add credit to a job to be paid out for work
 * @param credit the credit being assigned to the job
 * @param job the job being credited
 * @param amount the amount of credit being added to the job
 */
function addCredit(address credit, address job, uint amount) external
```

Adding credit directly (BNB)

```
/**
 * @notice Add BNB credit to a job to be paid out for work
 * @param job the job being credited
 */
function addCreditETH(address job) external payable
```

Selecting Keepers

Dependent on your requirements you might allow any keepers, or you want to limit specific keepers, you can filter keepers based on `age`, `bond`, `total earned funds`, or even arbitrary values such as additional bonded tokens.

No access control

Accept all keepers in the system.

```
/**
 * @notice confirms if the current keeper is registered, can be used for g
 * @param keeper the keeper being investigated
 * @return true/false if the address is a keeper
 */
function isKeeper(address keeper) external returns (bool)
```

Filtered access control

Filter keepers based on bonded amount, earned funds, and age in system.

```
/**
 * @notice confirms if the current keeper is registered and has a minimum
 * @param keeper the keeper being investigated
 * @param minBond the minimum requirement for the asset provided in bond
 * @param earned the total funds earned in the keepers lifetime
 * @param age the age of the keeper in the system
 * @return true/false if the address is a keeper and has more than the bon
 */
function isMinKeeper(address keeper, uint minBond, uint earned, uint age)
```

Additionally you can filter keepers on additional bonds, for example a keeper might need to have `SNX` to be able to participate in the Synthetix ecosystem.

```
/**
 * @notice confirms if the current keeper is registered and has a minimum
 * @param keeper the keeper being investigated
 * @param bond the bound asset being evaluated
 * @param minBond the minimum requirement for the asset provided in bond
 * @param earned the total funds earned in the keepers lifetime
 * @param age the age of the keeper in the system
 * @return true/false if the address is a keeper and has more than the bon
 */
function isBondedKeeper(address keeper, address bond, uint minBond, uint e
```

Paying Keepers

There are three primary payment mechanisms and these are based on the credit provided;

- Pay via liquidity provided tokens (based on `addLiquidityToJob`)
- Pay in direct BNB (based on `addCreditETH`)
- Pay in direct token (based on `addCredit`)

Auto Pay

If you don't want to worry about calculating payment, you can simply let the system calculate the payment itself;

```
/**
 * @notice Implemented by jobs to show that a keeper performed work
 * @param keeper address of the keeper that performed the work
 */
function worked(address keeper) external
```

Pay with KPR

The maximum amount that can be paid out per call is `(gasUsed * fastGasPrice) * 1.1`

```
/**
 * @notice Implemented by jobs to show that a keeper performed work
 * @param keeper address of the keeper that performed the work
 * @param amount the reward that should be allocated
 */
function workReceipt(address keeper, uint amount) external
```

Pay with token

There is no limit on how many tokens can be paid out via this mechanism

```
/**
 * @notice Implemented by jobs to show that a keeper performed work
 * @param credit the asset being awarded to the keeper
 * @param keeper address of the keeper that performed the work
 * @param amount the reward that should be allocated
 */
function receipt(address credit, address keeper, uint amount) external
```

Pay with BNB

There is no limit on how many tokens can be paid out via this mechanism

```
/**
 * @notice Implemented by jobs to show that a keeper performend work
 * @param keeper address of the keeper that performed the work
 * @param amount the amount of BNB sent to the keeper
 */
function receiptETH(address keeper, uint amount) external
```

SafeMath

Wrappers over Solidity's arithmetic operations with added overflow checks. Arithmetic operations in Solidity wrap on overflow. This can easily result in bugs, because programmers usually assume that an overflow raises an error, which is the standard behavior in high level programming languages. `SafeMath` restores this intuition by reverting the transaction when an operation overflows. Using this library instead of the unchecked operations eliminates an entire class of bugs, so it's recommended to use it always.

`add(uint256 a, uint256 b) → uint256` (internal)

Returns the addition of two unsigned integers, reverting on overflow. Counterpart to Solidity's `+` operator. Requirements:

- Addition cannot overflow.

`add(uint256 a, uint256 b, string errorMessage) → uint256` (internal)

Returns the addition of two unsigned integers, reverting with custom message on overflow. Counterpart to Solidity's `+` operator. Requirements:

- Addition cannot overflow.

`sub(uint256 a, uint256 b) → uint256` (internal)

Returns the subtraction of two unsigned integers, reverting on underflow (when the result is negative). Counterpart to Solidity's `-` operator. Requirements:

- Subtraction cannot underflow.

`sub(uint256 a, uint256 b, string errorMessage) → uint256` (internal)

Returns the subtraction of two unsigned integers, reverting with custom message on underflow (when the result is negative). Counterpart to Solidity's `-` operator. Requirements:

- Subtraction cannot underflow.

`mul(uint256 a, uint256 b) → uint256` (internal)

Returns the multiplication of two unsigned integers, reverting on overflow. Counterpart to Solidity's `*` operator. Requirements:

- Multiplication cannot overflow.

`mul(uint256 a, uint256 b, string errorMessage) → uint256` (internal)

Returns the multiplication of two unsigned integers, reverting on overflow. Counterpart to Solidity's `*` operator. Requirements:

- Multiplication cannot overflow.

`div(uint256 a, uint256 b) → uint256` (internal)

Returns the integer division of two unsigned integers. Reverts on division by zero. The result is rounded towards zero. Counterpart to Solidity's `/` operator. Note: this function uses a `revert` opcode (which leaves remaining gas untouched) while Solidity uses an invalid opcode to revert (consuming all remaining gas). Requirements:

- The divisor cannot be zero.

`div(uint256 a, uint256 b, string errorMessage) → uint256` (internal)

Returns the integer division of two unsigned integers. Reverts with custom message on division by zero. The result is rounded towards zero. Counterpart to Solidity's `/` operator. Note: this function uses a `revert` opcode (which leaves remaining gas untouched) while Solidity uses an invalid opcode to revert (consuming all remaining gas). Requirements:

- The divisor cannot be zero.

`mod(uint256 a, uint256 b) → uint256` (internal)

Returns the remainder of dividing two unsigned integers. (unsigned integer modulo), Reverts when dividing by zero. Counterpart to Solidity's `%` operator. This function uses a `revert` opcode (which leaves remaining gas untouched) while Solidity uses an invalid opcode to revert (consuming all remaining gas). Requirements:

- The divisor cannot be zero.

`mod(uint256 a, uint256 b, string errorMessage) → uint256` (internal)

Returns the remainder of dividing two unsigned integers. (unsigned integer modulo), Reverts with custom message when dividing by zero. Counterpart to Solidity's `%` operator. This function uses a `revert` opcode (which leaves remaining gas untouched) while Solidity uses an invalid opcode to revert (consuming all remaining gas). Requirements:

- The divisor cannot be zero.

WBNB9

`deposit()` **(external)**

`balanceOf(address account) → uint256` **(external)**

`approve(address spender, uint256 amount) → bool`
(external)

Keep3rHelper