KUB (KUB Chain) Technical Paper

(V 3.2)

Updated 10 March 2025

Table of contents

1. Introduction

Blockchain technology, which runs Bitcoin, has developed over the last decade into one of today's biggest ground-breaking technologies with the potential to impact every industry from financial to healthcare, to manufacturing, to educational institutions. However, Bitcoin (the first cryptocurrency) has only used blockchain as the infrastructure and created digital money with the ability to transfer value. Whereas Ethereum made a new paradigm shift in blockchain technology with programmable capability for digital money, global payments, and applications.

In the Blockchain Trilemma, Bitcoin and Ethereum originally used the Proof-of-Work (PoW) consensus type, which helps increase transparency and security. However, it generates environmental impact and the high cost of mining nodes that cause power consumption, leading to an implementation challenge, high transaction fees, and causing transaction delays. As a result, it creates a low rate of blockchain adoption and adds a barrier to people who aim to use blockchain technology.

To reconfigure and provide a better consensus solution, "KUB" (KUB Chain) introduced a technology platform and system that helped bridge the gap, lower impact on the environment, faster processing of transactions with lower fees, helps to save the mining cost, and solve cybersecurity issues.

"KUB Chain" started developing and reapplying the Ethereum blockchain [3] and forked from the Go-Ethereum [21] concept but changed the consensus from Proof-of-Work (PoW) based on Ethash [26] to Proof-of-Authority (PoA) based on Clique [10]. In the initial stage, KUB Chain uses the PoA consensus [9] to add the advantage of lower mining or running node cost and faster speed of verifying transactions. However, giving greater weight to scalability and increasing network performance from PoA might lower the decentralization level.

Therefore, to offer a better technology platform and system for blockchain adoption. Our team is developing a new consensus called Proof-of-Stake-Authority (PoSA) to increase the network's decentralization, corresponding and addressing a solution to all concerns from PoA, significantly improving network performance and providing truthful data with less environmental impact to democratize opportunity to use blockchain technology.

To summarize, the KUB Chain enhances the opportunity for everyone to participate in the decentralized economy without barriers to entry such as low transaction speed, high transaction fees, etc. KUB Chain was designed to be a public network providing a good user experience. Both blockchain and the application layer can increase the network's speed and scalability. It consists of increasing the block size, reducing the block time to maximize speed, and providing a dynamic gas price to stabilize the transaction fees. Moreover, KUB Chain has a specific infrastructure design for the validator set by connecting the validator node directly through the unlimited bandwidth network, which can help improve the synchronization and stability. In addition, on the application layer, we have a KAP which is a standard to implement the token and Wrapped KUB (KKUB), the KAP-20 version of KUB, which will be an important part of driving the De-Fi and DApp on the KUB Chain, and the HyperBlock system that combined off-chain and on-chain solutions together to improve the transaction scalability. Finally, all solutions provide the best experience for users, developers, and everyone to engage with the decentralized world seamlessly and efficiently through KUB Chain.

2. Blockchain paradigm

KUB Chain, a forked Ethereum base blockchain [3], can be interpreted as a transaction-based state machine. It starts from the first state, called the genesis state, and executes the transactions to produce some current state. The visualization can be shown in Figure-01.

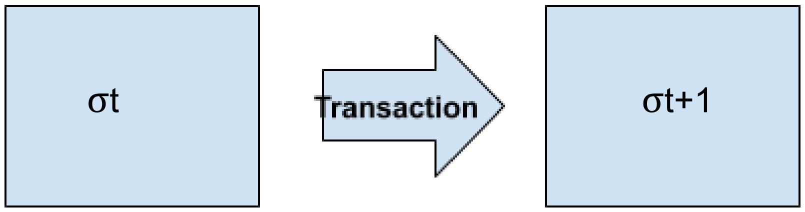


Figure-01

σt , $\sigma t+1$ is described as the previous state and current state, which is changed by the transaction (T) via the state transition function (Υ). According to the state machine concept, the general equation of state transformation can be written as follows (equation-1).

(1)[1] $$\sigma t+1 \equiv \Upsilon(\sigma t, T)$$

Transactions are packed into blocks and linked together to form a chain structure using a cryptographic hash. Using hash makes each block unique and unable to spoof. According to equation-2, a block (a series of transactions) can be applied to the equation. So, the equation is rewritten in terms of block-level as follows.

(2)[1] $$\sigma_{t+1} \equiv \Pi(\sigma_t, B)$$

(3)[1] $$B \equiv (...(T_0, T_1, ...), ...)$$

(4)[1] $$\Pi(\sigma, B) \equiv \Omega(B, \Upsilon(\Upsilon(\sigma, T_0), T_1)...)$$

From equation-3, a block(B) is a series of transactions and their components. After integrating with state-transition function (Υ) and block-finalization function (Ω), a block-level state-transition function (Π) is created. To summarize equation-2, 3, and 4, a new equation is formed as equation-5

(5) $$\sigma_{t+1} \equiv \Omega(B, \Upsilon(\Upsilon(\sigma_t, T_0), T_1)...)$$

2.1) Native currency units

KUB Chain has a native currency called **KUB** which is used to pay for transaction fees in order to reward node validators. The largest unit is KUB.

One KUB is defined as being 10^{18}. On the other hand, the smallest unit is called **Bit**. All sub denominations of KUB are listed in Table-01.

Power	Name
10^{0}	BIT
10^{9}	GBIT
10^{18}	KUB

Table-01

2.2) KUB Chain history

KUB Chain is a decentralized network; all validator nodes have the right to create a new block at every turn. However, in some cases, there is a possibility of producing a temporary forked block (chain is split) which occurs from coexisting with the states created by a different validator node. This situation can be shown in Figure-02.

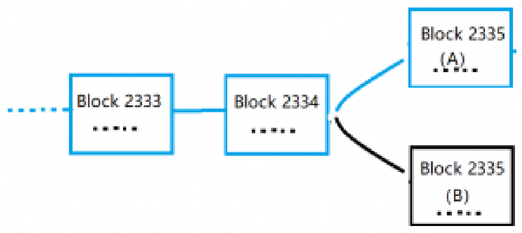


Figure-02 (source https://medium.com/@vrastromind)

The first chain created would be the main chain while the rest will be the uncle block as stated by the Ethereum GHOST protocol created by (Sompolinsky and Zohar ,2013). The uncle block will be discarded, but the canonical chain (the longest chain) as shown in Figure-03.

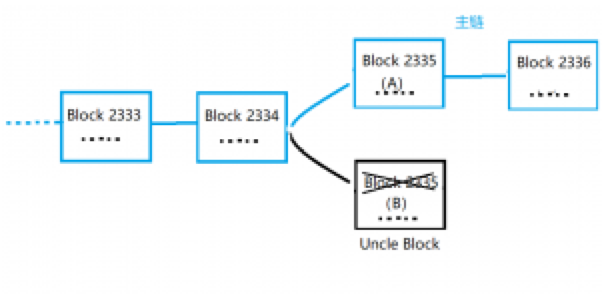


Figure-03 (source https://medium.com/@vrastromind)

Therefore, a new protocol must be implemented to upgrade the protocol occasionally. After implementation, it will modify an existing protocol and create a new one split from the original version, called "Hard Fork". As a result, the previous software using the old protocol version will be obsolete and have to be replaced with the latest software. In the first improvement phase, KUB Chain has improved and changed the consensus mechanism from Proof of Authority (PoA) to Proof of Stake Authority (PoSA) called Erawan Hard Fork [28]. Currently, KUB Chain has changed the consensus mechanism from PoSA to Proof of Stake (PoS) [Appendix F].

3. States & Tries

Like Ethereum, The trie is a core database in KUB Chain storing all significant states such as account, receipt, and transaction state.

3.1) Merkle Patricia Trie

The Merkle Patricia Trie [8] introduced by Ethereum [7] is a cryptographically authenticated data structure used to store key and value binding. It is a combination of the Merkle Tree and Patricia Tree (a binary radix tree) to improve the efficiency of the data's insertion, deletion, and selection. In addition, Merkle Patricia Trie is the primary data structure of KUB Chain (the same as Ethereum). There are three types of nodes: extension node, branch node, and leaf node [6].

3.1.1) Extension node

includes two items 1. `encodedPath` is a partial path encoded by RLP encoding function, and 2. its `key` is the next trie node to lookup.

3.1.2) Branch node

branch nodes are the path between one node to another node. For example, the node contains 17 array items. One will hold the possible value (hex character - nibble) in the path, and another one will hold the final target value after the path has been fully traversed.

3.1.3) Leaf node

is used to store the data, including two items inside a node encoded path and value. The `encodedPath` is the end path encoded by the RLP encoding function, and the `value` is the target value of the specific path.

3.2) States

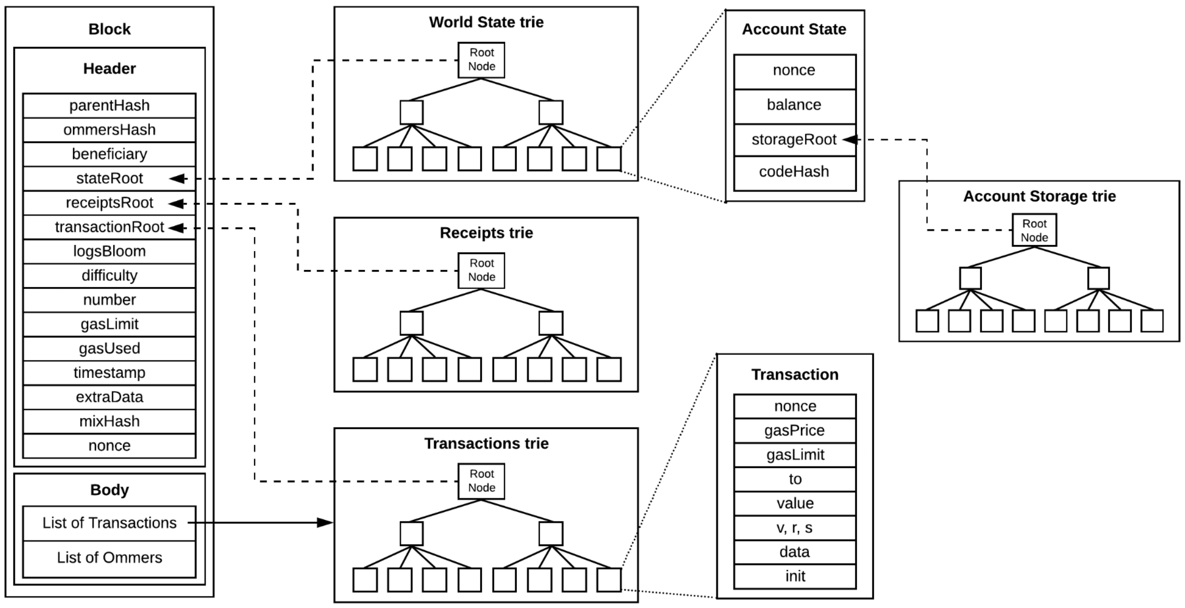


Figure-04: An overview of the tries [6]

3.2.1) State Root

KUB Chain stores the root of all the states in the block header. Every trie has a root, and each root is a 256-bit hash of the root node of each tree encoded into the trie as a mapping from the Keccak 256-bit hash of the 256-bit integer keys to the RLP-encoded 256-bit integer values.

(6) $$RootHash = KEC(RLP.encode(Trie))$$

Where KEC is a Keccak 256-bit hashing function, RLP.encode is an RLP encoding function, and the Trie is mapping the key-value set.

3.2.2) World State

The world state is one of the states that maps between addresses and account states. It is expressed that the world state and global state could be seen as one after it is constantly updated from transaction executions. Therefore, the balance of an account or the current state of a smart contract could be query from the world state.

a) World State Trie All the information about KUB Chain accounts is stored in the world state trie. The root node of the trie will be cryptographically hashed by using the Keccak-256 hashing function and included in the **stateRoot** field of the block header to represent the current state [14].

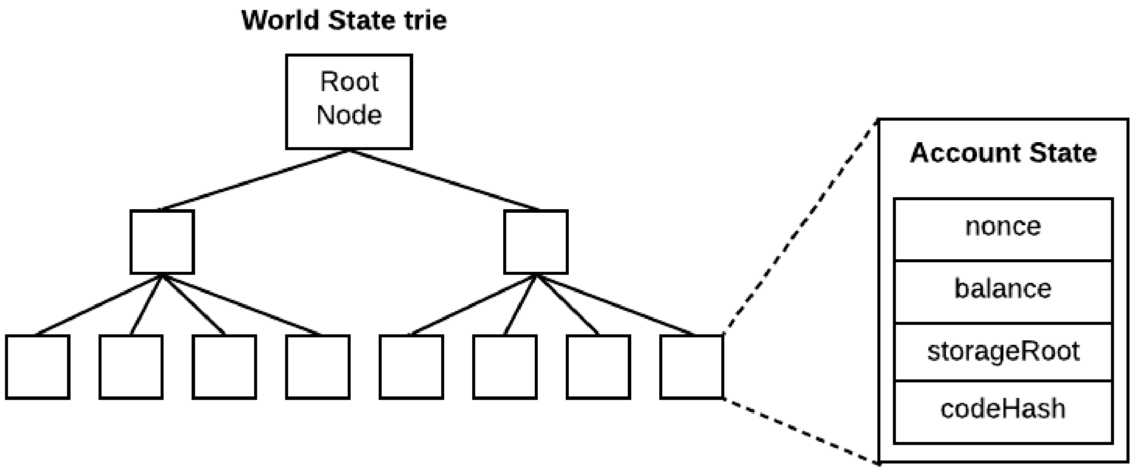


Figure-05: World State Trie [6]

b) Account State

i) Account state components

- **Nonce** is the number of transactions sent by an account.
- **Balance** is KUB balance of an account.
- **StorageRoot** consists of a 256-bit hash of the root node of the account storage trie.
- **codeHash** is the hash of the EVM code of an account.

ii) Account Storage Trie

The account storage trie is used to store the states of the contract accounts, as for External-Own Accounts (EOAs) the storageRoot will be empty. All smart contract data is persisted in the account storage trie as a mapping between 32-bytes integers.

3.2.3) Transaction State

a) Transaction state components

i) Nonce is the number of transactions sent by the account.
ii) gasPrice is the number of Bit that will be paid per unit of gas.
iii) gasLimit is the maximum amount of gas allowed to use for executing the transaction.
iv) To is the recipient address for a normal transaction, for a contract creation transaction will be zero address.
v) Value is the number of Bit that transferred to the recipient.
vi) R, S, V are the values used in the cryptographic signature to determine the sender's validity of the transaction.
vii) AccessList is a list of addresses and storage keys.
viii) Chain ID is the specific ID of the chain.
ix) Data is an input data in byte array of the message call (for transferring the value and sending a message call to a smart contract).
x) Init is the EVM-code utilized for initialization of the smart contract. (for contract creation only).

b) Transaction Trie

stores all of the transactions included in a block. The block header (in the transactionsRoot field) will include the hash of the root node of the transaction trie.

3.2.4) Transaction Receipt

a) Transaction Receipt Trie

stores all transaction receipts in the same block practice as Transaction Trie. The hash of the root node of the transaction receipts trie is also included in the receiptsRoot field of the block header.

4. GAS

All transactions sent on KUB Chain are subjected to fees called **Gas** to help prevent users from abusing their transactions and support the validator node's maintenance cost. The prices of gas units in each transaction varies based on the computational instructions of the Ethereum Virtual Machine (EVM), data length, memory usage, etc. (see Appendix A). The amount of gas collected is transferred to the in-turn validator node wallet.

4.1) Gas Limit and Gas Price

As stated above, every transaction spending gas has a limitation of gas usage known as **gasLimit**. The prices of gas units are implicitly purchased from the sender's account balance. The purchase happens at the **gasPrice** (Wood, 2022). The gasPrice pegs to 5 Gbit to promote the circulation of KUB Coin (KUB Chain‘s Native coin) on KUB Chain network and the company's gas tank. In addition, the remaining gas after the transaction had been executed would be refunded to the sender's address at the same gasPrice that why it was called gasLimit, and can express by the following equation:

$$g_r = T_{gasL} - G_{used} \quad (7)$$

Where g_r is the amount of gas refund sent back to the sender's address. T_{gasL}, G_{used} means gasLimit set by the sender and actual gas used.

4.2 Mechanism of gas calculation

KUB Chain, forked Ethereum, uses the same method of gas calculation as Ethereum main network. The gas will be charged based on the gasLimit that the sender input and the remaining gas will be refunded after finalizing the process. Before any EVM code gets executed, some amount of gas was deducted from the sender's defined gasLimit. This amount is named **intrinsic gas**, $\boldsymbol{g_0}$, and can be described from equation (8).

$$g_0 \equiv \sum_{i \in T_{\mathbf{i}}, T_{\mathbf{d}}} \begin{cases} G_{\text{txdatazero}} & \text{if} \quad i = 0 \\ G_{\text{txdatanonzero}} & \text{otherwise} \end{cases} + \begin{cases} G_{\text{txcreate}} & \text{if} \quad T_{\text{t}} = \varnothing \\ 0 & \text{otherwise} \end{cases}$$

(8)[1]

$$+ G_{\text{transaction}} + \sum_{j=0}^{\|T_{\mathbf{A}}\|-1} \left(G_{\text{accesslistaddress}} + \|T_{\mathbf{A}}[j]_{\mathbf{s}}\| G_{\text{accessliststorage}} \right)$$

T_i, T_d stands for the series of bytes. $G_{txcreate}$ gets value when a transaction is invoked in the contract creation process. In other words, if a contract creation transaction does not exist, $G_{txcreate}$ will be zero in value. $G_{transaction}$ is the initial fee that all transactions on the KUB Chain network have to pay. $G_{accesslistdaardss}$ and $G_{accessliststorage}$ are the cost of warm-up address and storage access. After intrinsic gas has been charged, the transaction will be executed. There are two types of interaction with KUB Chain.

4.2.1) Native coin transaction

This type of transaction is simple and does not invoke any smart contract or smart contract creation. Sending a KUB coin to a regular wallet address (not a smart contract address) is the only one for this type. According to equation (8), It does not have $G_{txcreate}$, $G_{accesslistdaardss}$, $G_{txdatanonzero}$, and $G_{txdatanonzero}$. Moreover, it does not interact with EVM (Only g_0 occurs to calculate the gas fee). To summarize, G_{used} can be re-written as follows**.**

$$G_{\text{used}} \equiv G_{\text{transaction}}$$

4.2.2) Smart contract creation and interaction

After intrinsic gas has been charged, EVM code will be executed, and gas is consumed by EVM instruction (also called OPCODE). Each instruction has a specific amount of gas consumption that can be clearly explained by equation $C(\sigma, \mu, A, I)$ (Appendix B). As a result, the actual gas used (G_{used}) is determined by equation (9).

$$G_{used} = g_0 + C(\sigma, \mu, A, I) \tag{9}$$

This equation describes the actual gas consumed by the whole blockchain network. However, the gas calculation equation (9) is only writing action on smart contracts and a contraction creation transaction, but reading action from only wallet address does not pay the gas for calling the smart contract.

5. Execution Engine (EVM)

The KUB Chain execution engine relies on Ethereum Virtual Machine (EVM). For processing the transaction and managing the state transition, the Ethereum Virtual Machine is a stack-based machine that executes the bytecode and performs the operation based on a set of instructions to transform the state from the previous state to the new state [18]. The EVM is a Turing-complete machine, the total number of computations limited by the amount of gas to prevent such an attack, e.g., denial-of-service (DDoS) attacks. The EVM also supports exception handling to avoid the invalid state transition.

The main components of EVM include:

- **World state *(σ)*** is a non-volatile state that stores a significant state, such as the account state.
- **Machine state *(μ)*** is a volatile state used during the execution, such as gas available ***g***, the program counter ***pc,*** the memory content ***m,*** and the stack content ***s.***
- **Stack** is a last-in-first-out (LIFO) storage with two abstract operations: `PUSH` and `POP`.
- **Memory** is a word-addressed byte array used to store all instructions. The memory is volatile because its state will be reset after the computation ends.
- **Gas *(g)*** is the remaining gas of the current execution.
- **Program Counter *(pc)*** is always set to zero at the beginning of execution and increased with instructions being read from memory.
- **Virtual ROM** is an immutable memory where the EVM bytecode or the program code is stored and accessible only through a particular instruction, e.g., `CODECOPY` instruction.
- **Storage** is persistent storage, a key-value store. The storage is non-volatile because the blockchain's system state is permanently persisted.

EVM architecture

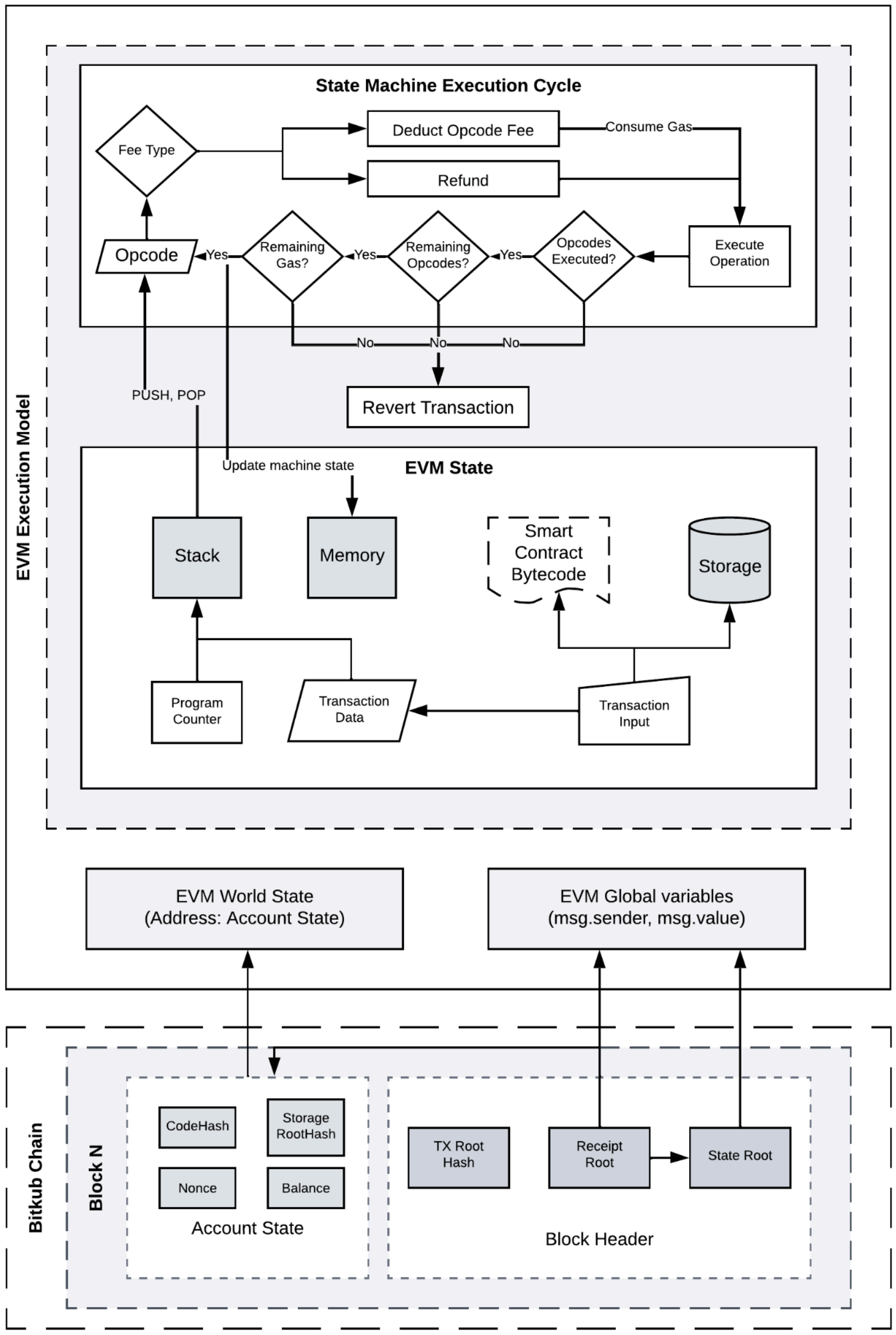


Figure-06: EVM architecture [18]

5.1) EVM Instructions Set

The smart contract code is written in a low-level, stack-based bytecode language called “EVM bytecode”. The bytecode consists of a series of bytes, where each byte represents an operation [13]. There are many types of operations:

5.1.1) Arithmetic operations, opcodes for arithmetic execution, e.g., ADD, MUL, SUB, DIV, MOD, SHA3, etc.

5.1.2) Stack operations, opcodes for stack, memory, and storage management instructions, e.g., POP, PUSH, MLOAD, MSTORE, SLOAD, STORE, etc.

5.1.3) Process flow operations, opcodes for control the execution flow, e.g., STOP, JUMP, PC, JUMPDEST.

5.1.4) System operations, opcodes for the system executing the program, e.g., CREATE, CALL, RETURN, REVERT, DELEGATECALL, STATICCALL*, etc.*

5.1.5) Logic operations, opcodes for comparisons, and bitwise logic, e.g., LT, GT, XOR, OR, EQ, ISZERO, etc.

5.1.6) Environmental operations, opcodes for dealing with the execution environment information, e.g., GAS, ADDRESS, BALANCE, ORIGIN, EXTCODECOPY and etc.

5.1.7) Block operations, opcodes for accessing the information on the current block, e.g., BLOCKHASH, COINBASE, TIMESTAMP, DIFFICULTY, etc.

For more information about an EVM instructions cost [1]

5.2) Execution Cycle

Several essential information is required in the execution model inside the execution environment, including the system state $\boldsymbol{\sigma}$, the remaining gas $\boldsymbol{g}$, substate $\boldsymbol{A}$ (Appendix C), and tuple **I** (Appendix D). The primary function in the

EVM is a state-progression function (**O**) which performs an iterator loop recursively to compute the resultant state ***σ’***, the remaining gas ***g’***, substate ***A’***, and the consequent output ***I***

(10)[1]
$$O\big((\boldsymbol{\sigma}, \boldsymbol{\mu}, A, I)\big) \quad \equiv \quad (\boldsymbol{\sigma}', \boldsymbol{\mu}', A', I)$$

In the normal situation of the execution, the state-progression function (O) should be performed until reaching the normal halting and return the output result of execution.

(11)[1]
$$H(\boldsymbol{\mu}, I) \equiv \begin{cases} H_{\text{RETURN}}(\boldsymbol{\mu}) \text{ if } & w \in \{\text{RETURN}, \text{REVERT}\} \\ () \quad\quad \text{if} & w \in \{\text{STOP}, \text{SELFDESTRUCT}\} \\ \varnothing \quad\quad \text{otherwise} & \end{cases}$$

The normal halting might be triggered by some opcodes such as RETURN, REVERT, STOP, SELFDESTRUCT. The other will be an empty set (∅). However, some unexpected cases could occur, and the execution would be halted and return to the exceptional halting state.

(12)[1]
$$\begin{aligned} Z(\boldsymbol{\sigma}, \boldsymbol{\mu}, A, I) \equiv \quad & \boldsymbol{\mu}_g < C(\boldsymbol{\sigma}, \boldsymbol{\mu}, A, I) \quad \vee \\ & \delta_w = \varnothing \quad \vee \\ & \|\boldsymbol{\mu}_\mathbf{s}\| < \delta_w \quad \vee \\ & (w = \text{JUMP} \ \wedge \ \boldsymbol{\mu}_\mathbf{s}[0] \notin D(I_\mathbf{b})) \quad \vee \\ & (w = \text{JUMPI} \ \wedge \ \boldsymbol{\mu}_\mathbf{s}[1] \neq 0 \ \wedge \\ & \quad \boldsymbol{\mu}_\mathbf{s}[0] \notin D(I_\mathbf{b})) \quad \vee \\ & (w = \text{RETURNDATACOPY} \ \wedge \\ & \quad \boldsymbol{\mu}_\mathbf{s}[1] + \boldsymbol{\mu}_\mathbf{s}[2] > \|\boldsymbol{\mu}_\mathbf{o}\|) \quad \vee \\ & \|\boldsymbol{\mu}_\mathbf{s}\| - \delta_w + \alpha_w > 1024 \quad \vee \\ & (\neg I_\text{w} \ \wedge \ W(w, \boldsymbol{\mu})) \quad \vee \\ & (w = \text{SSTORE} \ \wedge \ \boldsymbol{\mu}_g \leqslant G_{\text{callstipend}}) \end{aligned}$$

where

(13)[1]
$$\begin{aligned} W(w, \boldsymbol{\mu}) \equiv \quad & w \in \{\text{CREATE}, \text{CREATE2}, \text{SSTORE}, \\ & \text{SELFDESTRUCT}\} \ \vee \\ & \text{LOG0} \leq w \ \wedge \ w \leq \text{LOG4} \quad \vee \\ & w = \text{CALL} \ \wedge \ \boldsymbol{\mu}_\mathbf{s}[2] \neq 0 \end{aligned}$$

Many cases can cause the exceptional halting function (Z), such as insufficient gas, invalid instruction, insufficient stack items, JUMP/JUMPI invalid destination, the stack size larger than 1024, and state modification being attempted during a static call, etc.

6. Validation & Finalization

6.1) Finalization

Finalize means permanently added and can not be altered or reverted. In the KUB Chain, a block will be considered to be a finalized block when the block components are valid such as block header, transactions, and uncle block are valid.

6.2) Validation

6.2.1) Block header validation

Block header validation is done by the consensus engine. In KUB Chain, the block header will be validated based on the rules set by Clique consensus as follows:

a) Block timestamp must not be more than the current UNIX time and is not too close to its parent block timestamp.

b) Nonce must be either **nonce to vote on adding a new validator** 0xffffffffffffffff or **nonce to vote on removing a validator** 0x0000000000000000, otherwise will be considered as invalid.

c) Extra data must be contained in both the vanity and signature, ensures that the block doesn't contain any uncles.

d) Difficulty must be either **difficulty for in-turn validator** two or **difficulty for out-of-turn validator** 1. Otherwise, it will be considered invalid.

f) The gas limit must be lower than the maximum gas limit 2^63-1.

g) The gas used must be lower than the block gas limit.

6.2.2) Block body validation

a) Transaction validation

Check whether the transaction root hash in the block header T, is equal to the hash which return from the function C,

(14) $$(T', R') = C(TXs, To)$$

where the TXs is a list of transactions and To is an empty tries, the function C will compute the output as a transaction root hash T', requires T = T'

b) The uncle

In KUB Chain is based on Clique, which require an empty uncle block.

6.2.3) State validation

a) The amount of used gas in the state execution process must be equal to the actual gas used in the block header, otherwise it will be considered to be invalid.
b) Validate the received block's bloom with the one derived from the generated receipts.
c) Check whether the state root in the block header S, is equal to the state root S', which returns from the function D (15),

(15) $$S' = D(E)$$

Where E is a StateDB, and the function D will compute the output as a state root S', requires S = S'

7. Consensus

7.1) Consensus definition

In a centralized consensus system, it is easy to find an agreement because everything relies on the central authority, but there are some concerns, e.g., a single point of failure and unworthy of trust. In contrast with a centralized system, a distributed system such as Blockchain is reliable. However, there is a challenge to make everyone on the network agree on the same data without a centralized authority. Therefore, the consensus is used in a distributed system as a mechanism in order to find an agreement on a single version of the data between each node in the network.

7.2) Consensus Mechanism

The consensus mechanism is an essential component inside the Blockchain which both secures the network and plays on the crypto-economic game theory to incentivize the validators and prevent some kinds of attacks such as the 51% attack. There are many consensus mechanisms such as Proof-of-Work (PoW), Proof-of-Stake (PoS), Proof-of-Authority (PoA), etc.

7.2.1) Proof-of-Stake (PoS)

In the early stages of upgrading the consensus mechanism on KUB Chain, we started with changing the consensus from PoA to PoSA (Proof-of-Stake-Authority) by doing a hard fork called Erawan Hardfork [28].

Since the transitioning of consensus is smoothly running on KUB Chain. We have upgraded the consensus from PoSA to Proof-of-Stake (PoS) called Chaophraya Hard Fork [Ref.111]. The Proof-of-Stake mechanism was designed by Bitkub Blockchain Technology Co., Ltd (BBT) and is suitable for KUB Chain ecosystem. The PoS consensus was designed to reduce the restriction of PoSA that the validator node still needs permission or approval from the existing authority.

The proof of stake on KUB Chain includes the mechanism to random the validators and PoS contract. The consensus uses native KUB coins which stake in PoS contracts (Appendix F) as a signer's power. In every period of selecting validators, the PoS mechanism will randomly target numbers from the total stake of the available validators in the PoS contract. All the numbers will transform into the list of authorized validator addresses and be committed to the PoS contract. An architecture of PoS mechanism on KUB Chain shows in figure below,

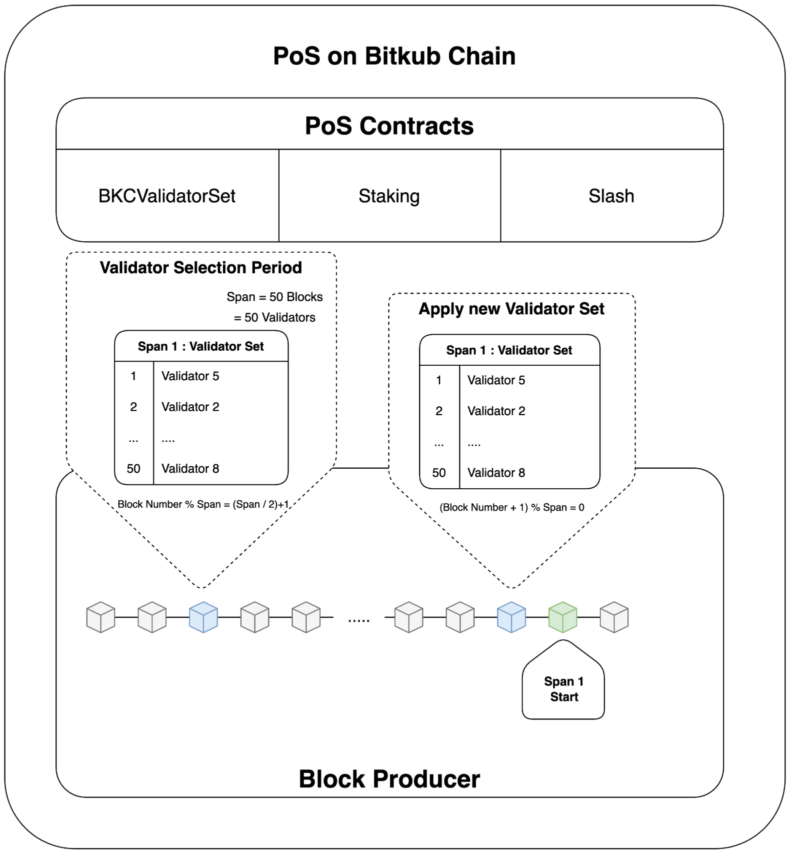
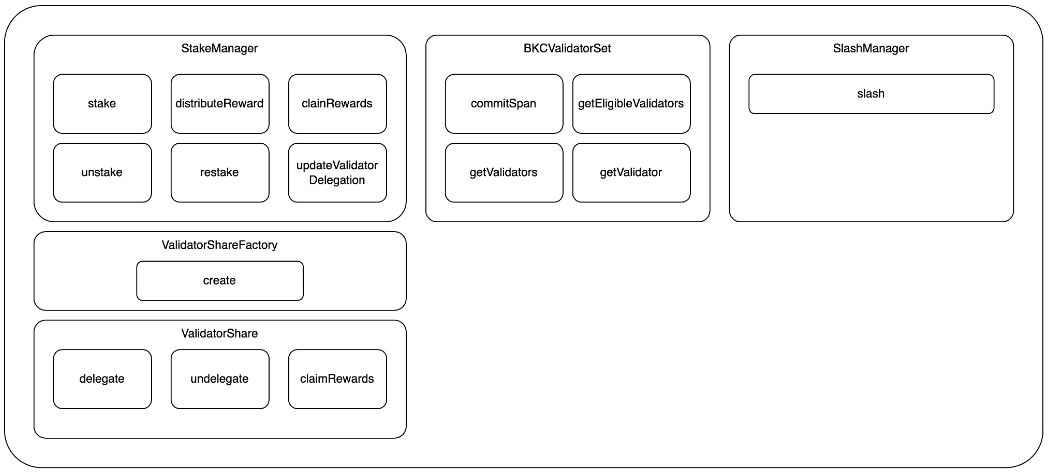


Figure 222: The PoS consensus on KUB Chain

a) PoS Contracts



The PoS contracts are used to operate the PoS mechanism on the KUB Chain. The PoS contracts have been designed in the concept of changeable contract by separating the operation contracts from the storage contract. The PoS contracts consist of the following contract,

i) stakeManagerStorage

The stakeManagerStorage contract contains structs, and variables which are used in the PoS contracts. The stakeManagerStorage also provides the logic to store the states of validators such as validator, delegator, slashing, etc.

ii) stakeManager

For the PoS on KUB Chain, anyone participating can become a validator node via stakeManager contract which is deployed on the KUB Chain. The stakeManage contract has the following features,

- **stake**

  To become a validator node, the user can start their node by calling the stake function with the option for selecting the types of validator and the stake amount. The stake amount is related to the type of validator.

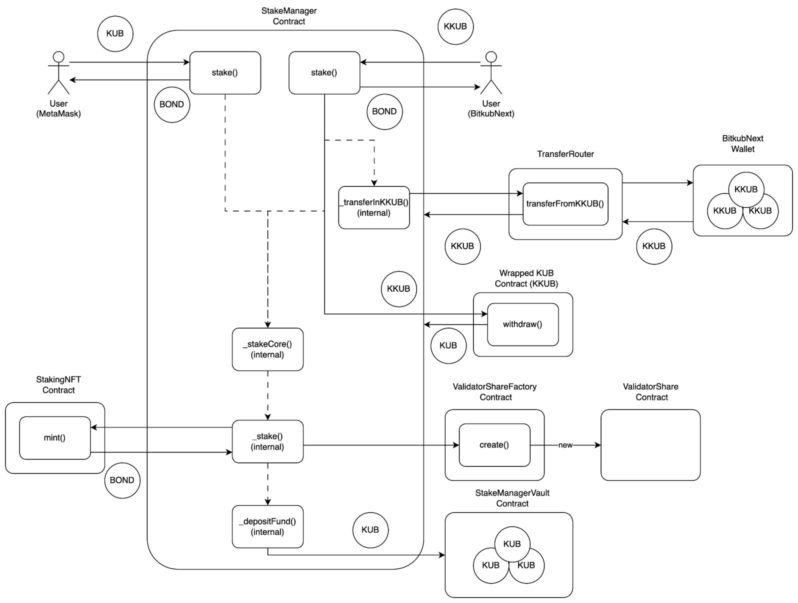


Figure-:StakeManager:Stake

- **claimReward**
  The owner of the validator node can claim all rewards anytime. The reward will be transferred from the stakeManagerValut to the address of the validator node owner.

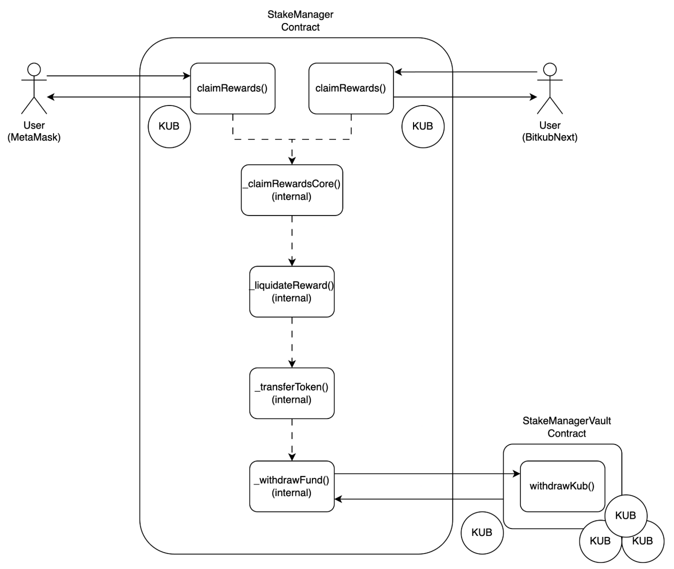

StakeManager Contract
User (MetaMask)
claimRewards()
claimRewards()
User (BitkubNext)
KUB
KUB
_claimRewardsCore() (internal)
_liquidateReward() (internal)
_transferToken() (internal)
_withdrawFund() (internal)
StakeManagerVault Contract
withdrawKub()
KUB
KUB
KUB
KUB

Figure-:StakeManager:ClaimReward

- **undelegate**
  The delegator can decide to leave contributions by calling undelegate method to withdraw their KUB which is delegated in the validatorShare contract and share tokens will be burned equal to the amount you want to withdraw.

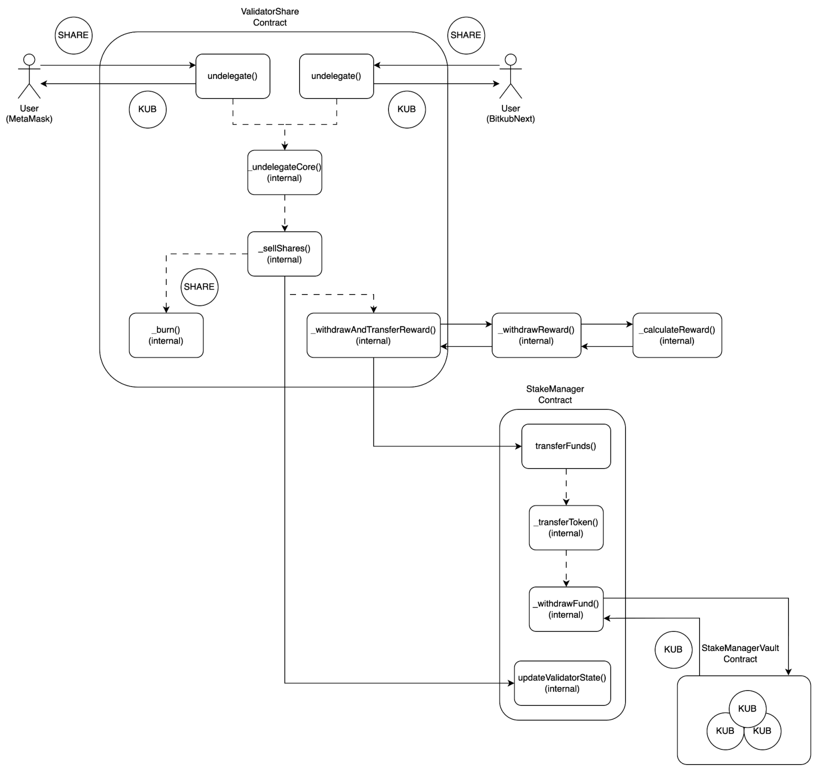

ValidatorShare Contract
SHARE
SHARE
User (MetaMask)
undelegate()
undelegate()
User (BitkubNext)
KUB
KUB
_undelegateCore() (internal)
_sellShares() (internal)
SHARE
_burn() (internal)
_withdrawAndTransferReward() (internal)
_withdrawReward() (internal)
_calculateReward() (internal)
StakeManager Contract
transferFunds()
_transferToken() (internal)
_withdrawFund() (internal)
KUB
StakeManagerVault Contract
updateValidatorState() (internal)
KUB
KUB
KUB

Figure-:ValidatorShare:Undelegate

iii) bkcValidatorSet

The BKCValidatorSet contract is the main contract which is used on the PoS consensus. The contract has interaction with the stakeManager contract that contains the available validators. The BKCValidatorSet contract has the function that allows the PoS consensus to get the list of eligible validators for use in the selecting validators process.

The contract also records the state of the validator list for each span or period of starting the new set of validators.

- commitSpan

  The commitSpan method will operate at the validator set preparation period. The commitSpan contains the bytes of the validator set for the next span. The commitSpan method will be called by system as a system transaction which is zero gas price.

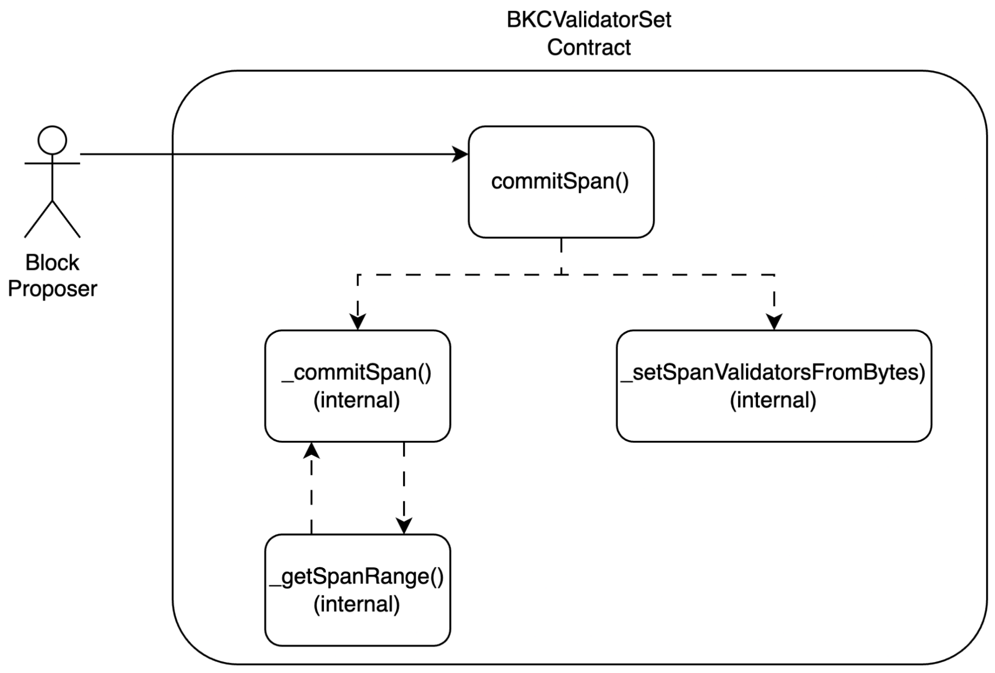


Figure-:BKCValidatorSet:commitSpan

iv) slashManager

The slashManager contract consists of logic for penalizing nodes that are trying to disturb the consensus or unable to propagate a block in a propagation period.

- slash

  The slash method will be called from the official node as a system transaction. The slash method will be operated when the block of the assigned validator does not appear in time. The slash method will deduct the stake from a validator. Those amounts of deduction will be distributed to the official pool as a reward.

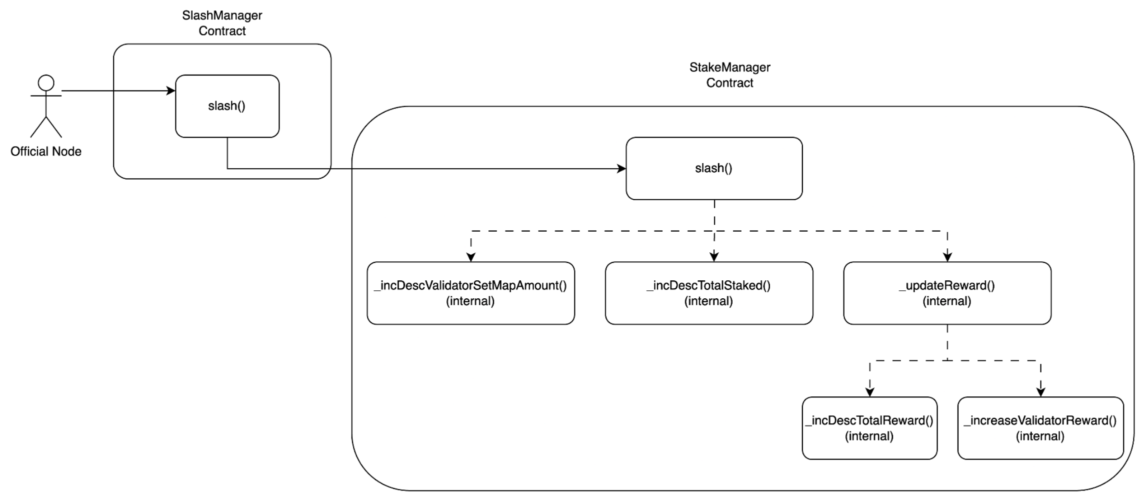

SlashManager Contract
slash()
Official Node
StakeManager Contract
slash()
_incDescValidatorSetMapAmount() (internal)
_incDescTotalStaked() (internal)
_updateReward() (internal)
_incDescTotalReward() (internal)
_increaseValidatorReward() (internal)

Figure-:SlashManager:slash

b) Authority Mechanism

i) Validator Types

After updating the consensus, now anyone can stake their KUB coin which is native coin on the KUB Chain on a staking contract. PoS on KUB Chain has categorized the type of validators to three types, official node, solo node, and pool node.

- Official Node

  KUB Chain introduces the new type of node validator called Official Node. The official node works the same as other validator pool nodes. The official node is operated by the group of authorized partners. The official node has 3 main operations.

  - The official node has to sign and propagate a block when it is in turn signer by using the same algorithm with other validator nodes.
  - The official node has to sign and propagate a block when in turn the signer does not propagate a block in time.
  - The official node has a right permission that can proceed to slash other validator nodes, who do not propagate blocks in time.

- Validator Pool Node
  The user can be the part of the validator node like a pool that allows the user to stake KUB and run their own validator node. The validator pool also has a function that allows other parties to contribute their validator nodes and earn rewards as delegators.

- Solo Node
  The solo node works like the validator pool node but disables the function that allows other parties to do the delegation. Only the validator node owner can take the rewards of running the node.

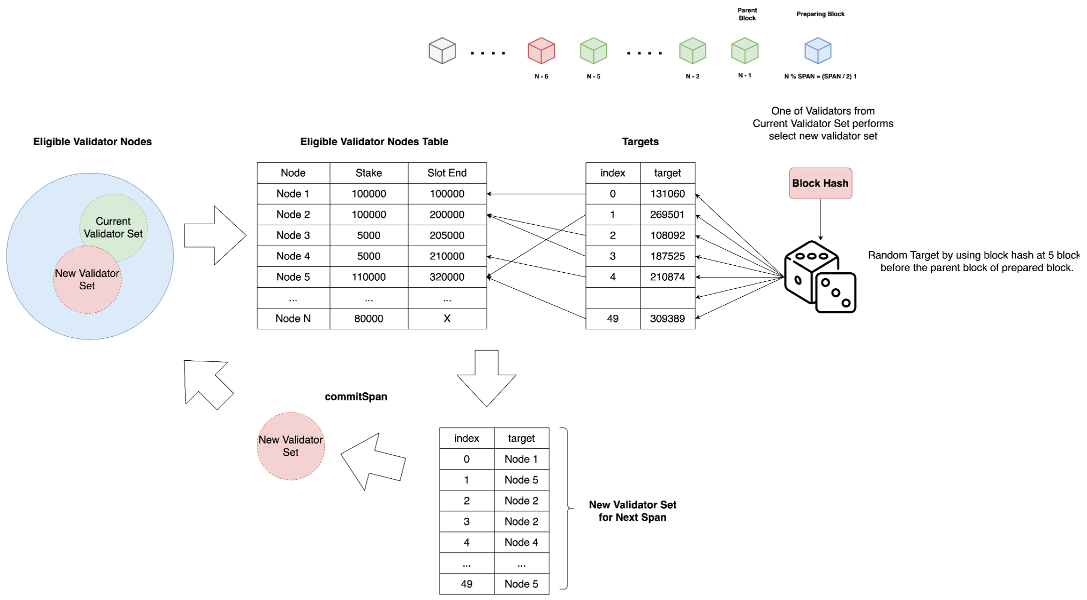
  In Chaophraya Hardfork has limited the maximum number of validator types. The maximum numbers are set in the PoS contract. The numbers of each type shows on the table below,

Validator type	Maximum number of node
Official Node	1
Validator Pool Node	50
Solo Node	100

Table-02: Maximum number of each type of nodes

A number of each node can represent a slot. A slot of each type is first come first serve, that means when node has staked KUB in a staking contract the slot will be allocated. The allocated slot will be released when an existing validator has unstaked their staked KUB from a staking contract.

ii) Validator Selection Mechanism


Parent Block
Preparing Block
N - 6
N - 5
N - 2
N - 1
N % SPAN = (SPAN / 2) 1
One of Validators from Current Validator Set performs select new validator set
Eligible Validator Nodes
Current Validator Set
New Validator Set
Eligible Validator Nodes Table
Targets
Block Hash
Random Target by using block hash at 5 block before the parent block of prepared block.
commitSpan
New Validator Set
New Validator Set for Next Span

Node	Stake	Slot End
Node 1	100000	100000
Node 2	100000	200000
Node 3	5000	205000
Node 4	5000	210000
Node 5	110000	320000
...	...	...
Node N	80000	X

index	target
0	131060
1	269501
2	108092
3	187525
4	210874
49	309389

index	target
0	Node 1
1	Node 5
2	Node 2
3	Node 2
4	Node 4
...	...
49	Node 5

Figure-:Validator Selection Mechanism

The above figure shows the validators selection mechanism will be operated on KUB Chain by the assigned validator. The assigned validator to perform the validator selection process is the validator at 26th of each span. The condition to trigger the validator selection period shows below,

(Block number) % SPAN = (SPAN / 2) + 1

In every selecting new validator period the nodes will perform the following,

- Call getEligibleValidators from BKCValidatorSet contract via KUB Chain EVM. The eligible validators consist of a list of the signer addresses and the total stake KUB of each validator that is on the stakeManager contract.

- Transform the eligible validator list to the slot according to the stake. The slot looks like the Eligible Validator Node Table in Fig.333

- Select the seed hash by using the block hash of the previous six blocks before the preparing block, meaning if the validator selection happens on block 26th then the block hash of block 20 will be chosen as a seed hash. The random function will utilize the seed hash for randomizing the 50 target numbers. The target number will relate to the slot number of each validator in the eligible validator table.

- Change the target number list to the list of validator addresses then transform the validator list to RLP format. The validator addresses list in RLP format will be used in the commitSpan process to update the state of the next validator set to all validator nodes via bkcValidatorSet contract.

iii) Propose a new validator set

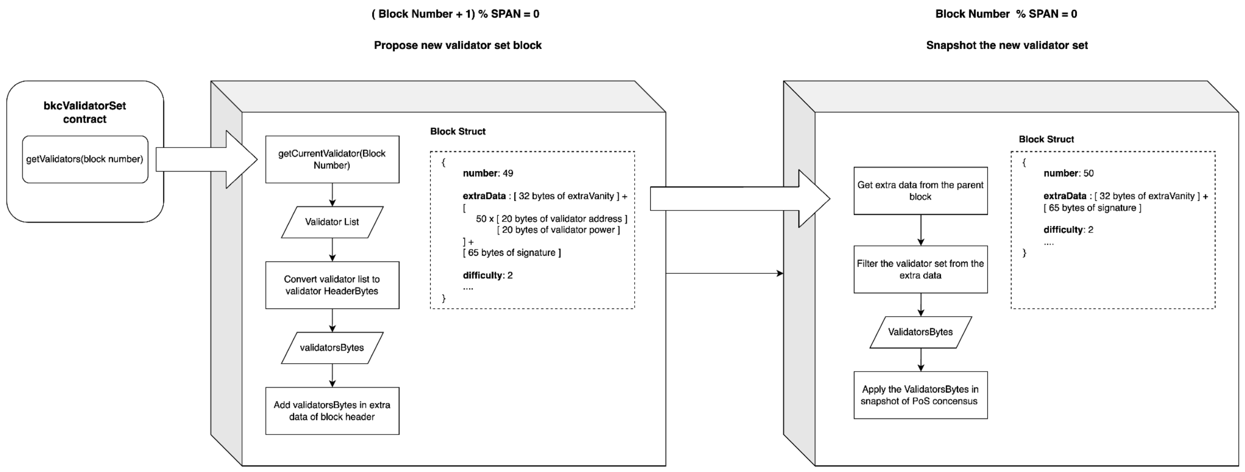

(Block Number + 1) % SPAN = 0
Propose new validator set block
bkcValidatorSet contract
getValidators(block number)
getCurrentValidator(Block Number)
Validator List
Convert validator list to validator HeaderBytes
validatorsBytes
Add validatorsBytes in extra data of block header
Block Struct
{
number: 49
extraData : [32 bytes of extraVanity] + [50 x [20 bytes of validator address] [20 bytes of validator power]] + [65 bytes of signature]
difficulty: 2
....
}
Block Number % SPAN = 0
Snapshot the new validator set
Get extra data from the parent block
Filter the validator set from the extra data
ValidatorsBytes
Apply the ValidatorsBytes in snapshot of PoS concensus
Block Struct
{
number: 50
extraData : [32 bytes of extraVanity] + [65 bytes of signature]
difficulty: 2
....
}

Figure:-Propose a new validator set process

The PoS consensus on KUB Chain has a mechanism to propose a new validator set for the next span. The block of propose the new validator set can be calculated as below,

$$(\text{Block Number} + 1) \% \text{SPAN} = 0$$

Every block 49th of each span the assigned validator has to propose a block with the new validator set by calling getValidators function with current block number from bkcValidatorSet contract. The bytes of the new validator set will be stored in the block extra data and broadcast to KUB Chain network.

At every starting span block, all validator nodes will snapshot the new validator set from the validator bytecode which is attached in the extra data of the previous block.

c) Slash mechanism

Slashing is a mechanism used by a consensus to discourage harmful behaviors from a group of validators who are responsible for mining and validating blocks in the PoS blockchain. Once the consensus has random a list of validators in each span, each block has its miner. If a block is not propagated by the miner, the official node will propagate a block that includes a slashing transaction instead. A slashing transaction makes a call to the smart contracts for penalizing a miner who misses a block. Slashing will be applied to the "Pool Node" and the "Solo Node". Every miner can be slashed once in each span. In other words, every miner who misses one or more block(s) in the same span will only be slashed once at the first block they have missed.

The slash mechanism also provides some compromise for the validators. Once the validator misses their blocks, it considers if they need to be warned or slashed. This mechanism gives chances for each validator to miss those blocks by warning with the "Warn" event on the SlashManager contract. Once a validator has reached the maximum chances it gives, they will be slashed.

In case of being warned by the SlashManager. The "Warn" event will be emitted. SlashManager doesn't make a slashing call to the StakeManager. So your stakes aren't deducted for any reason.

On the other hand, the case of being slashed by SlashManager. The "Slash" event will be emitted with the same arguments as the "Warn" event. SlashManager makes a slashing call to the StakeManager to penalize the validator with no responsibility to the network.

Deciding to warn or slash someone is quite sensitive. The penalties should be balanced to keep the ecosystem validators friendly. The rules make the game, but when it is too hard, no one joins. The mechanism uses a span as a metric for calculating your propagation performance. Once you miss the block, that whole span will be marked as your "bad span". Blocks in the rest of the span that you have missed will be taken over by the official node.

The mechanism counts every time validators miss their blocks. Each validator has its counter. When the counter starts counting from 1 (one), the mechanism gives you around 24 hours to not let the bad span happen again. This 24-hour period is called a slash epoch. The counter keeps counting your bad span(s) before it stops when it reaches the threshold. Once your personal counter reaches the threshold, the SlashManager calls the StakeManager to penalize your stakes. Finally, after you have been slashed, the personal counter will be reset. Next time you miss the block, the counter starts counting from 1 (one) again with the new 24-hour slash epoch.

On the KUB Chain, we have configured the different threshold for each KUB Chain's network. We use 43 spans for the threshold which is around 3 hours for the testnet, and 50 spans for the mainnet which is 3 hours and 20 minutes for the mainnet. We set 345 spans for the slash epoch in both networks of the KUB Chain which is 24 hours approximately.

For your benefit, we do not recommend you disconnect your node from the network for 3 hours in testnet, and 3 hours and 20 minutes in mainnet continuously.

The slashing penalty is the amount of KUB coins that miners will be deducted from their staked value when they are slashed by the official node. A penalty will be proportionally distributed to every participant (delegator) in the official pool provided by BBT and partners. The slashing penalty depends on the miner's node type which will be described next. This makes it fair for every validator to participate in the consensus.

	KUB Chain Testnet	KUB Chain Mainnet
"Solo Node" penalty	10% of stakings	1% of stakings
"Pool Node" penalty	1000 KUB	1000 KUB

The consensus has a period that every validator in the network will wait for a block which is propagated by the inturn miner. Once the period has ended without any block propagation, the official node will propagate a block with these properties

- Block difficulty is set to 1 (0x01) as an outturn signer
- The slash transaction is added to the block transactions. The slash transaction must be signed by the official node account and must call the SlashManager smart contract to begin a slashing procedure.

8. Mining and Reward distribution

8.1) Mining mechanism

KUB Chain constantly emits a new block every five seconds (see more in the block period section). Every validator will create a new block (an empty block) and take the pending transactions from the transaction pool into a new block. The pending transaction will be sequenced by the gas price (GBit), which means the transaction with more gas price will have more chances to be included in the new block.

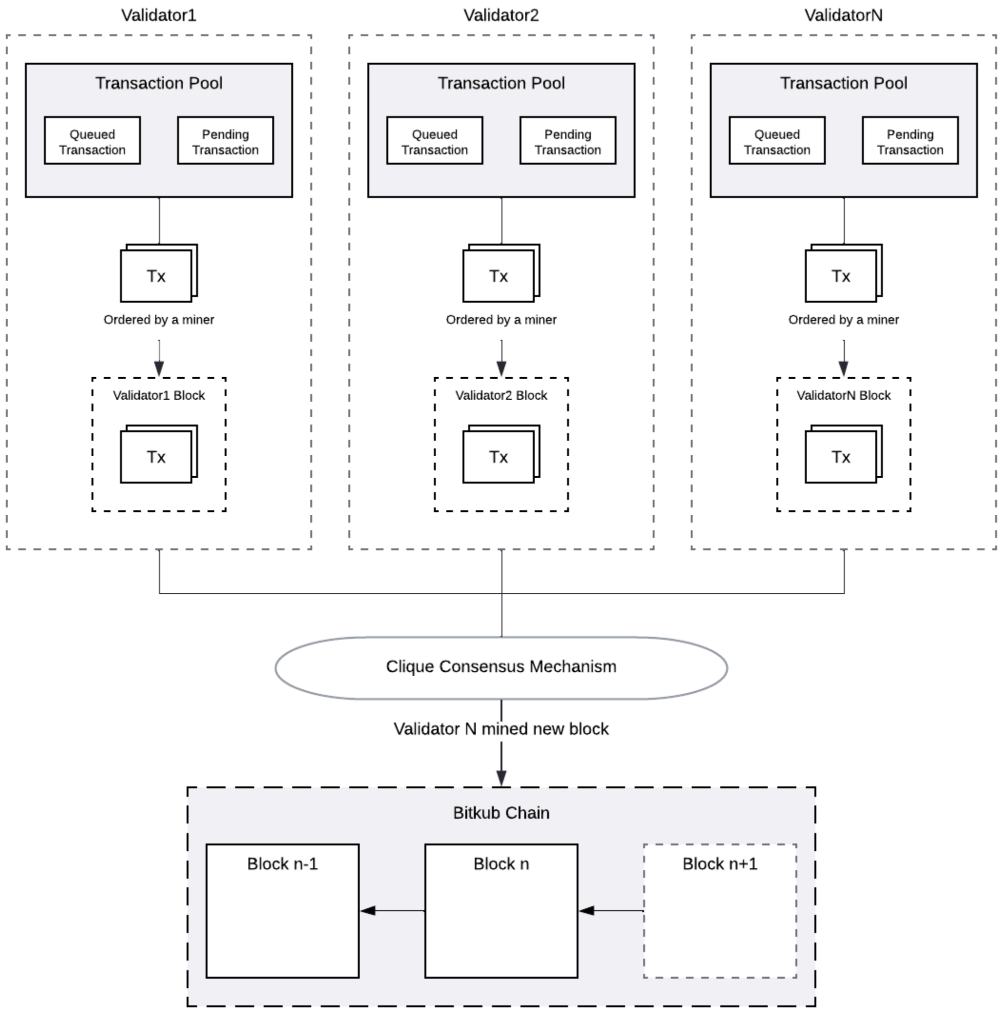


Figure-12: Mining a new block

KUB Chain uses the Clique mechanism to manage the consensus among the group of authorized validators to find a validator to mine a new block. The selected validator will commit, seal and append a new block to the canonical chain and then broadcast it to the other node in the network. The other nodes on the KUB Chain will receive a new block and perform a block validation algorithm including;

a) Check if the previous block referenced is valid and exists.
b) Check that the block timestamp is greater than the timestamp of the previous block, and not lower than the previous block timestamp plus the block period.
c) Check that the block number, transaction root, uncle root, difficulty, and block gas limit are valid.
d) Check if the signature of the validator is valid.
e) Check if the Merkle root of the state is equal to the final state root provided in the block header.

After all is executed and valid, each node will append a new block to the storage.

8.2) Block reward distribution mechanism

The system will incentivize the validator by rewarding the native coin (KUB coin) and the system will collect rewards from the fee of the transactions included in the newly mined block. The total block reward is a summation of the transaction fee in a block, following the equation of block reward.

(18)
$$\sum_{g=G1}^{g=Gn} g$$

g = Gas fee of a transaction
G1 = Gas fee of the first transaction in a block
Gn = Gas fee of the n^{th} transaction in a block

After the reward has been collected, the system will distribute it to the StakeManager contract and update the reward state for the validator who propagates the block. If the block does not have any transactions, the total block reward will be zero, and the system will not distribute a reward to StakeManager by default.

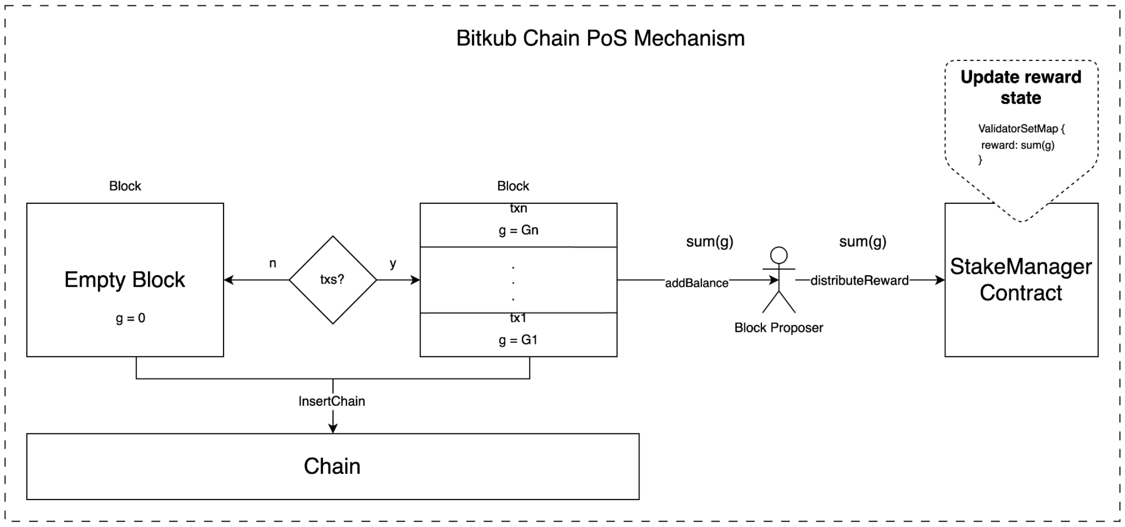


Figure-: Block reward distribution mechanism

9. Block period

Block period or block time is a period that the blockchain will produce a new block. KUB Chain has an average block time of 5 seconds and for this PoS upgrade, it still uses the same. But the PoS improves how it handles the block propagation when there are one or more no-turn signers in the network. No-turn signing happens when a block is mined by a no-turn signer for some reason. This often happens when the in-turn signer can not propagate the block in time. This may cause by disconnecting from the peer-to-peer network, or the node accidentally stopping. With the newly designed slash mechanism, all no-turn signings will be handled in a shorter time. This makes the average block time of KUB Chain with PoS not affected that much by a lot of no-turn signings.

9.1) Performance comparison

9.1.1) Impact from no-turn signers

In PoSA, a list of validators will be assigned to mine a block using a round-robin in each epoch. Each epoch contains 300 blocks. Without the network latency, KUB Chain block time is set to 5 seconds which means the fastest time to reach one epoch should take 1500 seconds $(300 * 5)$. However, the consensus still has time to wait for the in-turn signer to propagate a block to the network which will make a block time delay a little bit. The default waiting time is 5 seconds, if there is no block propagated, the delay will be added to each out-turn signer

Let m be the number of signers in the network. Let w be the wiggle time. The wiggle time is the time that will add on top of each no-turn signer which is a direct variation of m. And let k be a wiggle constant which uses 0.5 second in PoSA. The wiggle time can be calculated as following

$$w = (m / 2 + 1) \cdot k$$

Once the wiggle time is calculated in every node, each node will random the additional delay to be added on top of the default block period before propagating their no-turn blocks. The number which is in the range $[0, w)$ will be added to the delay which means the node with the lowest random delay will have the best chances to mine the block.

Now, let p be the default block time of KUB Chain which is equal to 5 seconds. And let $R(x, y)$ be a random function with an uniform distribution which random number from the range $[x, y)$. The mechanism of delaying the no-turn signers makes the block time increase up to $p + R(0, w)$. To simplify the calculation of the average PoSA block time, we will let z be a no-turn block time.

$$z = p + R(0, w)$$

$$\lim_{n \to w} R(0, n) = w$$

Let e be the size of the epoch in PoSA consensus. Given function $B(n)$ returns the amount of the no-turn blocks in an epoch when n is the amount of the no-turn signers. The function can be described as follows.

$$B(n) = \begin{cases} \frac{e}{m} \cdot n + n, & e \,\%\, m > n \\ \frac{e}{m} \cdot n + (e \,\%\, m), & e \,\%\, m \leq n \end{cases}$$

And with the variable defined above, given the function $\alpha(x)$ is the function that returns the average block time of PoSA when x is the number of the no-turn signers in an epoch. The function can be described as follows.

$$\alpha(x) = ((e - B(x)) \cdot p) + (B(x) \cdot z)$$

PoS has improved the average block time affected by the no-turn signers using the slash mechanism. This makes the KUB Chain block time close to 5 seconds as much as possible. The slash mechanism does the following.

Every signer in the network waits for the in-turn signer to propagate the block. This waiting period is also the same as used in the PoSA which is 5 seconds plus the wiggle time. The wiggle time is a random number from the $[0, k)$ when k is a wiggle constant like the PoSA which equals 0.5. Once the waiting is ended, the consensus gives you another extra delay for the in-turn block which takes about 2 seconds. If the block still has not been propagated, the no-turn signer (also known as the official node) comes up with the block with the difficulty of 1 (0x01). The block from the official node includes the system transactions that penalties the in-turn stakes. The overall progress mentioned above takes 7 seconds to complete which means the block time increases up to 7 seconds. But the mechanism will only allow each signer to be slashed once per span. Thus, in every span, the no-turn block time will only happen depending on the number of unique no-turn signers.

Let s be the size of the span. Let p be a block period (block time) of the blockchain. And let d be the consensus extra delay which is 2 seconds. Given the $R(x, y)$ as the random number function with an uniform distribution which returns the random number from the range $[x, y)$. And given that k is a wiggle constant which is 0.5. Given function $\beta(x)$ returns the average block time in a span when x is the number of unique no-turn signers.

$$\beta(x) = (x * (p + R(0, k) + d) + (s - x) * p) / s$$

$$R(0, k) = k$$

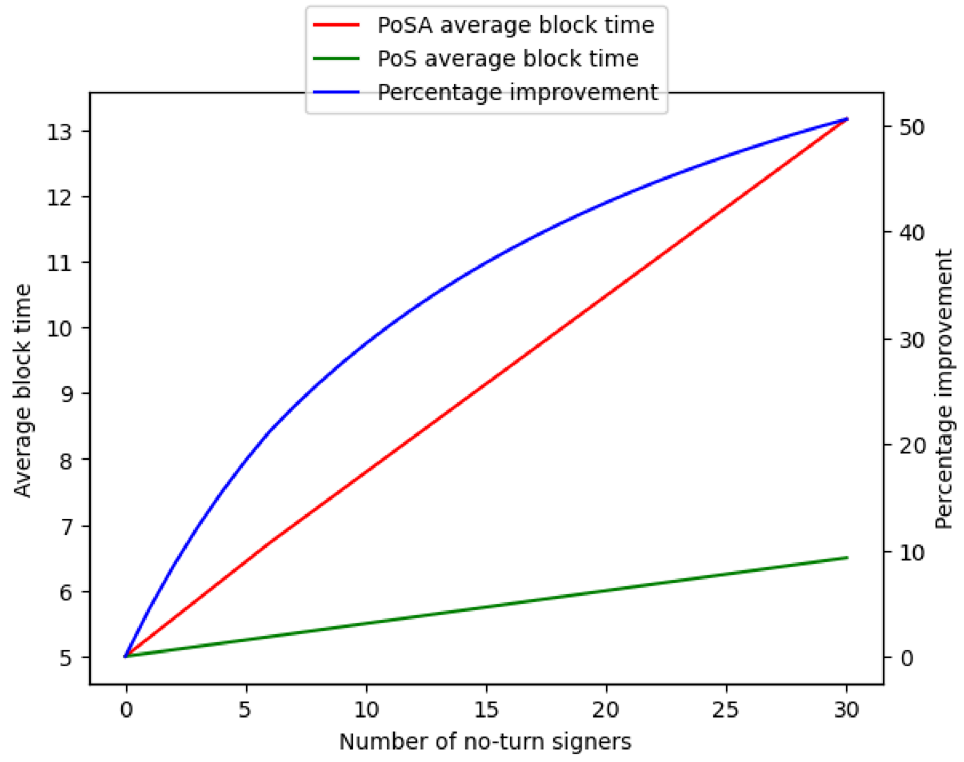


Figure-:Average block time of PoSA and PoS when the network includes no-turn signers.

The diagram above shows how the number of no-turn signers affects the average block time in the PoSA and PoS. This visualization maximizes outputs from the random function $R(0,x)$ which return the value x instead of the random value in the range $[0,x)$, such that $R(0,x) = x$. This shows the maximum block time affected by the random function from both consensuses. With the PoS consensus, the no-turn signers have less impact on KUB Chain average block time than the previous PoSA consensus. This can bring up benefits in long term conditions. Compared with the actual context of having 21 validators. If there are 20 validators down from the network, the block time impacted by no-turn signers in PoS can be up to 40% faster than the PoSA.

9.1.2) Scale of no-turn impact

The scalability of the number of validators is also one of the most important factors that affect the block time of the KUB Chain when there are one or more no-turn signers in the network. In this comparison, we assume that there is only one no-turn signer in the network. Let's see if the network scale can impact the delay time in an inverse relation.

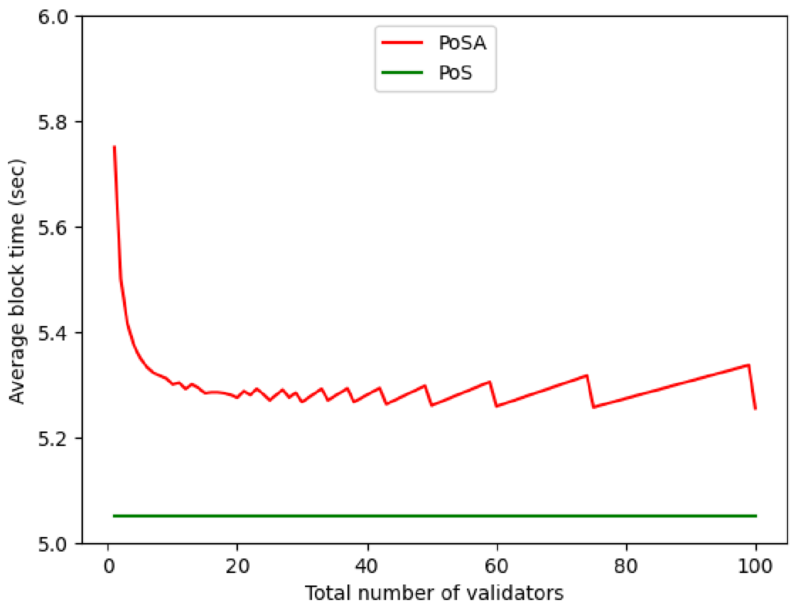


Figure-:Average block time of PoSA and PoS with the increasing number of validators when the network includes a no-turn signer.

As you can see from the diagram above, the PoSA should handle the block time impact better when the network has a large number of validators. But with PoS, the number of validators does not affect the block time when there is a no-turn signer in each span. Note that this test assumes that there is only one no-turn signer in a span or an epoch.

10. Architecture

10.1) Blockchain layer

KUB Chain relies on the internet protocol (IP), which is widely used nowadays. The blockchain protocol builds on top of TCP/IP. It transcends IP and becomes the Internet which can store or send the value called the Internet of value. The Architecture of the KUB Chain can explain as a layer (see Figure-15).

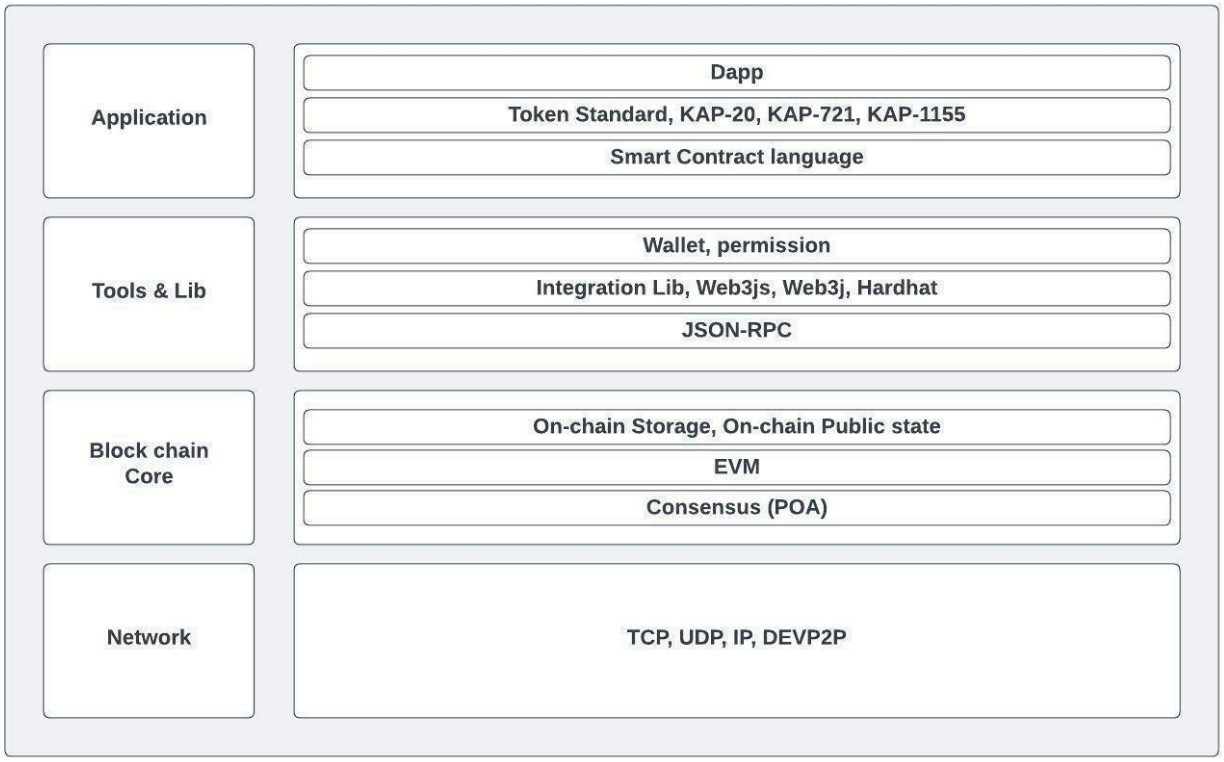


Figure-15: the layer of the chain

10.1.1) Network layer

is the foundation of blockchain because it consists of all components required to build other higher layers. Generally, DEVP2P [27] uses UDP to discover another node and TCP to make peer-to-peer connections.

10.1.2) Blockchain core

is the core component of blockchain. It is the source of consensus. After combining with EVM (Execution module) and on-chain data storage, it will become the concept of the blockchain paradigm (see section 2).

10.1.3) Tools & library

facilitate developers to develop applications on the chain more easily. For example, Web3 can provide ease of programming and connection. This tool connects web applications and the blockchain.

10.1.4) Application

can refer to a programming language, token standards, and decentralized application. The contract is nothing but byte code running on every node verified by the validator nodes.

10.2) Network Architecture

is the network structure of the blockchain connecting peer-to-peer. However, there is more than one type of node to create a blockchain service. Therefore, we need four kinds of nodes (see Figure-17).

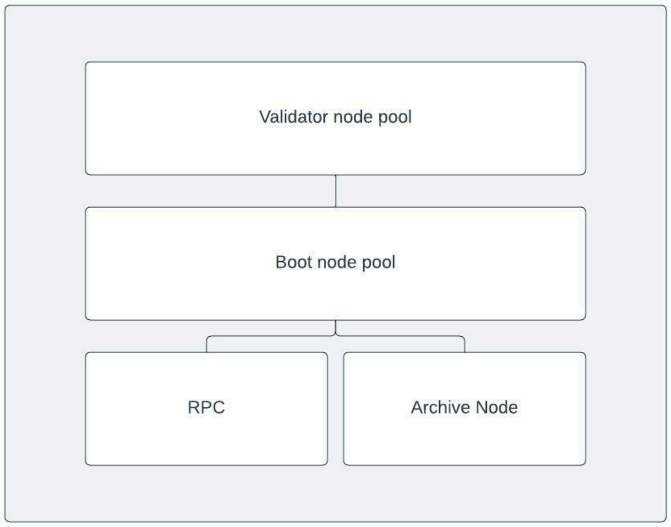


Figure-16: high-level architecture

10.2.1) Validator node pool

is a series of validator nodes. The primary function is to verify transactions and then create the block. This type of node is an essential part of the blockchain system. The chain will stop working if the number of running validator nodes is less than 50% of all validator nodes. Voting from the existing validator node is required to add a new validator node to the pool.

10.2.2) Boot node pool

is a bunch of boot nodes that support the discovered protocol to search other nodes. Boot nodes are like a relayer helping each node to find other nodes on the same network via UDP protocol.

10.2.3) Remote procedure call (RPC) Node

is to communicate between blockchain networks and applications. Everyone in the public domain can run the RPC node independently without permission from authorized parties.

10.2.4) Archive Node

collects all blockchain headers and contents. In a normal RPC node, it keeps only the last 128 blocks. However, the archive node collects the whole on-chain data.

11. The Advantages of KUB Chain

11.1) Blockchain Layer

11.1.1) Block size

The PoS integration on KUB Chain has increased the block size to 65,000,000 gas from 60,000,000. The purpose of increasing the block size in KUB Chain is to support the system transactions that will perform in each stage of the PoS mechanism. Increasing the block size will guarantee each validator still earns the maximum reward at 60,000,000 gas used for each block. The block size of KUB Chain is still more than other chains, such as the Ethereum blockchain, which sets the limit to 30,000,000 gas. The higher limit of gas per block means a higher number of transactions can be included in a block. However, the larger block size will also increase the storage cost and require a higher hardware specification to run a blockchain node.

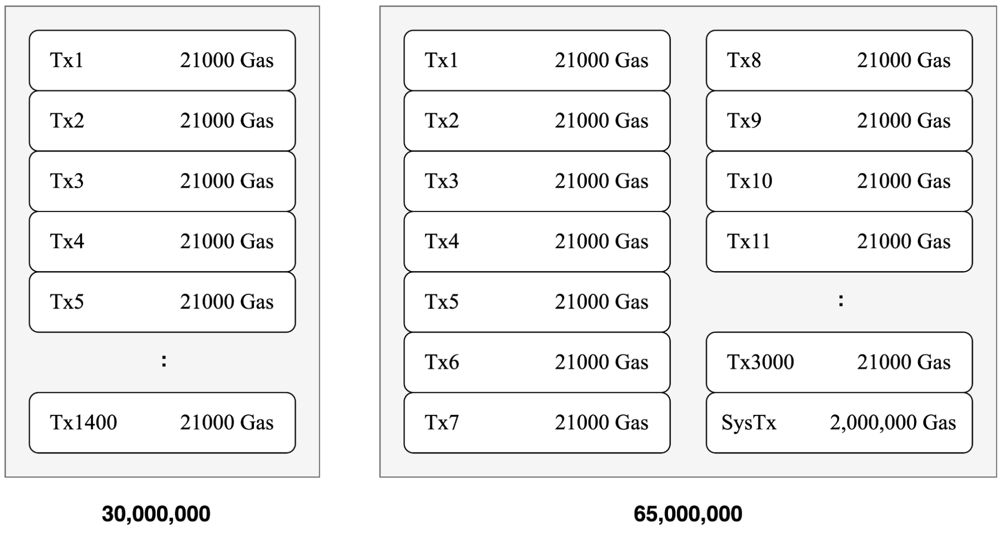


Figure-17: KUB Chain (PoS) block size

11.1.2) Block time

According to chapter 9 (Block period), KUB Chain’s system is fixed to 5 seconds which is faster than the Ethereum network (on average). Furthermore, the time variation is extremely low due to the low latency of KUB Chain’s network. The speed of a blockchain can be calculated from block size and block time in terms of TPB (transactions per block).

11.1.3) Transaction fees

The high transaction fees are one of the pain points on the big primitive blockchains such as Bitcoin or Ethereum. The sender might have to pay fees worth over a dollar to send a token worth a dollar. There are many reasons that cause high transaction fees like this, such as coin price, gas price per unit, and the amount of gas used by a transaction. The coin price is out of control because it depends on the market. But the gas price depends on the crypto-economic model. Ethereum uses the dynamic gas price mechanism based on network congestion. If the network is congested, the gas price and the transaction fees will be higher. However, KUB Chain uses a different mechanism called a dynamic gas price mechanism that will stabilize the transaction fees. KUB Chain sets the gas price to 5 GBit (see chapter 9), which is suitable for many business use cases.

11.1.4) Infrastructure design

KUB Chain introduces the new type of node which performs as other validator nodes called official nodes. The official nodes are operated with the group of authorized partners. The official nodes will be set in the multiple regions to maximize the load, reduce the network latency, and operate instead of other validator nodes.

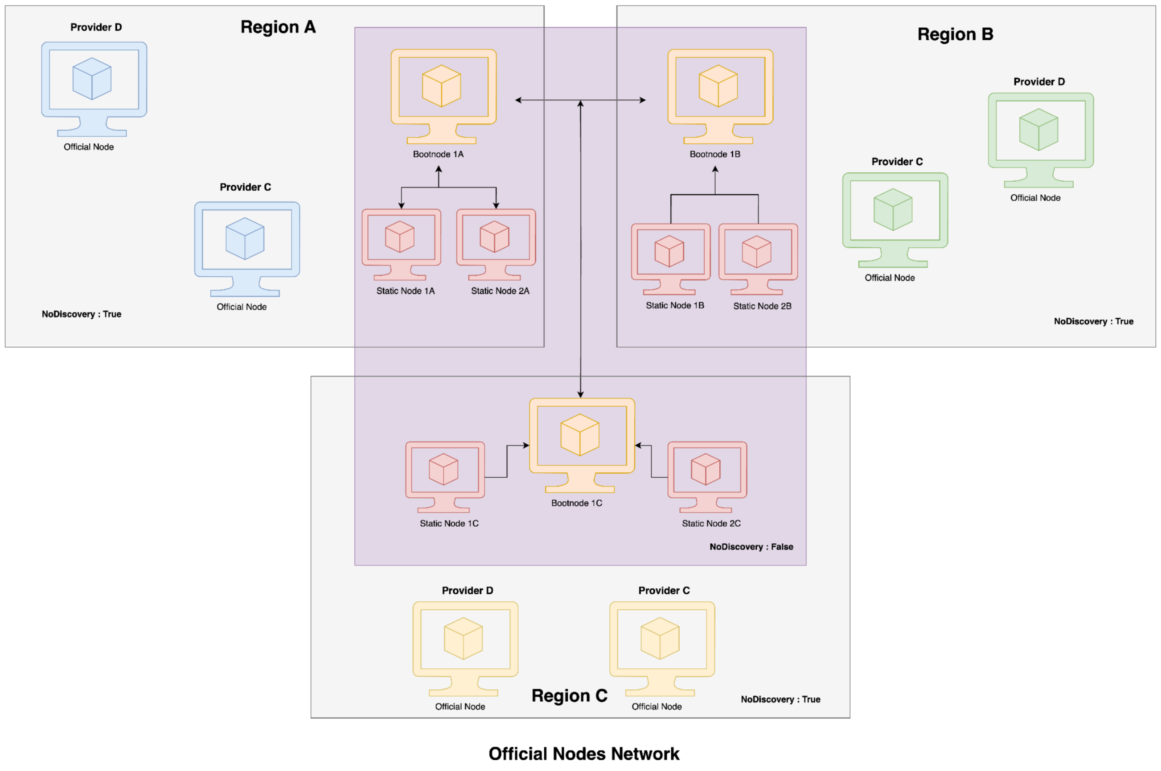


Figure-18: Official nodes network architecture

Figure-18 shows the infrastructure design of KUB Chain. KUB Chain designs the group of official nodes, bootnodes and static nodes in multiple regions and different providers. KUB Chain has provided the routing to connect to the KUB Chain network with the low network latency. A node can link to bootnodes or static nodes in each region as preferred. To start a validator node the owner of a validator node needs to consider the delay on the network to avoid the penalty of the slashing process (section 7c).

11.1.5) The comparison of KUB Chain

Chain	KUB Chain	KUB Chain	Ethereum	Huobi ECO Chain (HECO)
Consensus	Proof-of-Stake (Chaophraya Hardfork)	Proof-of-Stake-Authority (Erawan Hardfork)	Proof-of-Stake	Proof-of-Stake-Authority
Execution Engine	EVM	EVM	EVM	EVM
Smart Contract Language	Solidity, Vyper	Solidity, Vype	Solidity, Vyper	Solidity, Vyper
Transaction Per Block (TPB)	5997	5997	1400	1900
Gas Price	Dynamic-rate (min. 25 gwei)	Dynamic-rate (min. 5 gwei)	Variable-rate	Variable-rate
Block Time (second)	5	5	~15	3
Native Currency	KUB	KUB	ETH	HT

Table-04: The comparison of KUB Chain

11.2) Application Layer

11.2.1) Registration

Register Smart Contract (Appendix I) is used on KUB Chain for registering user's addresses. Moreover, it controls the access to the Bitkub NEXT.

a) Registration levels

i) Level 0 represents the unregistered addresses.
ii) Level 2 represents the primary registered address. This level will unlock the Unwrap KKUB feature.
iii) Level 4 represents the secondary registered address. This level will unlock additional features such as transfer Token or NFT.
iv) Level 16 represents the advanced registered address. This level will unlock the specific smart contract feature.

b) Register procedure

i) Registration

- Register via https://kkub-otp.kubchain.com by using a phone number, and their wallet address will be assigned to level 2
- Register via https://accounts.bitkubnext.com/register by using phone number, email address, and password will be assigned to level 4

Once registered successfully, the backend service will call the **batchSetKycCompleted** function on the smart contract to record the level on the blockchain.

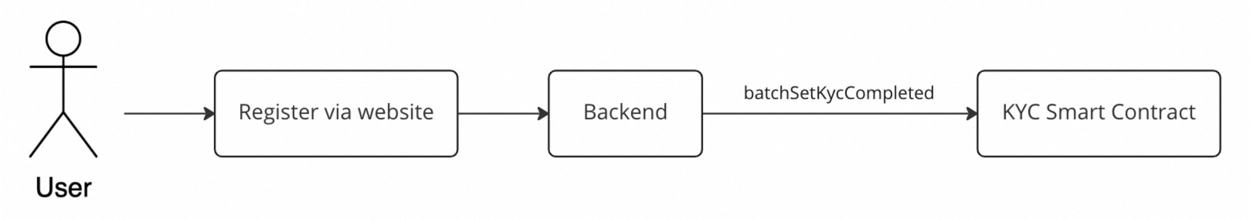


Figure-19: Registration flow

ii) Verify: All official and verified projects must verify the level of user's registration on Bitkub NEXT by calling **kycLevels** function in the smart contract.

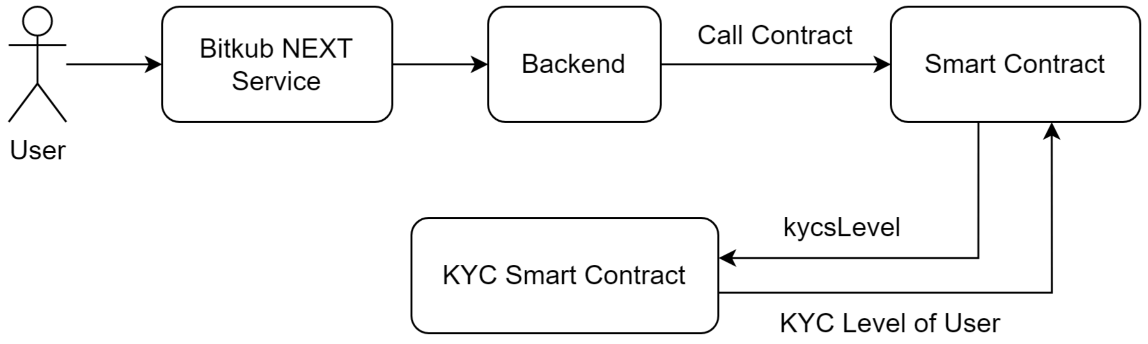


Figure-20: Verification flow

11.2.2) KAP token standard

KUB Chain has its token standards called KAP, which is inspired by the Ethereum Request for Comments (ERC) standard. The standard for fungible tokens is KAP-20 (see Appendix J), and the standard for non-fungible tokens is KAP-721 (see Appendix J). The main difference between ERC-20 and KAP-20 is the adminTransfer function. This standard is useful when the rug pull occurs because an admin of the token can transfer the token back from the attacker's address. Table-05 is the illustration of the difference between ERC-20 and KAP-20.

	ERC-20	KAP-20
Events	Transfer(from, to, value)	Transfer(from, to, tokens)
	Approval(owner, spender, value)	Approval(tokenOwner, spender, tokens)
Functions	totalSupply()	totalSupply()
	balanceOf(account)	balanceOf(account)
	allowance(owner, spender)	allowance(owner, spender)
	approve(spender, amount)	approve(spender, amount)
	transfer(to, amount)	transfer(to, amount)
	transferFrom(from, to, amount)	transferFrom(from, to, amount)
		getOwner()
		batchTransfer(from, to[], value[])
		adminTransfer(from, to, value)

Table-05: The comparison of ERC-20 and KAP-20

a) adminTransfer

adminTransfer is a KAP-specific function that can help the victims by recovering their tokens from the suspicious wallet in the case of a fraudulent. The function shall be used by a relevant issuer of tokens developed on KAP-20 standard.

The improvement of adminTransfer function
The latest upgrade on the adminTransfer function was done to make the technical process more secure and trustworthy. The significant difference between the previous version and the new version is the committee's ability to transfer tokens. Instead of having the full control of the tokens movement, the committee under the new version of adminTransfer has restricted ability to transfer tokens from the suspicious wallet to only one address, i.e. a recovery wallet which shall be specifically created for such alleged fraudulent transaction and managed by the Committee

Smart Contract. This ensures that the suspicious tokens are frozen during the verification process. To clearly illustrate, the distinction between the previous version and the new version is shown in Table-06.

Actions	**adminTransfer (Old)**	**adminTransfer (New)**
The ability to move tokens **from** the suspicious wallet	Yes	Yes
The ability to move tokens **to** any wallet	Yes (According to the relevant policy, the committee will move the tokens to the designated address instructed by the proven victims or the government authorities (as the case may be).)	No (Only to a recovery wallet or related KYC wallet)

Table-06: The distinction between the previous version and the new version

How does adminTransfer works

1. Wallet B is suspected of the token fraud.
2. The victim sends an adminTransfer usage request to the Execution Committee. Alongside the request, the following evidence should be attached: (a) a police report and/or written request from the government authority, (b) on-chain sequences of transaction hashes that correspond to the fraud between Wallet A and Wallet B.
3. The execution committee executes a function on the Committee Smart Contract and passes the related data of the wallets of the suspicious wallet (Wallet B) and the victim's wallet.
   3.1 The system stores the related data (Txid, wallet address, amount) of the wallets of the suspicious wallet (Wallet B) and the victim's wallet.
   3.2 The system automatically creates a new wallet (Recovery Wallet).

4. The Committee Smart Contract executes an adminTransfer function to force transferring the tokens in question from a suspicious wallet (Wallet B).
5. The tokens have been transferred from Wallet B to be locked in the Recovery Wallet.
6. In order to recover the locked tokens, the victim has to create a new wallet (Wallet C) and re-perform a KYC process.
   6.1 The system generates a cryptographic hash from the KYC data using a keccak256 hashing function (this hash will represent the identity of a victim).
   6.2 The system stores a hash on-chain.
7. The victim sends a writ, a court or tribunal order or judgment, and/or an order or request of any government or supervisory authorities (as the case may be) alongside new wallet (Wallet C) address to the KUB Chain operation team (in case of KKUB) or the relevant operation team of the issuer of any other KAP-20 based token.
8. The execution committee executes a function to transfer the locked tokens to a new wallet (Wallet C).
   8.1 The system verifies the previous hash and the current hash of a victim (which must be matched).
   8.2 The Committee Smart Contract executes a transfer function.
9. The locked tokens has been transferred from the Recovery Wallet to the new wallet (Wallet C) or transferred back to Wallet B in the case that the alleged transaction has been proven not a fraud pursuant to a writ, a court or tribunal order or judgment, and/or an order or request of any government or supervisory authorities (as the case may be).

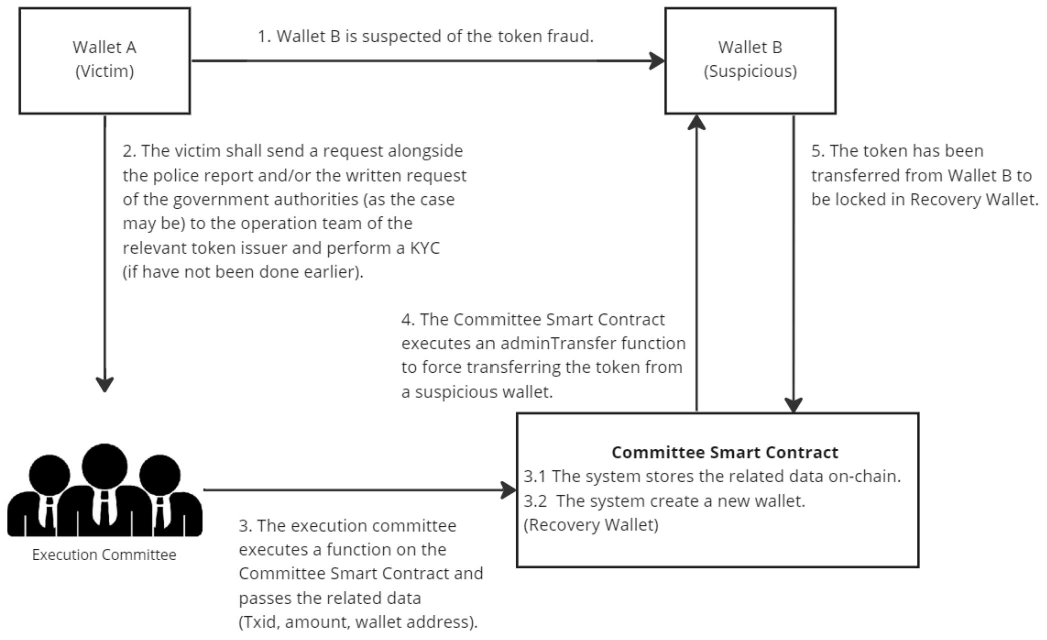

Wallet A
(Victim)
1. Wallet B is suspected of the token fraud.
Wallet B
(Suspicious)
2. The victim shall send a request alongside the police report and/or the written request of the government authorities (as the case may be) to the operation team of the relevant token issuer and perform a KYC (if have not been done earlier).
5. The token has been transferred from Wallet B to be locked in Recovery Wallet.
4. The Committee Smart Contract executes an adminTransfer function to force transferring the token from a suspicious wallet.
Committee Smart Contract
3.1 The system stores the related data on-chain.
3.2 The system create a new wallet. (Recovery Wallet)
Execution Committee
3. The execution committee executes a function on the Committee Smart Contract and passes the related data (Txid, amount, wallet address).

Figure-21: adminTransfer Process (1)

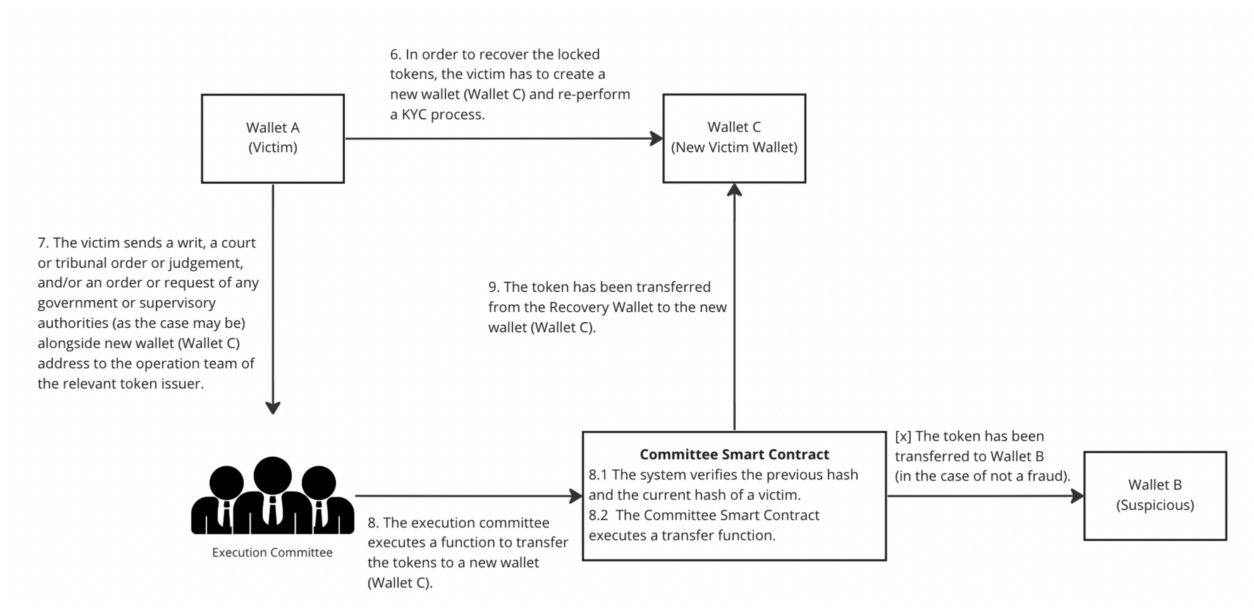

6. In order to recover the locked tokens, the victim has to create a new wallet (Wallet C) and re-perform a KYC process.
Wallet A
(Victim)
Wallet C
(New Victim Wallet)
7. The victim sends a writ, a court or tribunal order or judgement, and/or an order or request of any government or supervisory authorities (as the case may be) alongside new wallet (Wallet C) address to the operation team of the relevant token issuer.
9. The token has been transferred from the Recovery Wallet to the new wallet (Wallet C).
Committee Smart Contract
8.1 The system verifies the previous hash and the current hash of a victim.
8.2 The Committee Smart Contract executes a transfer function.
[x] The token has been transferred to Wallet B (in the case of not a fraud).
Wallet B
(Suspicious)
Execution Committee
8. The execution committee executes a function to transfer the tokens to a new wallet (Wallet C).

Figure-22: adminTransfer Process (2)

Note: If any person entitled to the locked tokens pursuant to a writ, a court or tribunal order or judgement, and/or an order or request of any government or supervisory authorities is deceased or unable to perform the KYC by himself/herself, e.g. a disabled person or incompetent person etc, the locked tokens shall be released to or for the benefit of the entitled person or, in case of the dead, his/her administrator only when (i) the aforementioned evidence of such person's entitlement and a court order appointing an administrator, a curator or a guardian (as the case may be) of such entitled person are present to the KUB Chain team (in case of KKUB) or the relevant issuer of any other KAP-20 based token; and (ii) such administrator, curator or guardian (as the case may be) conducts KYC. In such case, the operation team of the KUB Chain (in case of KKUB) or the relevant issuer of such other KAP-20 based token will revise the hash of the entitled person's original wallet (with traceable changes) and requests such administrator, curator or guardian (as the case may be) of such entitled person to create a new wallet and conduct KYC under his/her name such that the locked tokens of the entitled person can be transferred to the new wallet created by such administrator, curator or guardian (as the case may be) of the entitled person.

b) **batchTransfer**

The batch transfer is a KAP-specific function for transferring tokens to multiple wallets at the same time.

How does batchTransfer works

1. Collect multiple token transfer transaction data
2. Batch them together into a transaction
3. Send a batchTransfer transaction
4. Token transferred to the multiple wallets

The advantages of batchTransfer

1. Cost efficiency: Paying gas only once instead of multiple times.
2. Time reduction: Send and wait only a transaction instead of wait multiple transactions

   For example: sending a token to the 100 wallets without the batchTransfer function you have to send a token 100 times, make 100 transactions, spend enormously gas. On the other hand, utilizing a batchTransfer function can dramatically save your gas by sending just a single transaction and paying gas only once.

c) **getOwner**
Get a KYC wallet address of an owner of the smart contract

11.2.3) KKUB

is a token pegged to the KUB coin and conforms to the KAP-20 token standard. This process can easily convert KUB into KKUB through a process known as “wrapping” by depositing KUB coins to the smart contract. Then KKUB will be minted in the same amount as the deposit amount. or convert KKUB to KUB through a process known as “unwrapping” by withdrawing the KKUB; it will burn the KKUB and send the KUB back to the user. The advantage of KKUB (the KAP-20 version of the KUB coin) is the ability to integrate with Decentralized-Finance (De-Fi) or Decentralized Application (DApp), which is programmable through the smart contract. The smart contract’s code was written in solidity language (Appendix H).

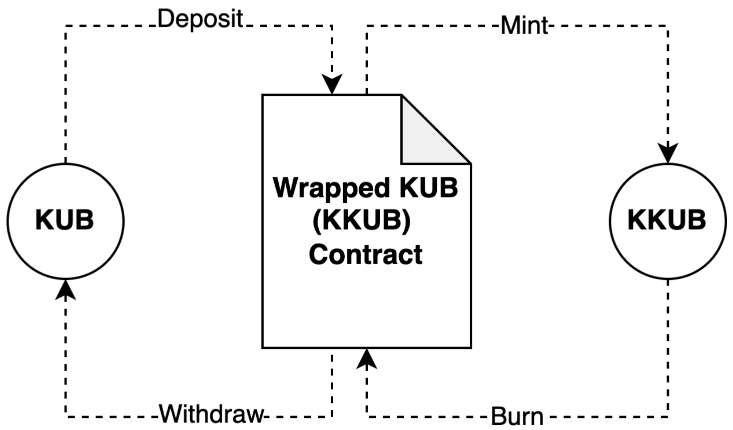


Figure-23: Wrap & Unwrap process of KKUB

11.2.4) HyperBlock

is an algorithm to aggregate transactions from many wallet addresses on the KUB Chain network by making a call to a smart contract (must contain a batchTransfer function). For example, if A wants to send a token to B, C, D, and E, A will send only one transaction as a batch transfer (function on smart contract) instant of sending four separate transactions. This process can increase the speed of transactions by over **1,000** transactions per second (TPS). HyperBlock consists of four parts:

a) Backend Service

is responsible for creating a backed transaction pool where an available thread runs on a background process and matches them with the admin pool addresses. The transactions matched with a specific admin address will send out to the smart contract address. The process can be explained below (Figure-24).

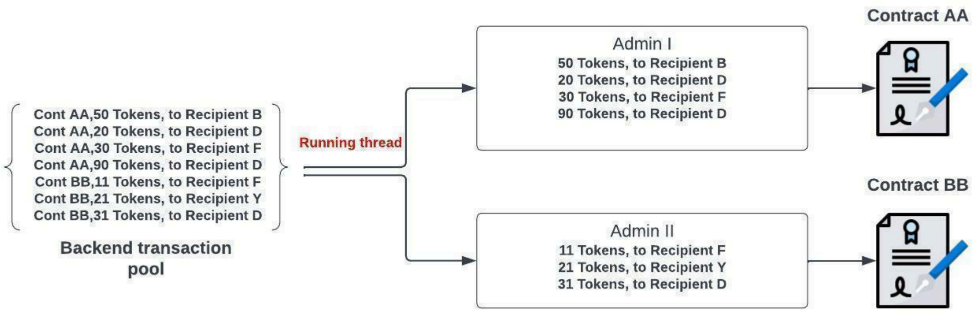

Cont AA,50 Tokens, to Recipient B
Cont AA,20 Tokens, to Recipient D
Cont AA,30 Tokens, to Recipient F
Cont AA,90 Tokens, to Recipient D
Cont BB,11 Tokens, to Recipient F
Cont BB,21 Tokens, to Recipient Y
Cont BB,31 Tokens, to Recipient D
Backend transaction pool
Running thread
Admin I
50 Tokens, to Recipient B
20 Tokens, to Recipient D
30 Tokens, to Recipient F
90 Tokens, to Recipient D
Admin II
11 Tokens, to Recipient F
21 Tokens, to Recipient Y
31 Tokens, to Recipient D
Contract AA
Contract BB

Figure-24 HyperBlock (backend service flow)

b) Gas Tank

is wallet addresses. Those addresses will pay the transaction fees (gas) for the users in KUB's official project. Therefore, the users do not need to store KUB coins in their wallets as long as they are under the KUB ecosystem (see Appendix J) such as Bitkub NEXT. In addition, the users will not be affected by the delay caused by changing the gasPrice because the Gas tank will always guarantee a reasonable gasPrice rate to maintain constant transaction speed.

c) Smart Contract

According to the KAP token standard (see section 11.2.2), batchTransfer function (Appendix H) is called by the admin from the pool to send a batch transaction. Without this function, Hyperblock will not work. All developers working on the KUB Chain's official projects must implement the KAP Token standard.

d) Fail-Safe mechanism

Normally, when a transaction fails to execute on the blockchain, the sender needs to manually re-sent it by themselves. However, KUB Chain has provided the mechanism to automatically re-sent the failed transaction to maximize the user experience. The Fail-Safe mechanism works by using an off-chain engine to detect the failed transaction in the transaction pool and trying to diagnose the root cause of the failure, and then trying to re-sent it. For example, the users might send a transaction with a low gas price relative to the determined gas price by the blockchain, which will cause the failure. In this case, the Fail-Safe mechanism will automatically increase the gas price and re-sent.

To summarize, HyperBlock is an engine that combines different modules and various techniques to maximize the ability to send the transaction to the KUB Chain. The **backend module** (build-up off-chain) is responsible for orchestrating the list of transactions requested by the users and then serving them to the **smart contract module** (build-up on-chain) as a batch of transactions. This technique practically improves the number of transactions (token transfers) to more than one thousand per second. Moreover, the **gas tank module** (build-up off-chain) will facilitate user interaction with the KUB Chain without paying the transaction fees by themselves (only KUB Chain official projects) to maximize the user experience and eliminate the barrier to using the KUB Chain. The overview of system flow can be shown in Figure-25.

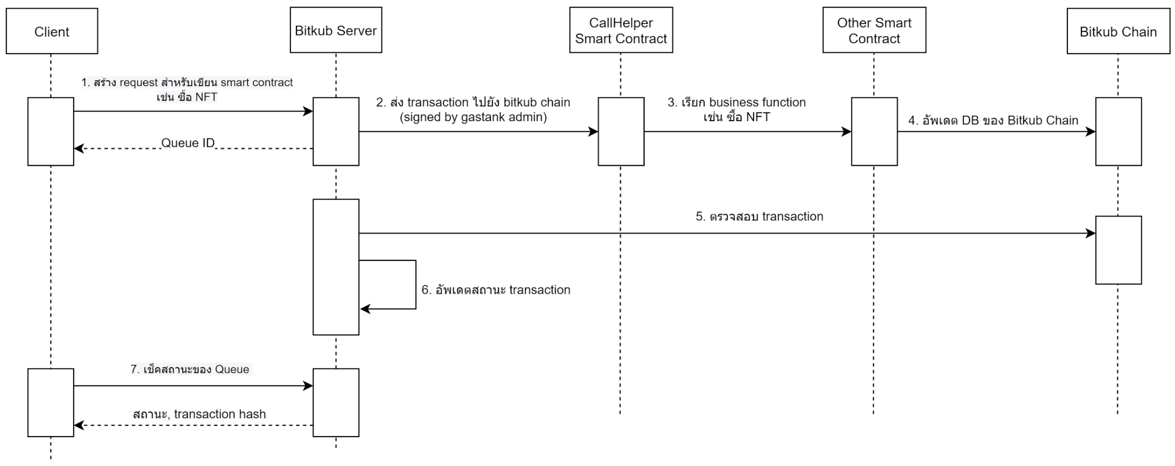

Client
Bitkub Server
CallHelper Smart Contract
Other Smart Contract
Bitkub Chain
1. สร้าง request สำหรับเขียน smart contract เช่น ซื้อ NFT
Queue ID
2. ส่ง transaction ไปยัง bitkub chain (signed by gastank admin)
3. เรียก business function เช่น ซื้อ NFT
4. อัพเดต DB ของ Bitkub Chain
5. ตรวจสอบ transaction
6. อัพเดตสถานะ transaction
7. เช็คสถานะของ Queue
สถานะ, transaction hash

Figure-25: HyperBlock Overview

12. Future Technical Goals

12.1) Scalability

12.1.1) Layer1 Scaling

Scalability is a major challenge in blockchain systems when the number of nodes and transactions increases. Thus, a Layer1 scaling technique becomes the primary step to solve this challenge and increase the block size.

a) Increasing block size

To scale up the limit of transaction KUB Chain plans to increase the block gas limit to more than 60,000,000 gas in the future. However, increasing block size is one of the Layer1 scaling techniques which helps enlarger block sizes and accommodate more transactions per block. However, this technique is simple, but it may come with many risks.

i) Increased block size will lead to the need to expand the storage size of the node as well as to upgrade the disk type to be more efficient. As a result, the cost of running the node increases.

ii) Increased block synchronization time As the number of data increases, it takes longer to download and verify the historical data.

iii) Increased the number of uncle block or chain reorganization as the increase in block size results in longer block execution time and longer broadcast block time to other nodes.

iv) Increased possibility to attack, such as an attack in the form of Denial-of-Service (DDoS). The larger the attackers could send the block size, the more malicious transactions. The attack may cause the network to halt.

b) Sharding

Sharding is one of the layer1 scaling solutions developed and introduced by Vitalik Buterin [15] Sharding is a technique that upgrades to improve the scalability and capacity of a blockchain network by splitting a blockchain into smaller partitions to increase the capability to spread the load and process more transactions. This solution will help the system improve efficiency and be able to handle more than a thousand transactions per second.

This sharding concept divides a chain into many sub-chains known as "shard chains" each shard comprises its data and consists of a unique set of validators called "committees". Each committee will be assigned to verify each block broadcasted on a different set, which it could run altogether in parallel. Once all blocks are executed and validated, validators will attest to the fact and sign a signature to verify that the block is valid. Hence, it added speed and accuracy to the verification process.

There are two types of transaction validation on Sharding; a) the validity of the processing (computation) and b) the availability of the data (data availability).

i) Validating computation

There are two methods of validating computation: Fraud proofs and ZK-SNARKs.

- **Fraud-proofs**
  By default, a system will accept the result of computation but still leaves open the opportunity for someone else with a staked deposit to make a challenge when they find or want to argue that the processing is invalid. However, they can challenge it by depositing tokens as collateral for proof.

- **ZK-SNARKs**
  use the principle of cryptographic proof to perform computation and prove that all processing results are correct, but this method is considerably more complex and complicated to develop compared to fraud-proof.

ii) Validating Data availability

Verifying data's existence is more difficult when compared to verifying processing validity. It is impossible to distinguish who was right and wrong makes it impossible to have a working fraud-proof scheme for data availability. Thus, It cannot use the same techniques such as fraud-proof or ZK-SNARKs but instead have to use the data availability sampling technique. Hence, instead of extracting all the information that we gathered for verification. We will use a random method to check some chucks of the data instead to improve slow performance. [23]

12.1.2) Layer2 Scaling

Layer2 Scaling is a solution designed to help increase scalability for handling more transactions off-chain. By executing it outside (off-chain) of the main chain (Layer1), it can be done by many techniques such as Roll-up, Validium, or Plasma.

a) Rollups

i) Optimistic Rollups: Optimistic Rollups is a technique in which transactions are processed at Layer2, then bundled into a series of transactions (batch), and then the data is posted (roll-up) to Layer 1. Optimistic Rollup uses a technique called fraud-proof to verify the authenticity and correctness of transactions. Fraud-proof will have the disadvantage of having a window of time to allow anyone to prove and challenge the frauds. The current standard delay was seven days.

ii) ZK-Rollups: ZK-Rollup: Zero Knowledge-Rollup is a similar concept to "Optimistic Rollups" but uses a different cryptographic technique called "Zero-Knowledge Proof" to validate transactions. Before it gets posted to Layer1, this technique can prove transactions without the need for a challenging period (7-day delay) like "Optimistic Rollups".

However, this solution may increase the level of difficulty and create complexity in the development process compared to the "Optimistic Roll Ups" technique.

b) Plasma

Plasma is a layer2 scaling solution that is built on top of the root chain (parent chain). The transaction from the parent chain will be off-loaded and executed by the Plasma chain (child chain) while maintaining the validity of the transaction state by using the fraud proof mechanism [24]. The child chain has to commit the child chain state periodically by submitting the Merkle roots of the state of the child chain to the Plasma contract deployed on-chain. The users can enter the plasma chain by bridging their token from the parent chain to the child chain via bridge smart contract and can exit the child chain by withdrawing the token back to the parent chain via bridge. However, withdrawing the token to Layer1 needs a delay time for the fraud proof which is normally set to 7 days.

c) Validium

Validiums is a Layer2 scaling solution that processes and stores the state off-chain only the proof that is stored on-chain (Layer1). Validium is like ZK-Rollups but the data availability of the Validium is stored off-chain rather than on-chain. Validiums can process ~9,000 transactions (or more) per second [24] higher than any other Layer2 solutions because of the off-chain operation. However, storing and executing data off-chain has a high risk and requires trust of the operator. To mitigate the risk validium uses the “Validity proofs” that can be computed and generated by using both ZK-SNARKs and ZK-STARKs to prove that the off-chain state is valid and submit it to the on-chain (Layer1) as a state commitment and finalization.

12.2) KUB SDK (Software Development Kit)

The developer is an essential actor in the decentralized network. However, creating or developing a decentralized application requires specific technical knowledge and tools. KUB Chain wants to encourage the developer to build on our ecosystem by introducing the KUB Chain software development kit (SDK), a development tool. The SDK supports developers in developing a project. Currently, we already publish the KUB Chain Javascript SDK [25] any blockchain developer can easily install by just using the npm or yarn. However, our team is developing the SDK in the different programming languages to maximize the development tools to develop the DApp on KUB Chain.

12.4) On-chain static page

Hosting static pages on the blockchain combines the encoding technique and the on-chain state together. Blockchain is used as a data layer to store the static page detail and a specific smart contract is used as an encoder and decoder module to encode the static page source code to the specific format of data and then pack the data and store on-chain. The on-chain static page will be distributed among the blockchain node making the static page available for everyone with high availability (HA) and publicly accessible by everyone in the world without censorship from the centralized authority (censorship-resistance).

12.5) Verifying standard bytecode

However, many smart contracts had been attacked and the asset in the contract because there is a bug in the code even though they are audited by the auditors. One of the reasons that causes the bug is the additional logic and functions that implemented add-on from the standard function. To prevent or reduce this kind of vulnerability in the KUB Chain ecosystem, we introduced the "Verifying Standard Bytecode" feature to verify the inheritance of the standard from the smart contract by comparing the bytecode of the contract with the bytecode of the standard contract.

13. Conclusion

KUB Chain was created to democratize the opportunity for everyone to access a new economy (digital asset economy) that gives the right to innovate, design, and develop under a decentralized infrastructure, and encourages the decentralization and ownership of digital assets. In addition, KUB Chain technically derived some of the technology from Ethereum, resulting in the same standards, such as EVM compatibility, making it easier for developers to create and develop applications on top of KUB Chain. Moreover, the faster block times, lower fees, and a modern consensus system makes users send faster transactions and have a better user experience.

References

[1] Ethereum Yellow Paper: a formal specification of Ethereum, a programmable blockchain
[2] Yonatan Sompolinsky and Aviv Zohar. Accelerating bitcoin’s transaction processing. Fast money grows on trees, not chains, 2013.
[3] Ethereum Whitepaper https://ethereum.org/en/whitepaper/
[4] Ethereum Development Documentation https://ethereum.org/en/developers/docs/
[5] How does Ethereum work, anyway
https://www.preethikasireddy.com/post/how-does-ethereum-work-anyway
[6] Merkle tree and Ethereum objects
https://medium.com/coinmonks/detailed-explanation-of-ethereum-yellow-paper-merkle-tree-and-ethereum-objects-d85edf5051b8
[7] Merkling in Ethereum https://blog.ethereum.org/2015/11/15/merkling-in-ethereum/
[8] Modified Merkle Patricia Trie — How Ethereum saves a state
https://medium.com/codechain/modified-merkle-patricia-trie-how-ethereum-saves-a-state-e6d7555078dd
[9] Proof of Authority Explained
https://academy.binance.com/en/articles/proof-of-authority-explained
[10] EIP-225: Clique proof-of-authority consensus protocol
https://eips.ethereum.org/EIPS/eip-225
[11] Proof-of-stake (PoS) definition
https://ethereum.org/en/developers/docs/consensus-mechanisms/pos/
[12] Layer2 Scaling https://ethereum.org/en/developers/docs/scaling/
[13] Diving Into The Ethereum Virtual Machine
https://blog.qtum.org/diving-into-the-ethereum-vm-6e8d5d2f3c30
[14] Ethereum State Trie Architecture Explained
https://medium.com/@eiki1212/ethereum-state-trie-architecture-explained-a30237009d4e
[15] Why sharding is great: demystifying the technical properties
https://vitalik.ca/general/2021/04/07/sharding.html
[16] The Limits to Blockchain Scalability https://vitalik.ca/general/2021/05/23/scaling.html
[17] Why Proof-of-Work Is Not Viable in the Long-Term
https://medium.com/logos-network/why-proof-of-work-is-not-viable-in-the-long-term-dd96d2775e99
[18] The Ethereum Virtual Machine https://faun.pub/the-ethereum-virtual-machine-d70dfa5f045b
[19] Mastering Ethereum https://cypherpunks-core.github.io/ethereumbook/
[20] Ethereum EVM illustrated
https://takenobu-hs.github.io/downloads/ethereum_evm_illustrated.pdf
[21] Go-Ethereum https://github.com/ethereum/go-ethereum
[22] The Central Limit Theorem and Means
https://www.statisticshowto.com/probability-and-statistics/normal-distributions/central-limit-theorem-definition-examples/

[23] An explanation of the sharding + DAS proposal https://hackmd.io/@vbuterin/sharding_proposal#ELI5-data-availability-sampling
[24] Scaling https://ethereum.org/en/developers/docs/scaling/
[25] KUB Chain JS SDK https://www.npmjs.com/package/@bitkub-blockchain/sdk
[26] Ethash
https://ethereum.org/th/developers/docs/consensus-mechanisms/pow/mining-algorithms/ethash
[27] Devp2p https://github.com/ethereum/devp2p
[28] BKC Erawan Hardfork https://github.com/kub/bkc/releases

Appendix A. Fee Schedule

The fee schedule G is a tuple of scalar values corresponding to the relative costs, in gas, of a number of abstract operations that a transaction may affect.

Name	**Value**	**Description**
G_{zero}	0	Nothing paid for operations of the set W_{zero}.
$G_{jumpdest}$	1	Amount of gas to pay for a JUMPDEST operation.
G_{base}	2	Amount of gas to pay for operations of the set W_{base}.
$G_{verylow}$	3	Amount of gas to pay for operations of the set $W_{verylow}$.
G_{low}	5	Amount of gas to pay for operations of the set W_{low}.
G_{mid}	8	Amount of gas to pay for operations of the set W_{mid}.
G_{high}	10	Amount of gas to pay for operations of the set W_{high}.
$G_{extcode}$	700	Amount of gas to pay for operations of the set $W_{extcode}$.
$G_{balance}$	700	Amount of gas to pay for a BALANCE operation.
G_{sload}	800	Paid for an SLOAD operation.
G_{sset}	20000	Paid for an SSTORE operation when the storage value is set to non-zero from zero.
G_{sreset}	5000	Paid for an SSTORE operation when the storage value's zeroness remains unchanged or is set to zero.
R_{sclear}	15000	Refund given (added into refund counter) when the storage value is set to zero from non-zero.
$R_{selfdestruct}$	24000	Refund given (added into refund counter) for self-destructing an account.
$G_{selfdestruct}$	5000	Amount of gas to pay for a SELFDESTRUCT operation.
G_{create}	32000	Paid for a CREATE operation.
$G_{codedeposit}$	200	Paid per byte for a CREATE operation to succeed in placing code into state.
G_{call}	700	Paid for a CALL operation.
$G_{callvalue}$	9000	Paid for a non-zero value transfer as part of the CALL operation.
$G_{callstipend}$	2300	A stipend for the called contract subtracted from $G_{callvalue}$ for a non-zero value transfer.
$G_{newaccount}$	25000	Paid for a CALL or SELFDESTRUCT operation which creates an account.
G_{exp}	10	Partial payment for an EXP operation.
$G_{expbyte}$	50	Partial payment when multiplied by the number of bytes in the exponent for the EXP operation.
G_{memory}	3	Paid for every additional word when expanding memory.
$G_{txcreate}$	32000	Paid by all contract-creating transactions after the Homestead transition.
$G_{txdatazero}$	4	Paid for every zero byte of data or code for a transaction.

$G_{txdatanonzero}$	16	Paid for every non-zero byte of data or code for a transaction.
$G_{transaction}$	21000	Paid for every transaction.
G_{log}	375	Partial payment for a LOG operation.
$G_{logdata}$	8	Paid for each byte in a LOG operation’s data.
$G_{logtopic}$	375	Paid for each topic of a LOG operation.
$G_{keccak256}$	30	Paid for each KECCAK256 operation.
$G_{keccak256word}$	6	Paid for each word (rounded up) for input data to a KECCAK256 operation.
G_{copy}	3	Partial payment for *COPY operations, multiplied by words copied, rounded up.
$G_{blockhash}$	20	Payment for each BLOCKHASH operation.
$G_{quaddivisor}$	20	The quadratic coefficient of the input sizes of the exponentiation-over-modulo precompiled contract.

Appendix B: Virtual Machine Specification

$$C(\boldsymbol{\sigma}, \boldsymbol{\mu}, A, I) \equiv C_{\text{mem}}(\boldsymbol{\mu}'_{\text{i}}) - C_{\text{mem}}(\boldsymbol{\mu}_{\text{i}}) + \begin{cases} C_{\text{SSTORE}}(\boldsymbol{\sigma}, \boldsymbol{\mu}, A, I) & \text{if} \quad w = \text{SSTORE} \\ G_{\text{exp}} & \text{if} \quad w = \text{EXP} \wedge \boldsymbol{\mu}_{\mathbf{s}}[1] = 0 \\ G_{\text{exp}} + G_{\text{expbyte}} \times (1 + \lfloor \log_{256}(\boldsymbol{\mu}_{\mathbf{s}}[1]) \rfloor) & \text{if} \quad w = \text{EXP} \wedge \boldsymbol{\mu}_{\mathbf{s}}[1] > 0 \\ G_{\text{verylow}} + G_{\text{copy}} \times \lceil \boldsymbol{\mu}_{\mathbf{s}}[2] \div 32 \rceil & \text{if} \quad w \in W_{\text{copy}} \\ C_{\text{aaccess}}(\boldsymbol{\mu}_{\mathbf{s}}[0] \bmod 2^{160}, A) + G_{\text{copy}} \times \lceil \boldsymbol{\mu}_{\mathbf{s}}[3] \div 32 \rceil & \text{if} \quad w = \text{EXTCODECOPY} \\ C_{\text{aaccess}}(\boldsymbol{\mu}_{\mathbf{s}}[0] \bmod 2^{160}, A) & \text{if} \quad w \in W_{\text{extaccount}} \\ G_{\text{log}} + G_{\text{logdata}} \times \boldsymbol{\mu}_{\mathbf{s}}[1] & \text{if} \quad w = \text{LOG0} \\ G_{\text{log}} + G_{\text{logdata}} \times \boldsymbol{\mu}_{\mathbf{s}}[1] + G_{\text{logtopic}} & \text{if} \quad w = \text{LOG1} \\ G_{\text{log}} + G_{\text{logdata}} \times \boldsymbol{\mu}_{\mathbf{s}}[1] + 2G_{\text{logtopic}} & \text{if} \quad w = \text{LOG2} \\ G_{\text{log}} + G_{\text{logdata}} \times \boldsymbol{\mu}_{\mathbf{s}}[1] + 3G_{\text{logtopic}} & \text{if} \quad w = \text{LOG3} \\ G_{\text{log}} + G_{\text{logdata}} \times \boldsymbol{\mu}_{\mathbf{s}}[1] + 4G_{\text{logtopic}} & \text{if} \quad w = \text{LOG4} \\ C_{\text{CALL}}(\boldsymbol{\sigma}, \boldsymbol{\mu}, A) & \text{if} \quad w \in W_{\text{call}} \\ C_{\text{SELFDESTRUCT}}(\boldsymbol{\sigma}, \boldsymbol{\mu}) & \text{if} \quad w = \text{SELFDESTRUCT} \\ G_{\text{create}} & \text{if} \quad w = \text{CREATE} \\ G_{\text{create}} + G_{\text{keccak256word}} \times \lceil \boldsymbol{\mu}_{\mathbf{s}}[2] \div 32 \rceil & \text{if} \quad w = \text{CREATE2} \\ G_{\text{keccak256}} + G_{\text{keccak256word}} \times \lceil \boldsymbol{\mu}_{\mathbf{s}}[1] \div 32 \rceil & \text{if} \quad w = \text{KECCAK256} \\ G_{\text{jumpdest}} & \text{if} \quad w = \text{JUMPDEST} \\ C_{\text{SLOAD}}(\boldsymbol{\mu}, A, I) & \text{if} \quad w = \text{SLOAD} \\ G_{\text{zero}} & \text{if} \quad w \in W_{\text{zero}} \\ G_{\text{base}} & \text{if} \quad w \in W_{\text{base}} \\ G_{\text{verylow}} & \text{if} \quad w \in W_{\text{verylow}} \\ G_{\text{low}} & \text{if} \quad w \in W_{\text{low}} \\ G_{\text{mid}} & \text{if} \quad w \in W_{\text{mid}} \\ G_{\text{high}} & \text{if} \quad w \in W_{\text{high}} \\ G_{\text{blockhash}} & \text{if} \quad w = \text{BLOCKHASH} \end{cases}$$

$$w \equiv \begin{cases} I_{\mathbf{b}}[\boldsymbol{\mu}_{\text{pc}}] & \text{if} \quad \boldsymbol{\mu}_{\text{pc}} < \lVert I_{\mathbf{b}} \rVert \\ \text{STOP} & \text{otherwise} \end{cases}$$

$$C_{\text{mem}}(a) \equiv G_{\text{memory}} \cdot a + \left\lfloor \frac{a^2}{512} \right\rfloor$$

$$C_{\text{aaccess}}(x, A) \equiv \begin{cases} G_{\text{warmaccess}} & \text{if} \quad x \in A_{\mathbf{a}} \\ G_{\text{coldaccountaccess}} & \text{otherwise} \end{cases}$$

with C_{CALL}, $C_{SELFDESTRUCT}$, C_{SLOAD} and C_{SSTORE} as specified in the appropriate section below. We define the following subsets of instructions:

W_{zero} = {STOP, RETURN, REVERT}

W_{base} = {ADDRESS, ORIGIN, CALLER, CALLVALUE, CALLDATASIZE, CODESIZE, GASPRICE, COINBASE,TIMESTAMP, NUMBER, DIFFICULTY, GASLIMIT, CHAINID, RETURNDATASIZE, POP, PC, MSIZE, GAS}

$W_{verylow}$ = {ADD, SUB, NOT, LT, GT, SLT, SGT, EQ, ISZERO, AND, OR, XOR, BYTE, SHL, SHR, SAR,CALLDATALOAD, MLOAD, MSTORE, MSTORE8, PUSH*, DUP*, SWAP*}

W_{low} = {MUL, DIV, SDIV, MOD, SMOD, SIGNEXTEND, SELFBALANCE}

W_{mid} = {ADDMOD, MULMOD, JUMP}

W_{high} = {JUMPI}

W_{copy} = {CALLDATACOPY, CODECOPY, RETURNDATACOPY}

W_{call} = {CALL, CALLCODE, DELEGATECALL, STATICCALL}

$W_{extaccount}$ = {BALANCE, EXTCODESIZE, EXTCODEHASH}

Note the memory cost component, given as the product of G_{memory} and the maximum of 0 & the ceiling of the number of words in size that the memory must be over the current number of words, μi in order that all accesses reference valid memory whether for read or write. Such accesses must be for non-zero number of bytes. Referencing a zero length range (e.g. by attempting to pass it as the input range to a CALL) does not require memory
to be extended to the beginning of the range. μ0i is defined as this new maximum number of words of active memory; special-cases are given where these two are not equal.

Note also that C_{mem} is the memory cost function (the expansion function being the difference between the cost before and after). It is a polynomial, with the higher-order coefficient divided and floored, and thus linear up to 724B of memory used, after which it costs substantially more.

Appendix C: Substrate

Substrate A, is a set of information which accumulated during the transaction execution process. The information is a tuple that include self-destruct set (S), the log series (L), the set of touched accounts (T), the refund balance (R), the set of accessed account addresses (C) and the set of accessed storage keys (K).

$$\boxed{A = \text{Tuple}(S, L, T, R, C, K)}$$

Appendix D: The execution environment information

List of the execution environment information refer as the tuple **I**, include:

- Address of the account that owns the code that is executing (A)
- Address of the sender of the transaction that originated this execution (O)
- Address of the account that caused the code to execute which could be different from the original sender (S)
- Gas price of the transaction that originated this execution (P)
- Input data for the execution (D)
- Value (in Wei) passed to this account as part of the current execution (V)
- Machine code to be executed (B)
- Block header of the current block (H)
- Depth of the present message call or contract creation stack (E)
- The permission to modify the state. (W)

$$\boxed{I = \text{Tuple}(A, O, S, P, D, V, B, H. E. W)}$$

Appendix E: Staking Constant Variables

MINIMUM_STAKE = 250,000 KUB

Appendix F: PoS Smart Contracts

1. StakeManagerStorage

```
 // * Copyright 2023 Bitkub Blockchain Technology Co., Ltd. - All Rights Reserved.
// * This code is proprietary.
// * Unauthorized copying, modification or distribution of this code, via any
medium, is strictly prohibited without express permission.

pragma solidity ^0.8.0;

abstract contract Committee {
    address public committee;

    event CommitteeSet(
        address indexed oldCommittee,
        address indexed newCommittee,
        address indexed caller
    );

    function _onlyCommittee() internal view {
        require(
            msg.sender == committee,
            "Committee: restricted only committee"
        );
    }

    modifier onlyCommittee() {
        _onlyCommittee();
        _;
    }

    constructor(address committee_) {
        committee = committee_;
    }

    function setCommittee(address _committee) public virtual onlyCommittee {
        emit CommitteeSet(committee, _committee, msg.sender);
        committee = _committee;
    }
}

abstract contract StakeManagerConstants {
    uint16 public constant MAX_RATE = 10000;
    uint16 public constant MAX_COMMISSION_RATE = 5000;
    uint16 public constant MAX_INFRA_COMMISSION_RATE = 5000;
    uint256 public constant INCORRECT_VALIDATOR_ID = type(uint256).max;
}

enum LiquidateMode {
    All,
    ValidatorReward,
    ValidatorCommission,
    DelegatorCommission
}

enum TransferTokenMode {
    None,
    Staked,
    Reward
}

enum ChangeMode {
    Increase,
    Decrease
}

enum Status {
    Uninitialized,
    Active,
    Unstaked
}

enum SlashErrorCode {
    Uninitialized,
    StakeAmountNotEnough,
    OfficialPool,
    NotPool,
    InactivatedPool
}

struct Validator {
    uint128 amount;
    uint128 delegatedAmount;
    uint128 reward;
    uint128 delegatorsReward;
    uint128 infraCommissionAmount;
    uint128 validatorCommissionAmount;
    uint128 delegatorCommissionAmount;
    uint128 minDeposit;
    address signer;
    address validatorShareContract;
    Status status;
    uint16 infraCommissionRate;
    uint16 commissionRate;
}

struct MinimalValidator {
    address signer;
    uint256 power;
}

interface IStakeManagerStorage {
    function stakeManager() external view returns (address);

    function nftContract() external view returns (address);

    function validatorShareFactory() external view returns (address);

    function slashManager() external view returns (address);

    function soloLimit() external view returns (uint256);

    function soloAmount() external view returns (uint256);

    function poolLimit() external view returns (uint256);

    function poolAmount() external view returns (uint256);

    function officialLimit() external view returns (uint256);

    function officialAmount() external view returns (uint256);

    function defaultInfraCommissionRate() external view returns (uint256);

    function totalStaked() external view returns (uint256);

    function totalRewards() external view returns (uint256);

    function totalRewardsLiquidated() external view returns (uint256);

    function soloSlashRate() external view returns (uint256);

    function poolSlashAmount() external view returns (uint256);
    function unallocatedReward() external view returns (uint256);

    function officialPool() external view returns (address);

    function getNewOfficialPoolValidBlock() external view returns (uint256);

    function getOldNewOfficialSignerAndValidBlock()
        external
        view
        returns (address, address, uint256);

    function changeOfficialSigner(
        address newOfficialSigner_,
        string memory checksummedNewOfficialSigner_
    ) external;

    function setStakeManager(address stakeManager_) external;

    function setValidatorShareFactory(address validatorShareFactory_) external;

    function setSlashManager(address slashManager_) external;

    function setSoloLimit(uint256 val_) external;

    function incDescSoloLimit(uint256 val_, ChangeMode mode_) external;

    function setSoloAmount(uint256 val_) external;

    function incDescSoloAmount(uint256 val_, ChangeMode mode_) external;

    function setPoolLimit(uint256 val_) external;

    function incDescPoolLimit(uint256 val_, ChangeMode mode_) external;

    function setPoolAmount(uint256 val_) external;

    function incDescPoolAmount(uint256 val_, ChangeMode mode_) external;

    function setOfficialLimit(uint256 val_) external;

    function incDescOfficialLimit(uint256 val_, ChangeMode mode_) external;

    function setOfficialAmount(uint256 val_) external;
    function incDescOfficialAmount(uint256 val_, ChangeMode mode_) external;

    function setDefaultInfraCommissionRate(uint256 val_) external;

    function incDescDefaultInfraCommissionRate(
        uint256 val_,
        ChangeMode mode_
    ) external;

    function setTotalStaked(uint256 val_) external;

    function incDescTotalStaked(uint256 val_, ChangeMode mode_) external;

    function setTotalRewards(uint256 val_) external;

    function incDescTotalRewards(uint256 val_, ChangeMode mode_) external;

    function setTotalRewardsLiquidated(uint256 val_) external;

    function incDescTotalRewardsLiquidated(
        uint256 val_,
        ChangeMode mode_
    ) external;

    function setPoolSlashAmount(uint256 val_) external;

    function incDescPoolSlashAmount(uint256 val_, ChangeMode mode_) external;

    function setUnallocatedReward(uint256 val_) external;

    function incDescUnallocatedReward(uint256 val_, ChangeMode mode_) external;

    function addValidator(
        address key_,
        Validator memory val_
    ) external returns (uint256);

    function setValidatorIds(
        address key_,
        uint256[] memory validatorIds_
    ) external;

    function setValidatorInfo(address key_, Validator memory val_) external;

    function setValidatorInfoByIndex(
        uint256 index_,
        Validator memory val_
    ) external;

    function setValidatorInfoMinDeposit(address key_, uint256 val_) external;

    function setValidatorInfoMinDepositByIndex(
        uint256 index_,
        uint256 val_
    ) external;

    function incDescValidatorInfoMinDeposit(
        address key_,
        uint256 val_,
        ChangeMode mode_
    ) external;

    function incDescValidatorInfoMinDepositByIndex(
        uint256 index_,
        uint256 val_,
        ChangeMode mode_
    ) external;

    function setValidatorInfoAmount(address key_, uint256 val_) external;

    function setValidatorInfoAmountByIndex(
        uint256 index_,
        uint256 val_
    ) external;

    function incDescValidatorInfoAmount(
        address key_,
        uint256 val_,
        ChangeMode mode_
    ) external;

    function incDescValidatorInfoAmountByIndex(
        uint256 index_,
        uint256 val_,
        ChangeMode mode_
    ) external;

    function setValidatorInfoReward(address key_, uint256 val_) external;

    function setValidatorInfoRewardByIndex(
        uint256 index_,
        uint256 val_
    ) external;

    function incDescValidatorInfoReward(
        address key_,
        uint256 val_,
        ChangeMode mode_
    ) external;

    function incDescValidatorInfoRewardByIndex(
        uint256 index_,
        uint256 val_,
        ChangeMode mode_
    ) external;

    function setValidatorInfoSigner(address key_, address val_) external;

    function setValidatorInfoSignerByIndex(
        uint256 index_,
        address val_
    ) external;

    function setValidatorInfoValidatorShareContract(
        address key_,
        address val_
    ) external;

    function setValidatorInfoValidatorShareContractByIndex(
        uint256 index_,
        address val_
    ) external;

    function setValidatorInfoStatus(address key_, Status val_) external;

    function setValidatorInfoStatusByIndex(
        uint256 index_,
        Status val_
    ) external;

    function setValidatorInfoInfraCommissionRate(
        address key_,
        uint256 val_
    ) external;
    function setValidatorInfoInfraCommissionRateByIndex(
        uint256 index_,
        uint256 val_
    ) external;

    function incDescValidatorInfoInfraCommissionRate(
        address key_,
        uint256 val_,
        ChangeMode mode_
    ) external;

    function incDescValidatorInfoInfraCommissionRateByIndex(
        uint256 index_,
        uint256 val_,
        ChangeMode mode_
    ) external;

    function setValidatorInfoInfraCommissionAmount(
        address key_,
        uint256 val_
    ) external;

    function setValidatorInfoInfraCommissionAmountByIndex(
        uint256 index_,
        uint256 val_
    ) external;

    function incDescValidatorInfoInfraCommissionAmount(
        address key_,
        uint256 val_,
        ChangeMode mode_
    ) external;

    function incDescValidatorInfoInfraCommissionAmountByIndex(
        uint256 index_,
        uint256 val_,
        ChangeMode mode_
    ) external;

    function setValidatorInfoCommissionRate(
        address key_,
        uint256 val_
    ) external;

    function setValidatorInfoCommissionRateByIndex(
        uint256 index_,
        uint256 val_
    ) external;

    function incDescValidatorInfoCommissionRate(
        address key_,
        uint256 val_,
        ChangeMode mode_
    ) external;

    function incDescValidatorInfoCommissionRateByIndex(
        uint256 index_,
        uint256 val_,
        ChangeMode mode_
    ) external;

    function setValidatorInfoValidatorCommissionAmount(
        address key_,
        uint256 val_
    ) external;

    function setValidatorInfoValidatorCommissionAmountByIndex(
        uint256 index_,
        uint256 val_
    ) external;

    function incDescValidatorInfoValidatorCommissionAmount(
        address key_,
        uint256 val_,
        ChangeMode mode_
    ) external;

    function incDescValidatorInfoValidatorCommissionAmountByIndex(
        uint256 index_,
        uint256 val_,
        ChangeMode mode_
    ) external;

    function setValidatorInfoDelegatorCommissionAmount(
        address key_,
        uint256 val_
    ) external;

    function setValidatorInfoDelegatorCommissionAmountByIndex(
        uint256 index_,
        uint256 val_
    ) external;

    function incDescValidatorInfoDelegatorCommissionAmount(
        address key_,
        uint256 val_,
        ChangeMode mode_
    ) external;

    function incDescValidatorInfoDelegatorCommissionAmountByIndex(
        uint256 index_,
        uint256 val_,
        ChangeMode mode_
    ) external;

    function setValidatorInfoDelegatorsReward(
        address key_,
        uint256 val_
    ) external;

    function setValidatorInfoDelegatorsRewardByIndex(
        uint256 index_,
        uint256 val_
    ) external;

    function incDescValidatorInfoDelegatorsReward(
        address key_,
        uint256 val_,
        ChangeMode mode_
    ) external;

    function incDescValidatorInfoDelegatorsRewardByIndex(
        uint256 index_,
        uint256 val_,
        ChangeMode mode_
    ) external;

    function setValidatorInfoDelegatedAmount(
        address key_,
        uint256 val_
    ) external;

    function setValidatorInfoDelegatedAmountByIndex(
        uint256 index_,
        uint256 val_
    ) external;

    function incDescValidatorInfoDelegatedAmount(
        address key_,
        uint256 val_,
        ChangeMode mode_
    ) external;

    function incDescValidatorInfoDelegatedAmountByIndex(
        uint256 index_,
        uint256 val_,
        ChangeMode mode_
    ) external;

    function getValidatorInfo(
        address key_
    ) external view returns (Validator memory);

    function getValidatorInfoByIndex(
        uint256 index_
    ) external view returns (Validator memory);

    function getValidatorInfoMinDeposit(
        address key_
    ) external view returns (uint256);

    function getValidatorInfoMinDepositByIndex(
        uint256 index_
    ) external view returns (uint256);

    function getValidatorInfoAmount(
        address key_
    ) external view returns (uint256);

    function getValidatorInfoAmountByIndex(
        uint256 index_
    ) external view returns (uint256);

    function getValidatorInfoReward(
        address key_
    ) external view returns (uint256);

    function getValidatorInfoRewardByIndex(
        uint256 index_
    ) external view returns (uint256);
    function getValidatorInfoSigner(
        address key_
    ) external view returns (address);

    function getValidatorInfoSignerByIndex(
        uint256 index_
    ) external view returns (address);

    function getValidatorInfoValidatorShareContract(
        address key_
    ) external view returns (address);

    function getValidatorInfoValidatorShareContractByIndex(
        uint256 index_
    ) external view returns (address);

    function getValidatorInfoStatus(
        address key_
    ) external view returns (Status);

    function getValidatorInfoStatusByIndex(
        uint256 index_
    ) external view returns (Status);

    function getValidatorInfoInfraCommissionRate(
        address key_
    ) external view returns (uint256);

    function getValidatorInfoInfraCommissionRateByIndex(
        uint256 index_
    ) external view returns (uint256);

    function getValidatorInfoInfraCommissionAmount(
        address key_
    ) external view returns (uint256);

    function getValidatorInfoInfraCommissionAmountByIndex(
        uint256 index_
    ) external view returns (uint256);

    function getValidatorInfoCommissionRate(
        address key_
    ) external view returns (uint256);
    function getValidatorInfoCommissionRateByIndex(
        uint256 index_
    ) external view returns (uint256);

    function getValidatorInfoValidatorCommissionAmount(
        address key_
    ) external view returns (uint256);

    function getValidatorInfoValidatorCommissionAmountByIndex(
        uint256 index_
    ) external view returns (uint256);

    function getValidatorInfoDelegatorCommissionAmount(
        address key_
    ) external view returns (uint256);

    function getValidatorInfoDelegatorCommissionAmountByIndex(
        uint256 index_
    ) external view returns (uint256);

    function getValidatorInfoDelegatorsReward(
        address key_
    ) external view returns (uint256);

    function getValidatorInfoDelegatorsRewardByIndex(
        uint256 index_
    ) external view returns (uint256);

    function getValidatorInfoDelegatedAmount(
        address key_
    ) external view returns (uint256);

    function getValidatorInfoDelegatedAmountByIndex(
        uint256 index_
    ) external view returns (uint256);

    function getValidatorListLength() external view returns (uint256);

    function isInValidatorList(address val_) external view returns (bool);

    function getValidatorCurrentIndex(
        address val_,
        bool revertIfNotFound_
    ) external view returns (uint256, bool);
    function getValidatorByIndex(
        uint256 index_
    ) external view returns (address);

    function getAllValidator() external view returns (address[] memory);

    function getValidatorByPage(
        uint256 page,
        uint256 limit
    ) external view returns (address[] memory);

    function getValidatorIndexLength(
        address val_
    ) external view returns (uint256);

    function getValidatorIndexByIndex(
        address val_,
        uint256 index_
    ) external view returns (uint256);

    function getMinimalValidatorListLength() external view returns (uint256);

    function isInMinimalValidatorList(
        address val_
    ) external view returns (bool);

    function isInMinimalValidatorListByValidatorId(
        uint256 validatorId_
    ) external view returns (bool);

    function getMinimalValidatorIndex(
        address val_,
        bool revertIfNotFound_
    ) external view returns (uint256, bool);

    function getMinimalValidatorByIndex(
        uint256 index_
    ) external view returns (address);

    function getAllMinimalValidators() external view returns (address[] memory);

    function getMinimalValidatorsByPage(
        uint256 page,
        uint256 limit
    ) external view returns (address[] memory);
    function addMinimalValidator(address val_) external;

    function removeMinimalValidator(address val_) external;

    function removeMinimalValidatorByIndex(uint256 index_) external;

    function getMinimalValidatorsWithValidatorPowerByPage(
        uint256 page_,
        uint256 limit_
    ) external view returns (address[] memory, uint256[] memory);
}

library EnumerableSetAddress {
    struct AddressSet {
        address[] _values;
        mapping(address => uint256) _indexes;
    }

    function add(
        AddressSet storage set,
        address value
    ) internal returns (bool) {
        if (!contains(set, value)) {
            set._values.push(value);
            set._indexes[value] = set._values.length;
            return true;
        } else {
            return false;
        }
    }

    function remove(
        AddressSet storage set,
        address value
    ) internal returns (bool) {
        uint256 valueIndex = set._indexes[value];
        if (valueIndex != 0) {
            uint256 toDeleteIndex = valueIndex - 1;
            uint256 lastIndex = set._values.length - 1;
            address lastvalue = set._values[lastIndex];
            set._values[toDeleteIndex] = lastvalue;
            set._indexes[lastvalue] = toDeleteIndex + 1;
            set._values.pop();
            delete set._indexes[value];
            return true;
        } else {
            return false;
        }
    }

    function contains(
        AddressSet storage set,
        address value
    ) internal view returns (bool) {
        return set._indexes[value] != 0;
    }

    function length(AddressSet storage set) internal view returns (uint256) {
        return set._values.length;
    }

    function at(
        AddressSet storage set,
        uint256 index
    ) internal view returns (address) {
        require(
            set._values.length > index,
            "EnumerableSet: index out of bounds"
        );
        return set._values[index];
    }

    function index(
        AddressSet storage set,
        address value
    ) internal view returns (uint256, bool) {
        if (!contains(set, value)) {
            return (type(uint256).max, false);
        }
        return (set._indexes[value] - 1, true);
    }

    function getAll(
        AddressSet storage set
    ) internal view returns (address[] memory) {
        return set._values;
    }

    function get(
        AddressSet storage set,
        uint256 _page,
        uint256 _limit
    ) internal view returns (address[] memory) {
        require(_page > 0 && _limit > 0);
        uint256 tempLength = _limit;
        uint256 cursor = (_page - 1) * _limit;
        uint256 _addressLength = length(set);
        if (cursor >= _addressLength) {
            return new address[](0);
        }
        if (tempLength > _addressLength - cursor) {
            tempLength = _addressLength - cursor;
        }
        address[] memory addresses = new address[](tempLength);
        for (uint256 i = 0; i < tempLength; i++) {
            addresses[i] = at(set, cursor + i);
        }
        return addresses;
    }
}

/**
 * @title ChecksumAddress
 * @dev This library provides a function for computing the EIP-55 checksummed
hexadecimal representation without
 * 0x prefix of an address.
 * This library uses toHexString implementation from Strings library of OpenZeppelin
Contracts v4.7.0 to compute
 * hexadecimal string of an address.
 */
library ChecksumAddress {
    function toChecksumAddress(
        address src_
    ) internal pure returns (string memory) {
        bytes32 srcBytes = bytes32(bytes20(src_));
        bytes32 hash = keccak256(abi.encodePacked(_toHexString(src_)));

        // string of length 40 bytes (address without 0x prefix)
        string memory res = new string(40);

        assembly {
            // skip length word
            let resPtr := add(res, 32)
            // initialize shift right amount
            let shiftRightAmount := 256

            for {
                let hexIndex := 0
            } lt(hexIndex, 40) {
                hexIndex := add(hexIndex, 1)
            } {
                // shift right by 252 - (4 * hexIndex) bits
                shiftRightAmount := sub(shiftRightAmount, 4)

                // shift right then extract only the 4 last bits
                let selectedAddressHex := and(
                    shr(shiftRightAmount, srcBytes),
                    0xf
                )

                // if selectedAddressHex is a character
                switch gt(selectedAddressHex, 9)
                // if
                case 1 {
                    // shift right then extract only the 4 last bits
                    let selectedHashHex := and(shr(shiftRightAmount, hash), 0xf)

                    // if first bit of selectedHashHex is 1 (1XXX)
                    switch gt(selectedHashHex, 7)
                    // if
                    case 1 {
                        // append upper case character to res string
                        // offset 55

                        // store 8 bits (selectedAddressHex + 55) at res + 32 +
hexIndex
                        mstore8(
                            add(resPtr, hexIndex),
                            add(selectedAddressHex, 55)
                        )
                    }
                    case 0 {
                        // append lower case character to res string
                        // offset 87

                        // store 8 bits (selectedAddressHex + 87) at res + 32 +
hexIndex
                        mstore8(
                            add(resPtr, hexIndex),
                            add(selectedAddressHex, 87)
                        )
                    }
                }
                // else
                case 0 {
                    // append number to res string
                    // offset 48

                    // store 8 bits (selectedAddressHex + 48) at res + 32 + hexIndex
                    mstore8(add(resPtr, hexIndex), add(selectedAddressHex, 48))
                }
            }
        }

        return res;
    }


////////////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////

    // OpenZeppelin Contracts (last updated v4.7.0) (utils/Strings.sol)
    // Source file is from OpenZeppelin Contracts v4.7.0 MIT License.
(https://github.com/OpenZeppelin/openzeppelin-contracts/blob/release-v4.7/contracts/
utils/Strings.sol)
    // 0x prefix and unused functions are removed.

    bytes16 private constant _HEX_SYMBOLS = "0123456789abcdef";
    uint8 private constant _ADDRESS_LENGTH = 20;

    /**
     * @dev Converts an `address` with fixed length of 20 bytes to its not
checksummed ASCII `string` hexadecimal representation.
     *
     * Edit: remove 0x prefix from buffer string.
     */
    function _toHexString(address addr) private pure returns (string memory) {
        uint256 value = uint256(uint160(addr));
        bytes memory buffer = new bytes(2 * _ADDRESS_LENGTH);
        for (int256 i = 2 * int256(uint256(_ADDRESS_LENGTH)) - 1; i > -1; --i) {
            buffer[uint256(i)] = _HEX_SYMBOLS[value & 0xf];
            value >>= 4;
        }
        return string(buffer);
    }
}

struct StakeManagerStorageConstructorInput {
    address committee;
    uint256 defaultInfraCommissionRate;
    uint256 soloSlashRate;
    uint256 poolSlashAmount;
    address officialPool;
}

contract StakeManagerStorage is
    Committee,
    IStakeManagerStorage,
    StakeManagerConstants
{
    function _onlyCommitteeOrStakeManager() private view {
        require(
            msg.sender == committee || msg.sender == stakeManager,
            "StakeManagerStorage: restricted only committee or stake manager"
        );
    }

    modifier onlyCommitteeOrStakeManager() {
        _onlyCommitteeOrStakeManager();
        _;
    }

    function _onlyStakeManager() private view {
        require(
            msg.sender == stakeManager,
            "StakeManagerStorage: restricted only stake manager"
        );
    }

    modifier onlyStakeManager() {
        _onlyStakeManager();
        _;
    }

    function _checkIndexOutOfRange(
        uint256 index_,
        uint256 length_
    ) private pure {
        require(index_ < length_, "StakeManagerStorage: index out of range");
    }

    function _checkValidatorIndex(uint256 index_) private view {
        _checkIndexOutOfRange(index_, _validatorList.length);
    }

    modifier checkValidatorIndex(uint256 index_) {
        _checkValidatorIndex(index_);
        _;
    }

    function _checkExceedUint128(uint256 val_) private pure returns (uint128) {
        require(
            val_ <= type(uint128).max,
            "StakeManagerStorage: value exceeds uint128"
        );
        return uint128(val_);
    }

    function _checkExceedUint16(uint256 val_) private pure returns (uint16) {
        require(
            val_ <= type(uint16).max,
            "StakeManagerStorage: value exceeds uint16"
        );
        return uint16(val_);
    }

    using EnumerableSetAddress for EnumerableSetAddress.AddressSet;

    uint256 private constant _SPAN_SIZE = 50;

    bool[2] public initialized = [false, false];

    address public override stakeManager;

    address public override nftContract;
    address public override validatorShareFactory;
    address public override slashManager;

    uint256 public override soloLimit = 100;
    uint256 public override soloAmount;
    uint256 public override poolLimit = 50;
    uint256 public override poolAmount;
    uint256 public override officialLimit = 1;
    uint256 public override officialAmount;
    uint256 public override defaultInfraCommissionRate;

    uint256 public override totalStaked;
    uint256 public override totalRewards;
    uint256 public override totalRewardsLiquidated;
    uint256 public immutable override soloSlashRate;
    uint256 public override poolSlashAmount;
    uint256 public override unallocatedReward;

    bytes32[2] private _officialPools;

    mapping(address => uint256[]) private _validatorAddressToValidatorIds;
    Validator[] private _validatorList;

    EnumerableSetAddress.AddressSet private _minimalValidatorList;

    event OfficialSignerChanged(
        address indexed oldOfficialSigner,
        address indexed newOfficialSigner,
        uint256 indexed currentValidatorId,
        uint256 newValidBlock,
        address caller
    );
    event StakeManagerSet(
        address indexed oldStakeManager,
        address indexed newStakeManager,
        address indexed caller
    );
    event ValidatorShareFactorySet(
        address indexed oldValidatorShareFactory,
        address indexed newValidatorShareFactory,
        address indexed caller
    );
    event SlashManagerSet(
        address indexed oldSlashManager,
        address indexed newSlashManager,
        address indexed caller
    );

    event ValueSet(
        bytes32 indexed hash,
        bytes32 oldValue,
        bytes32 newValue,
        address indexed caller
    );

    event ValidatorAdded(address indexed val, uint256 length);
    event ValidatorRemoved(address indexed val, uint256 length);
    event MinimalValidatorAdded(address indexed val, uint256 length);
    event MinimalValidatorRemoved(address indexed val, uint256 length);

    constructor(
        StakeManagerStorageConstructorInput memory input_
    ) Committee(input_.committee) {
        _setInfraCommissionRateValueCheck(input_.defaultInfraCommissionRate);
        _setSoloSlashRateValueCheck(input_.soloSlashRate);
        _checkExceedUint128(input_.poolSlashAmount);

        defaultInfraCommissionRate = input_.defaultInfraCommissionRate;
        soloSlashRate = input_.soloSlashRate;
        poolSlashAmount = input_.poolSlashAmount;
        _setOfficialPoolWithBlockAndSlot(input_.officialPool, 0, true);
    }

    // ********************************* official pool related functions
********************************

    function _setOfficialPoolWithBlockAndSlot(
        address officialPool_,
        uint256 blockNumber_,
        bool slot_
    ) private {
        require(
            blockNumber_ <= type(uint96).max,
            "StakeManagerStorage: the designated block number can't exceed uint96"
        );

        uint256 tmpUint256 = uint256(uint160(officialPool_));
        tmpUint256 += blockNumber_ << 160;
        _officialPools[!slot_ ? 0 : 1] = bytes32(tmpUint256);
    }

    function _unpackUint96AndAddress(
        bytes32 payload_
    ) private pure returns (uint96, address) {
        return (
            uint96(
                uint256(
                    (bytes32(
0xFFFFFFFFFFFFFFFFFFFFFFFF0000000000000000000000000000000000000000
                    ) & payload_) >> 160
                )
            ),
            address(
                uint160(
                    uint256(
                        bytes32(

0x000000000000000000000000FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
                        ) & payload_
                    )
                )
            )
        );
    }

    function officialPool() public view override returns (address) {
        (, address tmp) = _unpackUint96AndAddress(_officialPools[1]);
        return tmp;
    }

    function getNewOfficialPoolValidBlock()
        external
        view
        override
        returns (uint256)
    {
        (uint96 targetBlock, ) = _unpackUint96AndAddress(_officialPools[1]);
        return uint256(targetBlock);
    }

    function getOldNewOfficialSignerAndValidBlock()
        external
        view
        override
        returns (address, address, uint256)
    {
        (, address officialPool1) = _unpackUint96AndAddress(_officialPools[0]);
        (uint96 validBlock2, address officialPool2) = _unpackUint96AndAddress(
            _officialPools[1]
        );
        return (officialPool1, officialPool2, validBlock2);
    }
    function changeOfficialSigner(
        address newOfficialSigner_,
        string memory checksummedNewOfficialSigner_
    ) external override {
        require(
            keccak256(
                abi.encodePacked(
                    "0x",
                    ChecksumAddress.toChecksumAddress(newOfficialSigner_)
                )
            ) == keccak256(bytes(checksummedNewOfficialSigner_)),
            "StakeManagerStorage: bad address checksum"
        );

        (
            uint96 validBlock2,
            address currentOfficialPool
        ) = _unpackUint96AndAddress(_officialPools[1]);

        require(
            msg.sender == currentOfficialPool || msg.sender == committee,
            "StakeManagerStorage: not allowed"
        );
        require(
            block.number >= validBlock2,
            "StakeManagerStorage: must wait until the valid block"
        );
        require(
            currentOfficialPool != newOfficialSigner_,
            "StakeManagerStorage: can't set to the same signer"
        );
        require(
            _validatorAddressToValidatorIds[newOfficialSigner_].length == 0,
            "StakeManagerStorage: the new signer address is already occupied"
        );

        // adjust new valid block
        // 0 - 25, 50 - 75 = span + 1
        // 26 - 49, 76 - 99 = span + 2
        uint256 lastTwoDigits = block.number % 100;
        bool extraSpan = (lastTwoDigits > 25 && lastTwoDigits < _SPAN_SIZE) ||
            (lastTwoDigits > 75);
        uint256 newValidBlock = block.number +
            (_SPAN_SIZE - (block.number % _SPAN_SIZE));
        if (extraSpan) {
            newValidBlock += _SPAN_SIZE;
        }

        (uint256 currentValidatorId, bool valid) = getValidatorCurrentIndex(
            currentOfficialPool,
            false
        );

        // _officialPools
        _setOfficialPoolWithBlockAndSlot(
            newOfficialSigner_,
            newValidBlock,
            true
        );
        _setOfficialPoolWithBlockAndSlot(
            currentOfficialPool,
            newValidBlock,
            false
        );

        if (valid) {
            // _validatorList
            _validatorList[currentValidatorId].signer = newOfficialSigner_;

            // _validatorAddressToValidatorIds
            uint256[] memory validatorIds = _validatorAddressToValidatorIds[
                currentOfficialPool
            ];
            delete _validatorAddressToValidatorIds[currentOfficialPool];
            _validatorAddressToValidatorIds[newOfficialSigner_] = validatorIds;

            // _minimalValidatorList
            _removeMinimalValidator(currentOfficialPool);
            _addMinimalValidator(newOfficialSigner_);
        }

        emit OfficialSignerChanged(
            currentOfficialPool,
            newOfficialSigner_,
            currentValidatorId,
            newValidBlock,
            msg.sender
        );
    }
    //
************************************************************************************
***************

    function _emitValueSet(
        bytes32 hash_,
        bytes32 oldValue_,
        bytes32 newValue_
    ) private {
        emit ValueSet(hash_, oldValue_, newValue_, msg.sender);
    }

    function _checkInitializedAndInitialize(uint256 index_) private {
        require(
            !initialized[index_],
            "StakeManagerStorage: already initialized"
        );
        initialized[index_] = true;
    }

    function initialize1(
        address nftContract_,
        address validatorShareFactory_,
        address slashManager_
    ) external {
        _checkInitializedAndInitialize(0);

        emit SlashManagerSet(slashManager, slashManager_, msg.sender);
        emit ValidatorShareFactorySet(
            validatorShareFactory,
            validatorShareFactory_,
            msg.sender
        );

        nftContract = nftContract_;
        validatorShareFactory = validatorShareFactory_;
        slashManager = slashManager_;
    }

    function initialize2(address stakeManager_) external {
        _checkInitializedAndInitialize(1);

        emit StakeManagerSet(stakeManager, stakeManager_, msg.sender);
        stakeManager = stakeManager_;
    }

    function setStakeManager(
        address stakeManager_
    ) external override onlyCommittee {
        emit StakeManagerSet(stakeManager, stakeManager_, msg.sender);
        stakeManager = stakeManager_;
    }

    function setValidatorShareFactory(
        address validatorShareFactory_
    ) external override onlyCommittee {
        emit ValidatorShareFactorySet(
            validatorShareFactory,
            validatorShareFactory_,
            msg.sender
        );
        validatorShareFactory = validatorShareFactory_;
    }

    function setSlashManager(
        address slashManager_
    ) external override onlyCommittee {
        emit SlashManagerSet(slashManager, slashManager_, msg.sender);
        slashManager = slashManager_;
    }

    // ******************************** stake manager function
********************************

    function _tryGetLatestValidatorId(
        address key_
    ) private view returns (uint256, bool) {
        uint256 tmp = _validatorAddressToValidatorIds[key_].length;
        if (tmp == 0) {
            return (INCORRECT_VALIDATOR_ID, false);
        } else {
            return (_validatorAddressToValidatorIds[key_][tmp - 1], true);
        }
    }

    function _getLatestValidatorId(
        address key_
    ) private view returns (uint256) {
        (uint256 validatorId, bool valid) = _tryGetLatestValidatorId(key_);
        require(valid, "StakeManagerStorage: validator id not found");
        return validatorId;
    }

    function _getValidatorInfoStoragePointer(
        address key_
    ) private view returns (Validator storage) {
        return
            _getValidatorInfoStoragePointerByIndex(_getLatestValidatorId(key_));
    }

    function _getValidatorInfoStoragePointerByIndex(
        uint256 index_
    ) private view checkValidatorIndex(index_) returns (Validator storage) {
        return _validatorList[index_];
    }

    function setSoloLimit(
        uint256 val_
    ) external override onlyCommitteeOrStakeManager {
        _emitValueSet(
            keccak256("soloLimit"),
            bytes32(soloLimit),
            bytes32(val_)
        );

        soloLimit = val_;
    }

    function incDescSoloLimit(
        uint256 val_,
        ChangeMode mode_
    ) external override onlyCommitteeOrStakeManager {
        uint256 oldSoloLimit = soloLimit;

        if (mode_ == ChangeMode.Decrease) {
            soloLimit -= val_;
        } else if (mode_ == ChangeMode.Increase) {
            soloLimit += val_;
        }

        _emitValueSet(
            keccak256("soloLimit"),
            bytes32(oldSoloLimit),
            bytes32(soloLimit)
        );
    }

    function setSoloAmount(
        uint256 val_
    ) external override onlyCommitteeOrStakeManager {
        soloAmount = val_;
    }

    function incDescSoloAmount(
        uint256 val_,
        ChangeMode mode_
    ) external override onlyCommitteeOrStakeManager {
        if (mode_ == ChangeMode.Decrease) {
            soloAmount -= val_;
        } else if (mode_ == ChangeMode.Increase) {
            soloAmount += val_;
        }
    }

    function setPoolLimit(
        uint256 val_
    ) external override onlyCommitteeOrStakeManager {
        _emitValueSet(
            keccak256("poolLimit"),
            bytes32(poolLimit),
            bytes32(val_)
        );

        poolLimit = val_;
    }

    function incDescPoolLimit(
        uint256 val_,
        ChangeMode mode_
    ) external override onlyCommitteeOrStakeManager {
        uint256 oldPoolLimit = poolLimit;

        if (mode_ == ChangeMode.Decrease) {
            poolLimit -= val_;
        } else if (mode_ == ChangeMode.Increase) {
            poolLimit += val_;
        }

        _emitValueSet(
            keccak256("poolLimit"),
            bytes32(oldPoolLimit),
            bytes32(poolLimit)
        );
    }

    function setPoolAmount(
        uint256 val_
    ) external override onlyCommitteeOrStakeManager {
        poolAmount = val_;
    }

    function incDescPoolAmount(
        uint256 val_,
        ChangeMode mode_
    ) external override onlyCommitteeOrStakeManager {
        if (mode_ == ChangeMode.Decrease) {
            poolAmount -= val_;
        } else if (mode_ == ChangeMode.Increase) {
            poolAmount += val_;
        }
    }

    function setOfficialLimit(
        uint256 val_
    ) external override onlyCommitteeOrStakeManager {
        _emitValueSet(
            keccak256("officialLimit"),
            bytes32(officialLimit),
            bytes32(val_)
        );

        officialLimit = val_;
    }

    function incDescOfficialLimit(
        uint256 val_,
        ChangeMode mode_
    ) external override onlyCommitteeOrStakeManager {
        uint256 oldOfficialLimit = officialLimit;

        if (mode_ == ChangeMode.Decrease) {
            officialLimit -= val_;
        } else if (mode_ == ChangeMode.Increase) {
            officialLimit += val_;
        }

        _emitValueSet(
            keccak256("officialLimit"),
            bytes32(oldOfficialLimit),
            bytes32(officialLimit)
        );
    }

    function setOfficialAmount(
        uint256 val_
    ) external override onlyCommitteeOrStakeManager {
        officialAmount = val_;
    }

    function incDescOfficialAmount(
        uint256 val_,
        ChangeMode mode_
    ) external override onlyCommitteeOrStakeManager {
        if (mode_ == ChangeMode.Decrease) {
            officialAmount -= val_;
        } else if (mode_ == ChangeMode.Increase) {
            officialAmount += val_;
        }
    }

    function _setInfraCommissionRateValueCheck(uint256 val_) private pure {
        require(
            val_ <= MAX_INFRA_COMMISSION_RATE,
            "StakeManagerStorage: max infra commission rate exceeded"
        );
    }

    function setDefaultInfraCommissionRate(
        uint256 val_
    ) external override onlyCommitteeOrStakeManager {
        _emitValueSet(
            keccak256("defaultInfraCommissionRate"),
            bytes32(defaultInfraCommissionRate),
            bytes32(val_)
        );

        _setInfraCommissionRateValueCheck(val_);
        defaultInfraCommissionRate = val_;
    }
    function incDescDefaultInfraCommissionRate(
        uint256 val_,
        ChangeMode mode_
    ) external override onlyCommitteeOrStakeManager {
        uint256 localDefaultInfraCommissionRate = defaultInfraCommissionRate;
        uint256 oldInfraCommissionRate = localDefaultInfraCommissionRate;

        if (mode_ == ChangeMode.Decrease) {
            localDefaultInfraCommissionRate -= val_;
        } else if (mode_ == ChangeMode.Increase) {
            localDefaultInfraCommissionRate += val_;
        }

        _setInfraCommissionRateValueCheck(localDefaultInfraCommissionRate);
        defaultInfraCommissionRate = localDefaultInfraCommissionRate;

        _emitValueSet(
            keccak256("defaultInfraCommissionRate"),
            bytes32(oldInfraCommissionRate),
            bytes32(localDefaultInfraCommissionRate)
        );
    }

    function setTotalStaked(uint256 val_) external override onlyStakeManager {
        totalStaked = val_;
    }

    function incDescTotalStaked(
        uint256 val_,
        ChangeMode mode_
    ) external override onlyStakeManager {
        if (mode_ == ChangeMode.Decrease) {
            totalStaked -= val_;
        } else if (mode_ == ChangeMode.Increase) {
            totalStaked += val_;
        }
    }

    function setTotalRewards(uint256 val_) external override onlyStakeManager {
        totalRewards = val_;
    }

    function incDescTotalRewards(
        uint256 val_,
        ChangeMode mode_
    ) external override onlyStakeManager {
        if (mode_ == ChangeMode.Decrease) {
            totalRewards -= val_;
        } else if (mode_ == ChangeMode.Increase) {
            totalRewards += val_;
        }
    }

    function setTotalRewardsLiquidated(
        uint256 val_
    ) external override onlyStakeManager {
        totalRewardsLiquidated = val_;
    }

    function incDescTotalRewardsLiquidated(
        uint256 val_,
        ChangeMode mode_
    ) external override onlyStakeManager {
        if (mode_ == ChangeMode.Decrease) {
            totalRewardsLiquidated -= val_;
        } else if (mode_ == ChangeMode.Increase) {
            totalRewardsLiquidated += val_;
        }
    }

    function _setSoloSlashRateValueCheck(uint256 val_) private pure {
        require(
            val_ <= MAX_RATE,
            "StakeManagerStorage: max solo slash rate exceeded"
        );
    }

    function setPoolSlashAmount(
        uint256 val_
    ) external override onlyCommitteeOrStakeManager {
        _checkExceedUint128(val_);

        _emitValueSet(
            keccak256("poolSlashAmount"),
            bytes32(poolSlashAmount),
            bytes32(val_)
        );

        poolSlashAmount = val_;
    }

    function incDescPoolSlashAmount(
        uint256 val_,
        ChangeMode mode_
    ) external override onlyCommitteeOrStakeManager {
        _checkExceedUint128(val_);
        uint256 oldPoolSlashAmount = poolSlashAmount;

        if (mode_ == ChangeMode.Decrease) {
            poolSlashAmount -= val_;
        } else if (mode_ == ChangeMode.Increase) {
            poolSlashAmount += val_;
        }

        _checkExceedUint128(poolSlashAmount);

        _emitValueSet(
            keccak256("poolSlashAmount"),
            bytes32(oldPoolSlashAmount),
            bytes32(poolSlashAmount)
        );
    }

    function setUnallocatedReward(
        uint256 val_
    ) external override onlyStakeManager {
        unallocatedReward = val_;
    }

    function incDescUnallocatedReward(
        uint256 val_,
        ChangeMode mode_
    ) external override onlyStakeManager {
        if (mode_ == ChangeMode.Decrease) {
            unallocatedReward -= val_;
        } else if (mode_ == ChangeMode.Increase) {
            unallocatedReward += val_;
        }
    }

    function addValidator(
        address key_,
        Validator memory val_
    ) external override onlyStakeManager returns (uint256) {
        if (_validatorAddressToValidatorIds[key_].length != 0) {
            (uint256 tmp, ) = _tryGetLatestValidatorId(key_);
            require(
                _validatorList[tmp].status == Status.Unstaked,
                "StakeManagerStorage: can't add as a validator while active"
            );
        }

        uint256 validatorId = _validatorList.length;
        _validatorAddressToValidatorIds[key_].push(validatorId);
        _validatorList.push(val_);

        emit ValidatorAdded(key_, _validatorList.length);
        return validatorId;
    }

    function setValidatorIds(
        address key_,
        uint256[] memory validatorIds_
    ) external override onlyStakeManager {
        _validatorAddressToValidatorIds[key_] = validatorIds_;
    }

    function setValidatorInfo(
        address key_,
        Validator memory val_
    ) public override onlyStakeManager {
        _validatorList[_getLatestValidatorId(key_)] = val_;
    }

    function setValidatorInfoByIndex(
        uint256 index_,
        Validator memory val_
    ) external override onlyStakeManager checkValidatorIndex(index_) {
        _validatorList[index_] = val_;
    }

    function setValidatorInfoMinDeposit(
        address key_,
        uint256 val_
    ) external override onlyStakeManager {
        _getValidatorInfoStoragePointer(key_).minDeposit = _checkExceedUint128(
            val_
        );
    }
    function setValidatorInfoMinDepositByIndex(
        uint256 index_,
        uint256 val_
    ) external override onlyStakeManager {
        _getValidatorInfoStoragePointerByIndex(index_)
            .minDeposit = _checkExceedUint128(val_);
    }

    function incDescValidatorInfoMinDeposit(
        address key_,
        uint256 val_,
        ChangeMode mode_
    ) external override onlyStakeManager {
        uint128 uint128Val = _checkExceedUint128(val_);

        if (mode_ == ChangeMode.Decrease) {
            _getValidatorInfoStoragePointer(key_).minDeposit -= uint128Val;
        } else if (mode_ == ChangeMode.Increase) {
            _getValidatorInfoStoragePointer(key_).minDeposit += uint128Val;
        }
    }

    function incDescValidatorInfoMinDepositByIndex(
        uint256 index_,
        uint256 val_,
        ChangeMode mode_
    ) external override onlyStakeManager {
        uint128 uint128Val = _checkExceedUint128(val_);

        if (mode_ == ChangeMode.Decrease) {
            _getValidatorInfoStoragePointerByIndex(index_)
                .minDeposit -= uint128Val;
        } else if (mode_ == ChangeMode.Increase) {
            _getValidatorInfoStoragePointerByIndex(index_)
                .minDeposit += uint128Val;
        }
    }

    function setValidatorInfoAmount(
        address key_,
        uint256 val_
    ) external override onlyStakeManager {
        _getValidatorInfoStoragePointer(key_).amount = _checkExceedUint128(
            val_
        );
    }

    function setValidatorInfoAmountByIndex(
        uint256 index_,
        uint256 val_
    ) external override onlyStakeManager {
        _getValidatorInfoStoragePointerByIndex(index_)
            .amount = _checkExceedUint128(val_);
    }

    function incDescValidatorInfoAmount(
        address key_,
        uint256 val_,
        ChangeMode mode_
    ) external override onlyStakeManager {
        uint128 uint128Val = _checkExceedUint128(val_);

        if (mode_ == ChangeMode.Decrease) {
            _getValidatorInfoStoragePointer(key_).amount -= uint128Val;
        } else if (mode_ == ChangeMode.Increase) {
            _getValidatorInfoStoragePointer(key_).amount += uint128Val;
        }
    }

    function incDescValidatorInfoAmountByIndex(
        uint256 index_,
        uint256 val_,
        ChangeMode mode_
    ) external override onlyStakeManager {
        uint128 uint128Val = _checkExceedUint128(val_);

        if (mode_ == ChangeMode.Decrease) {
            _getValidatorInfoStoragePointerByIndex(index_).amount -= uint128Val;
        } else if (mode_ == ChangeMode.Increase) {
            _getValidatorInfoStoragePointerByIndex(index_).amount += uint128Val;
        }
    }

    function setValidatorInfoReward(
        address key_,
        uint256 val_
    ) external override onlyStakeManager {
        _getValidatorInfoStoragePointer(key_).reward = _checkExceedUint128(
            val_
        );
    }

    function setValidatorInfoRewardByIndex(
        uint256 index_,
        uint256 val_
    ) external override onlyStakeManager {
        _getValidatorInfoStoragePointerByIndex(index_)
            .reward = _checkExceedUint128(val_);
    }

    function incDescValidatorInfoReward(
        address key_,
        uint256 val_,
        ChangeMode mode_
    ) external override onlyStakeManager {
        uint128 uint128Val = _checkExceedUint128(val_);

        if (mode_ == ChangeMode.Decrease) {
            _getValidatorInfoStoragePointer(key_).reward -= uint128Val;
        } else if (mode_ == ChangeMode.Increase) {
            _getValidatorInfoStoragePointer(key_).reward += uint128Val;
        }
    }

    function incDescValidatorInfoRewardByIndex(
        uint256 index_,
        uint256 val_,
        ChangeMode mode_
    ) external override onlyStakeManager {
        uint128 uint128Val = _checkExceedUint128(val_);

        if (mode_ == ChangeMode.Decrease) {
            _getValidatorInfoStoragePointerByIndex(index_).reward -= uint128Val;
        } else if (mode_ == ChangeMode.Increase) {
            _getValidatorInfoStoragePointerByIndex(index_).reward += uint128Val;
        }
    }

    function setValidatorInfoSigner(
        address key_,
        address val_
    ) external override onlyStakeManager {
        _getValidatorInfoStoragePointer(key_).signer = val_;
    }
    function setValidatorInfoSignerByIndex(
        uint256 index_,
        address val_
    ) external override onlyStakeManager {
        _getValidatorInfoStoragePointerByIndex(index_).signer = val_;
    }

    function setValidatorInfoValidatorShareContract(
        address key_,
        address val_
    ) external override onlyStakeManager {
        _getValidatorInfoStoragePointer(key_).validatorShareContract = val_;
    }

    function setValidatorInfoValidatorShareContractByIndex(
        uint256 index_,
        address val_
    ) external override onlyStakeManager {
        _getValidatorInfoStoragePointerByIndex(index_)
            .validatorShareContract = val_;
    }

    function setValidatorInfoStatus(
        address key_,
        Status val_
    ) external override onlyStakeManager {
        _getValidatorInfoStoragePointer(key_).status = val_;
    }

    function setValidatorInfoStatusByIndex(
        uint256 index_,
        Status val_
    ) external override onlyStakeManager {
        _getValidatorInfoStoragePointerByIndex(index_).status = val_;
    }

    function setValidatorInfoInfraCommissionRate(
        address key_,
        uint256 val_
    ) external override onlyStakeManager {
        _setInfraCommissionRateValueCheck(val_);
        _getValidatorInfoStoragePointer(key_).infraCommissionRate = uint16(
            val_
        );
    }

    function setValidatorInfoInfraCommissionRateByIndex(
        uint256 index_,
        uint256 val_
    ) external override onlyStakeManager {
        _setInfraCommissionRateValueCheck(val_);
        _getValidatorInfoStoragePointerByIndex(index_)
            .infraCommissionRate = uint16(val_);
    }

    function _incDescValidatorInfoInfraCommissionRate(
        Validator storage validator_,
        uint256 val_,
        ChangeMode mode_
    ) private {
        uint256 localInfraCommissionRate = validator_.infraCommissionRate;

        if (mode_ == ChangeMode.Decrease) {
            localInfraCommissionRate -= val_;
        } else if (mode_ == ChangeMode.Increase) {
            localInfraCommissionRate += val_;
        }

        _setInfraCommissionRateValueCheck(localInfraCommissionRate);

        validator_.infraCommissionRate = uint16(localInfraCommissionRate);
    }

    function incDescValidatorInfoInfraCommissionRate(
        address key_,
        uint256 val_,
        ChangeMode mode_
    ) external override onlyStakeManager {
        Validator storage validator = _getValidatorInfoStoragePointer(key_);
        _incDescValidatorInfoInfraCommissionRate(validator, val_, mode_);
    }

    function incDescValidatorInfoInfraCommissionRateByIndex(
        uint256 index_,
        uint256 val_,
        ChangeMode mode_
    ) external override onlyStakeManager {
        Validator storage validator = _getValidatorInfoStoragePointerByIndex(
            index_
        );
        _incDescValidatorInfoInfraCommissionRate(validator, val_, mode_);
    }

    function setValidatorInfoInfraCommissionAmount(
        address key_,
        uint256 val_
    ) external override onlyStakeManager {
        _getValidatorInfoStoragePointer(key_)
            .infraCommissionAmount = _checkExceedUint128(val_);
    }

    function setValidatorInfoInfraCommissionAmountByIndex(
        uint256 index_,
        uint256 val_
    ) external override onlyStakeManager {
        _getValidatorInfoStoragePointerByIndex(index_)
            .infraCommissionAmount = _checkExceedUint128(val_);
    }

    function _incDescValidatorInfoInfraCommissionAmount(
        Validator storage validator_,
        uint256 val_,
        ChangeMode mode_
    ) private {
        uint128 uint128Val = _checkExceedUint128(val_);
        uint128 localInfraCommissionAmount = validator_.infraCommissionAmount;

        if (mode_ == ChangeMode.Decrease) {
            localInfraCommissionAmount -= uint128Val;
        } else if (mode_ == ChangeMode.Increase) {
            localInfraCommissionAmount += uint128Val;
        }

        validator_.infraCommissionAmount = localInfraCommissionAmount;
    }

    function incDescValidatorInfoInfraCommissionAmount(
        address key_,
        uint256 val_,
        ChangeMode mode_
    ) external override onlyStakeManager {
        Validator storage validator = _getValidatorInfoStoragePointer(key_);
        _incDescValidatorInfoInfraCommissionAmount(validator, val_, mode_);
    }
    function incDescValidatorInfoInfraCommissionAmountByIndex(
        uint256 index_,
        uint256 val_,
        ChangeMode mode_
    ) external override onlyStakeManager {
        Validator storage validator = _getValidatorInfoStoragePointerByIndex(
            index_
        );
        _incDescValidatorInfoInfraCommissionAmount(validator, val_, mode_);
    }

    function _setValidatorInfoCommissionRateValueCheck(
        uint256 val_
    ) private pure {
        require(
            val_ <= MAX_COMMISSION_RATE,
            "StakeManagerStorage: max commission rate exceeded"
        );
    }

    function setValidatorInfoCommissionRate(
        address key_,
        uint256 val_
    ) external override onlyStakeManager {
        _setValidatorInfoCommissionRateValueCheck(val_);
        _getValidatorInfoStoragePointer(key_).commissionRate = uint16(val_);
    }

    function setValidatorInfoCommissionRateByIndex(
        uint256 index_,
        uint256 val_
    ) external override onlyStakeManager {
        _setValidatorInfoCommissionRateValueCheck(val_);
        _getValidatorInfoStoragePointerByIndex(index_).commissionRate = uint16(
            val_
        );
    }

    function _incDescValidatorInfoCommissionRate(
        Validator storage validator_,
        uint256 val_,
        ChangeMode mode_
    ) private {
        uint16 uint16Val = _checkExceedUint16(val_);
        uint16 localCommissionRate = validator_.commissionRate;

        if (mode_ == ChangeMode.Decrease) {
            localCommissionRate -= uint16Val;
        } else if (mode_ == ChangeMode.Increase) {
            localCommissionRate += uint16Val;
        }

        _setValidatorInfoCommissionRateValueCheck(localCommissionRate);

        validator_.commissionRate = localCommissionRate;
    }

    function incDescValidatorInfoCommissionRate(
        address key_,
        uint256 val_,
        ChangeMode mode_
    ) external override onlyStakeManager {
        Validator storage validator = _getValidatorInfoStoragePointer(key_);
        _incDescValidatorInfoCommissionRate(validator, val_, mode_);
    }

    function incDescValidatorInfoCommissionRateByIndex(
        uint256 index_,
        uint256 val_,
        ChangeMode mode_
    ) external override onlyStakeManager {
        Validator storage validator = _getValidatorInfoStoragePointerByIndex(
            index_
        );
        _incDescValidatorInfoCommissionRate(validator, val_, mode_);
    }

    function setValidatorInfoValidatorCommissionAmount(
        address key_,
        uint256 val_
    ) external override onlyStakeManager {
        _getValidatorInfoStoragePointer(key_)
            .validatorCommissionAmount = _checkExceedUint128(val_);
    }

    function setValidatorInfoValidatorCommissionAmountByIndex(
        uint256 index_,
        uint256 val_
    ) external override onlyStakeManager {
        _getValidatorInfoStoragePointerByIndex(index_)
            .validatorCommissionAmount = _checkExceedUint128(val_);
    }

    function incDescValidatorInfoValidatorCommissionAmount(
        address key_,
        uint256 val_,
        ChangeMode mode_
    ) external override onlyStakeManager {
        uint128 uint128Val = _checkExceedUint128(val_);

        if (mode_ == ChangeMode.Decrease) {
            _getValidatorInfoStoragePointer(key_)
                .validatorCommissionAmount -= uint128Val;
        } else if (mode_ == ChangeMode.Increase) {
            _getValidatorInfoStoragePointer(key_)
                .validatorCommissionAmount += uint128Val;
        }
    }

    function incDescValidatorInfoValidatorCommissionAmountByIndex(
        uint256 index_,
        uint256 val_,
        ChangeMode mode_
    ) external override onlyStakeManager {
        uint128 uint128Val = _checkExceedUint128(val_);

        if (mode_ == ChangeMode.Decrease) {
            _getValidatorInfoStoragePointerByIndex(index_)
                .validatorCommissionAmount -= uint128Val;
        } else if (mode_ == ChangeMode.Increase) {
            _getValidatorInfoStoragePointerByIndex(index_)
                .validatorCommissionAmount += uint128Val;
        }
    }

    function setValidatorInfoDelegatorCommissionAmount(
        address key_,
        uint256 val_
    ) external override onlyStakeManager {
        _getValidatorInfoStoragePointer(key_)
            .delegatorCommissionAmount = _checkExceedUint128(val_);
    }

    function setValidatorInfoDelegatorCommissionAmountByIndex(
        uint256 index_,
        uint256 val_
    ) external override onlyStakeManager {
        _getValidatorInfoStoragePointerByIndex(index_)
            .delegatorCommissionAmount = _checkExceedUint128(val_);
    }

    function incDescValidatorInfoDelegatorCommissionAmount(
        address key_,
        uint256 val_,
        ChangeMode mode_
    ) external override onlyStakeManager {
        uint128 uint128Val = _checkExceedUint128(val_);

        if (mode_ == ChangeMode.Decrease) {
            _getValidatorInfoStoragePointer(key_)
                .delegatorCommissionAmount -= uint128Val;
        } else if (mode_ == ChangeMode.Increase) {
            _getValidatorInfoStoragePointer(key_)
                .delegatorCommissionAmount += uint128Val;
        }
    }

    function incDescValidatorInfoDelegatorCommissionAmountByIndex(
        uint256 index_,
        uint256 val_,
        ChangeMode mode_
    ) external override onlyStakeManager {
        uint128 uint128Val = _checkExceedUint128(val_);

        if (mode_ == ChangeMode.Decrease) {
            _getValidatorInfoStoragePointerByIndex(index_)
                .delegatorCommissionAmount -= uint128Val;
        } else if (mode_ == ChangeMode.Increase) {
            _getValidatorInfoStoragePointerByIndex(index_)
                .delegatorCommissionAmount += uint128Val;
        }
    }

    function setValidatorInfoDelegatorsReward(
        address key_,
        uint256 val_
    ) external override onlyStakeManager {
        _getValidatorInfoStoragePointer(key_)
            .delegatorsReward = _checkExceedUint128(val_);
    }

    function setValidatorInfoDelegatorsRewardByIndex(
        uint256 index_,
        uint256 val_
    ) external override onlyStakeManager {
        _getValidatorInfoStoragePointerByIndex(index_)
            .delegatorsReward = _checkExceedUint128(val_);
    }

    function incDescValidatorInfoDelegatorsReward(
        address key_,
        uint256 val_,
        ChangeMode mode_
    ) external override onlyStakeManager {
        uint128 uint128Val = _checkExceedUint128(val_);

        if (mode_ == ChangeMode.Decrease) {
            _getValidatorInfoStoragePointer(key_)
                .delegatorsReward -= uint128Val;
        } else if (mode_ == ChangeMode.Increase) {
            _getValidatorInfoStoragePointer(key_)
                .delegatorsReward += uint128Val;
        }
    }

    function incDescValidatorInfoDelegatorsRewardByIndex(
        uint256 index_,
        uint256 val_,
        ChangeMode mode_
    ) external override onlyStakeManager {
        uint128 uint128Val = _checkExceedUint128(val_);

        if (mode_ == ChangeMode.Decrease) {
            _getValidatorInfoStoragePointerByIndex(index_)
                .delegatorsReward -= uint128Val;
        } else if (mode_ == ChangeMode.Increase) {
            _getValidatorInfoStoragePointerByIndex(index_)
                .delegatorsReward += uint128Val;
        }
    }

    function setValidatorInfoDelegatedAmount(
        address key_,
        uint256 val_
    ) external override onlyStakeManager {
        _getValidatorInfoStoragePointer(key_)
            .delegatedAmount = _checkExceedUint128(val_);
    }

    function setValidatorInfoDelegatedAmountByIndex(
        uint256 index_,
        uint256 val_
    ) external override onlyStakeManager {
        _getValidatorInfoStoragePointerByIndex(index_)
            .delegatedAmount = _checkExceedUint128(val_);
    }

    function incDescValidatorInfoDelegatedAmount(
        address key_,
        uint256 val_,
        ChangeMode mode_
    ) external override onlyStakeManager {
        uint128 uint128Val = _checkExceedUint128(val_);

        if (mode_ == ChangeMode.Decrease) {
            _getValidatorInfoStoragePointer(key_).delegatedAmount -= uint128Val;
        } else if (mode_ == ChangeMode.Increase) {
            _getValidatorInfoStoragePointer(key_).delegatedAmount += uint128Val;
        }
    }

    function incDescValidatorInfoDelegatedAmountByIndex(
        uint256 index_,
        uint256 val_,
        ChangeMode mode_
    ) external override onlyStakeManager {
        uint128 uint128Val = _checkExceedUint128(val_);

        if (mode_ == ChangeMode.Decrease) {
            _getValidatorInfoStoragePointerByIndex(index_)
                .delegatedAmount -= uint128Val;
        } else if (mode_ == ChangeMode.Increase) {
            _getValidatorInfoStoragePointerByIndex(index_)
                .delegatedAmount += uint128Val;
        }
    }

    function getValidatorInfo(
        address key_
    ) external view override returns (Validator memory) {
        return _getValidatorInfoStoragePointer(key_);
    }

    function getValidatorInfoByIndex(
        uint256 index_
    ) external view override returns (Validator memory) {
        return _getValidatorInfoStoragePointerByIndex(index_);
    }

    function getValidatorInfoMinDeposit(
        address key_
    ) external view override returns (uint256) {
        return _getValidatorInfoStoragePointer(key_).minDeposit;
    }

    function getValidatorInfoMinDepositByIndex(
        uint256 index_
    ) external view override returns (uint256) {
        return _getValidatorInfoStoragePointerByIndex(index_).minDeposit;
    }

    function getValidatorInfoAmount(
        address key_
    ) external view override returns (uint256) {
        return _getValidatorInfoStoragePointer(key_).amount;
    }

    function getValidatorInfoAmountByIndex(
        uint256 index_
    ) external view override returns (uint256) {
        return _getValidatorInfoStoragePointerByIndex(index_).amount;
    }

    function getValidatorInfoReward(
        address key_
    ) external view override returns (uint256) {
        return _getValidatorInfoStoragePointer(key_).reward;
    }

    function getValidatorInfoRewardByIndex(
        uint256 index_
    ) external view override returns (uint256) {
        return _getValidatorInfoStoragePointerByIndex(index_).reward;
    }
    function getValidatorInfoSigner(
        address key_
    ) external view override returns (address) {
        return _getValidatorInfoStoragePointer(key_).signer;
    }

    function getValidatorInfoSignerByIndex(
        uint256 index_
    ) external view override returns (address) {
        return _getValidatorInfoStoragePointerByIndex(index_).signer;
    }

    function getValidatorInfoValidatorShareContract(
        address key_
    ) external view override returns (address) {
        return _getValidatorInfoStoragePointer(key_).validatorShareContract;
    }

    function getValidatorInfoValidatorShareContractByIndex(
        uint256 index_
    ) external view override returns (address) {
        return
            _getValidatorInfoStoragePointerByIndex(index_)
                .validatorShareContract;
    }

    function getValidatorInfoStatus(
        address key_
    ) external view override returns (Status) {
        return _getValidatorInfoStoragePointer(key_).status;
    }

    function getValidatorInfoStatusByIndex(
        uint256 index_
    ) external view override returns (Status) {
        return _getValidatorInfoStoragePointerByIndex(index_).status;
    }

    function getValidatorInfoInfraCommissionRate(
        address key_
    ) external view override returns (uint256) {
        return _getValidatorInfoStoragePointer(key_).infraCommissionRate;
    }
    function getValidatorInfoInfraCommissionRateByIndex(
        uint256 index_
    ) external view override returns (uint256) {
        return
            _getValidatorInfoStoragePointerByIndex(index_).infraCommissionRate;
    }

    function getValidatorInfoInfraCommissionAmount(
        address key_
    ) external view override returns (uint256) {
        return _getValidatorInfoStoragePointer(key_).infraCommissionAmount;
    }

    function getValidatorInfoInfraCommissionAmountByIndex(
        uint256 index_
    ) external view override returns (uint256) {
        return
            _getValidatorInfoStoragePointerByIndex(index_)
                .infraCommissionAmount;
    }

    function getValidatorInfoCommissionRate(
        address key_
    ) external view override returns (uint256) {
        return _getValidatorInfoStoragePointer(key_).commissionRate;
    }

    function getValidatorInfoCommissionRateByIndex(
        uint256 index_
    ) external view override returns (uint256) {
        return _getValidatorInfoStoragePointerByIndex(index_).commissionRate;
    }

    function getValidatorInfoValidatorCommissionAmount(
        address key_
    ) external view override returns (uint256) {
        return _getValidatorInfoStoragePointer(key_).validatorCommissionAmount;
    }

    function getValidatorInfoValidatorCommissionAmountByIndex(
        uint256 index_
    ) external view override returns (uint256) {
        return
            _getValidatorInfoStoragePointerByIndex(index_)
                .validatorCommissionAmount;
    }

    function getValidatorInfoDelegatorCommissionAmount(
        address key_
    ) external view override returns (uint256) {
        return _getValidatorInfoStoragePointer(key_).delegatorCommissionAmount;
    }

    function getValidatorInfoDelegatorCommissionAmountByIndex(
        uint256 index_
    ) external view override returns (uint256) {
        return
            _getValidatorInfoStoragePointerByIndex(index_)
                .delegatorCommissionAmount;
    }

    function getValidatorInfoDelegatorsReward(
        address key_
    ) external view override returns (uint256) {
        return _getValidatorInfoStoragePointer(key_).delegatorsReward;
    }

    function getValidatorInfoDelegatorsRewardByIndex(
        uint256 index_
    ) external view override returns (uint256) {
        return _getValidatorInfoStoragePointerByIndex(index_).delegatorsReward;
    }

    function getValidatorInfoDelegatedAmount(
        address key_
    ) external view override returns (uint256) {
        return _getValidatorInfoStoragePointer(key_).delegatedAmount;
    }

    function getValidatorInfoDelegatedAmountByIndex(
        uint256 index_
    ) external view override returns (uint256) {
        return _getValidatorInfoStoragePointerByIndex(index_).delegatedAmount;
    }

    function getValidatorListLength() external view override returns (uint256) {
        return _validatorList.length;
    }

    function isInValidatorList(
        address val_
    ) external view override returns (bool) {
        (, bool isValid) = _tryGetLatestValidatorId(val_);
        return isValid;
    }

    function getValidatorCurrentIndex(
        address val_,
        bool revertIfNotFound_
    ) public view override returns (uint256, bool) {
        (uint256 val, bool valid) = _tryGetLatestValidatorId(val_);
        if (revertIfNotFound_) {
            require(valid, "StakeManagerStorage: value not found");
        }

        if (valid) {
            return (val, valid);
        } else {
            return (INCORRECT_VALIDATOR_ID, valid);
        }
    }

    function getValidatorByIndex(
        uint256 index_
    ) external view override checkValidatorIndex(index_) returns (address) {
        return _validatorList[index_].signer;
    }

    function getAllValidator()
        external
        view
        override
        returns (address[] memory validatorAddresses)
    {
        uint256 length = _validatorList.length;
        validatorAddresses = new address[](length);
        for (uint256 i = 0; i < length; i++) {
            validatorAddresses[i] = _validatorList[i].signer;
        }
    }

    function getValidatorByPage(
        uint256 page_,
        uint256 limit_
    ) external view override returns (address[] memory validatorAddresses) {
        require(
            page_ > 0 && limit_ > 0,
            "StakeManagerStorage: invalid page and limit"
        );
        uint256 tempLength = limit_;
        uint256 cursor = (page_ - 1) * limit_;
        uint256 _addressLength = _validatorList.length;
        if (cursor >= _addressLength) {
            return new address[](0);
        }
        if (tempLength > _addressLength - cursor) {
            tempLength = _addressLength - cursor;
        }
        validatorAddresses = new address[](tempLength);
        for (uint256 i = 0; i < tempLength; i++) {
            validatorAddresses[i] = _validatorList[cursor + i].signer;
        }
        return validatorAddresses;
    }

    function getValidatorIndexLength(
        address val_
    ) external view override returns (uint256) {
        return _validatorAddressToValidatorIds[val_].length;
    }

    function getValidatorIndexByIndex(
        address val_,
        uint256 index_
    ) external view override returns (uint256) {
        _checkIndexOutOfRange(
            index_,
            _validatorAddressToValidatorIds[val_].length
        );

        return _validatorAddressToValidatorIds[val_][index_];
    }

    function getMinimalValidatorListLength()
        external
        view
        override
        returns (uint256)
    {
        return _minimalValidatorList.length();
    }

    function isInMinimalValidatorList(
        address val_
    ) external view override returns (bool) {
        return _minimalValidatorList.contains(val_);
    }

    function isInMinimalValidatorListByValidatorId(
        uint256 validatorId_
    ) external view override returns (bool) {
        return
            _minimalValidatorList.contains(
                _getValidatorInfoStoragePointerByIndex(validatorId_).signer
            );
    }

    function getMinimalValidatorIndex(
        address val_,
        bool revertIfNotFound_
    ) external view override returns (uint256, bool) {
        (uint256 val, bool valid) = _minimalValidatorList.index(val_);
        if (revertIfNotFound_) {
            require(valid, "StakeManagerStorage: value not found");
        }
        return (val, valid);
    }

    function getMinimalValidatorByIndex(
        uint256 index_
    ) external view override returns (address) {
        return _minimalValidatorList.at(index_);
    }

    function getAllMinimalValidators()
        external
        view
        override
        returns (address[] memory)
    {
        return _minimalValidatorList.getAll();
    }

    function getMinimalValidatorsByPage(
        uint256 page,
        uint256 limit
    ) external view override returns (address[] memory) {
        return _minimalValidatorList.get(page, limit);
    }

    function _addMinimalValidator(address val_) private {
        require(
            _minimalValidatorList.add(val_),
            "StakeManagerStorage: already added"
        );
        emit MinimalValidatorAdded(val_, _minimalValidatorList.length());
    }

    function addMinimalValidator(
        address val_
    ) public override onlyStakeManager {
        _addMinimalValidator(val_);
    }

    function _removeMinimalValidator(address val_) private {
        require(
            _minimalValidatorList.remove(val_),
            "StakeManagerStorage: unknown value"
        );
        emit MinimalValidatorRemoved(val_, _minimalValidatorList.length());
    }

    function removeMinimalValidator(
        address val_
    ) public override onlyStakeManager {
        _removeMinimalValidator(val_);
    }

    function removeMinimalValidatorByIndex(
        uint256 index_
    ) external override onlyStakeManager {
        removeMinimalValidator(_minimalValidatorList.at(index_));
    }

    function _validatorPowers(
        address[] memory minimalValidators_
    ) private view returns (uint256[] memory validatorPowers) {
        uint256 length = minimalValidators_.length;
        validatorPowers = new uint256[](length);
        for (uint256 i = 0; i < length; i++) {
            (uint256 index, ) = getValidatorCurrentIndex(
                minimalValidators_[i],
                false
            );

            // storage will cause evm to only load necessary slots
            // in this case only one slot
            // not sure if there're better syntaxs
            Validator storage a = _validatorList[index];
            uint256 b = a.amount;
            uint256 c = a.delegatedAmount;
            validatorPowers[i] = b + c;
        }
    }

    function getMinimalValidatorsWithValidatorPowerByPage(
        uint256 page_,
        uint256 limit_
    )
        external
        view
        override
        returns (address[] memory minimalValidators, uint256[] memory)
    {
        if (page_ > 0) {
            minimalValidators = _minimalValidatorList.get(page_, limit_);
        } else {
            minimalValidators = _minimalValidatorList.getAll();
        }
        return (minimalValidators, _validatorPowers(minimalValidators));
    }
}
```

2. StakeManager

```
 // * Copyright 2023 Bitkub Blockchain Technology Co., Ltd. - All Rights Reserved.
// * This code is proprietary.
// * Unauthorized copying, modification or distribution of this code, via any
medium, is strictly prohibited without express permission.

pragma solidity ^0.8.0;
```

```
enum LiquidateMode {
    All,
    ValidatorReward,
    ValidatorCommission,
    DelegatorCommission
}

enum TransferTokenMode {
    None,
    Staked,
    Reward
}

enum ChangeMode {
    Increase,
    Decrease
}

enum Status {
    Uninitialized,
    Active,
    Unstaked
}

enum SlashErrorCode {
    Uninitialized,
    StakeAmountNotEnough,
    OfficialPool,
    NotPool,
    InactivatedPool
}

struct Validator {
    uint128 amount;
    uint128 delegatedAmount;
    uint128 reward;
    uint128 delegatorsReward;
    uint128 infraCommissionAmount;
    uint128 validatorCommissionAmount;
    uint128 delegatorCommissionAmount;
    uint128 minDeposit;
    address signer;
    address validatorShareContract;
    Status status;
    uint16 infraCommissionRate;
    uint16 commissionRate;
}

struct MinimalValidator {
    address signer;
    uint256 power;
}

interface IStakeManager {
    function setOfficialPoolStaker(address officialPoolStaker_) external;

    function transferUnallocatedReward(
        address payable to,
        uint256 amount
    ) external returns (bool);

    function transferInfraCommission(
        uint256[] memory validatorIds_,
        uint256[] memory amounts_,
        address payable to_
    ) external;

    function transferFunds(
        uint256 validatorId,
        uint256 amount,
        address delegator,
        TransferTokenMode mode
    ) external returns (bool);

    function delegationDeposit(
        uint256 validatorId
    ) external payable returns (bool);

    function stake(address signer, bool delegation) external payable;

    function stake(
        address signer,
        bool delegation,
        uint256 amount,
        address bitkubNext
    ) external;

    function unstake(uint256 validatorId) external;
    function unstake(uint256 validatorId, address bitkubNext) external;

    function restake(uint256 validatorId) external payable;

    function restake(
        uint256 validatorId,
        uint256 amount,
        address bitkubNext
    ) external;

    function distributeReward() external payable;

    function claimRewards(uint256 validatorId) external;

    function claimRewards(uint256 validatorId, address bitkubNext) external;

    function claimCommissionRewards(uint256 validatorId) external;

    function claimCommissionRewards(
        uint256 validatorId,
        address bitkubNext
    ) external;

    function updateInfraCommissionRate(
        uint256 validatorId_,
        uint256 newInfraCommissionRate_
    ) external;

    function updateCommissionRate(
        uint256 validatorId,
        uint256 newCommissionRate
    ) external;

    function updateCommissionRate(
        uint256 validatorId,
        uint256 newCommissionRate,
        address bitkubNext
    ) external;

    function updateMinDelegated(
        uint256 validatorId,
        uint256 newMinDelegated
    ) external;

    function updateMinDelegated(
        uint256 validatorId,
        uint256 newMinDelegated,
        address bitkubNext
    ) external;

    function withdrawDelegatorsReward(
        uint256 validatorId
    ) external returns (uint256);

    function updateValidatorDelegation(
        uint256 validatorId,
        bool delegation
    ) external;

    function updateValidatorDelegation(
        uint256 validatorId,
        bool delegation,
        address bitkubNext
    ) external;

    function slash(address _signer) external returns (bool);

    function updateValidatorState(uint256 validatorId, int256 amount) external;

    function getMinimalValidators()
        external
        view
        returns (MinimalValidator[] memory);

    function getMinimalValidatorsByPage(
        uint256 page_,
        uint256 limit_
    ) external view returns (MinimalValidator[] memory);

    function getMinimalValidatorsLength() external view returns (uint256);

    function getValidatorId(address _signer) external view returns (uint256);

    function getMinimalValidatorIndex(
        address _signer
    ) external view returns (uint256);

    function getValidators() external view returns (address[] memory);
}
```

```
// note this contract interface is only for stakeManager use
interface IValidatorShare {
    function delegate() external payable returns (uint256 amountToDeposit);

    function undelegate(uint256 claimAmount) external returns (uint256);

    function claimRewards() external returns (uint256);

    function updateDelegation(bool _delegation) external;

    function updateMinDelegated(uint256 _minDelegated) external;

    function getLiquidRewards(address user) external view returns (uint256);
}

// * Copyright 2023 Bitkub Blockchain Technology Co., Ltd. - All Rights Reserved.
// * This code is proprietary.
// * Unauthorized copying, modification or distribution of this code, via any
medium, is strictly prohibited without express permission.

// * Copyright 2023 Bitkub Blockchain Technology Co., Ltd. - All Rights Reserved.
// * This code is proprietary.
// * Unauthorized copying, modification or distribution of this code, via any
medium, is strictly prohibited without express permission.

interface IStakeManagerStorage {
    function stakeManager() external view returns (address);

    function nftContract() external view returns (address);

    function validatorShareFactory() external view returns (address);

    function slashManager() external view returns (address);

    function soloLimit() external view returns (uint256);

    function soloAmount() external view returns (uint256);

    function poolLimit() external view returns (uint256);

    function poolAmount() external view returns (uint256);

    function officialLimit() external view returns (uint256);

    function officialAmount() external view returns (uint256);
    function defaultInfraCommissionRate() external view returns (uint256);

    function totalStaked() external view returns (uint256);

    function totalRewards() external view returns (uint256);

    function totalRewardsLiquidated() external view returns (uint256);

    function soloSlashRate() external view returns (uint256);

    function poolSlashAmount() external view returns (uint256);

    function unallocatedReward() external view returns (uint256);

    function officialPool() external view returns (address);

    function getNewOfficialPoolValidBlock() external view returns (uint256);

    function getOldNewOfficialSignerAndValidBlock()
        external
        view
        returns (address, address, uint256);

    function changeOfficialSigner(
        address newOfficialSigner_,
        string memory checksummedNewOfficialSigner_
    ) external;

    function setStakeManager(address stakeManager_) external;

    function setValidatorShareFactory(address validatorShareFactory_) external;

    function setSlashManager(address slashManager_) external;

    function setSoloLimit(uint256 val_) external;

    function incDescSoloLimit(uint256 val_, ChangeMode mode_) external;

    function setSoloAmount(uint256 val_) external;

    function incDescSoloAmount(uint256 val_, ChangeMode mode_) external;

    function setPoolLimit(uint256 val_) external;
    function incDescPoolLimit(uint256 val_, ChangeMode mode_) external;

    function setPoolAmount(uint256 val_) external;

    function incDescPoolAmount(uint256 val_, ChangeMode mode_) external;

    function setOfficialLimit(uint256 val_) external;

    function incDescOfficialLimit(uint256 val_, ChangeMode mode_) external;

    function setOfficialAmount(uint256 val_) external;

    function incDescOfficialAmount(uint256 val_, ChangeMode mode_) external;

    function setDefaultInfraCommissionRate(uint256 val_) external;

    function incDescDefaultInfraCommissionRate(
        uint256 val_,
        ChangeMode mode_
    ) external;

    function setTotalStaked(uint256 val_) external;

    function incDescTotalStaked(uint256 val_, ChangeMode mode_) external;

    function setTotalRewards(uint256 val_) external;

    function incDescTotalRewards(uint256 val_, ChangeMode mode_) external;

    function setTotalRewardsLiquidated(uint256 val_) external;

    function incDescTotalRewardsLiquidated(
        uint256 val_,
        ChangeMode mode_
    ) external;

    function setPoolSlashAmount(uint256 val_) external;

    function incDescPoolSlashAmount(uint256 val_, ChangeMode mode_) external;

    function setUnallocatedReward(uint256 val_) external;

    function incDescUnallocatedReward(uint256 val_, ChangeMode mode_) external;

    function addValidator(
        address key_,
        Validator memory val_
    ) external returns (uint256);

    function setValidatorIds(
        address key_,
        uint256[] memory validatorIds_
    ) external;

    function setValidatorInfo(address key_, Validator memory val_) external;

    function setValidatorInfoByIndex(
        uint256 index_,
        Validator memory val_
    ) external;

    function setValidatorInfoMinDeposit(address key_, uint256 val_) external;

    function setValidatorInfoMinDepositByIndex(
        uint256 index_,
        uint256 val_
    ) external;

    function incDescValidatorInfoMinDeposit(
        address key_,
        uint256 val_,
        ChangeMode mode_
    ) external;

    function incDescValidatorInfoMinDepositByIndex(
        uint256 index_,
        uint256 val_,
        ChangeMode mode_
    ) external;

    function setValidatorInfoAmount(address key_, uint256 val_) external;

    function setValidatorInfoAmountByIndex(
        uint256 index_,
        uint256 val_
    ) external;

    function incDescValidatorInfoAmount(
        address key_,
        uint256 val_,
        ChangeMode mode_
    ) external;

    function incDescValidatorInfoAmountByIndex(
        uint256 index_,
        uint256 val_,
        ChangeMode mode_
    ) external;

    function setValidatorInfoReward(address key_, uint256 val_) external;

    function setValidatorInfoRewardByIndex(
        uint256 index_,
        uint256 val_
    ) external;

    function incDescValidatorInfoReward(
        address key_,
        uint256 val_,
        ChangeMode mode_
    ) external;

    function incDescValidatorInfoRewardByIndex(
        uint256 index_,
        uint256 val_,
        ChangeMode mode_
    ) external;

    function setValidatorInfoSigner(address key_, address val_) external;

    function setValidatorInfoSignerByIndex(
        uint256 index_,
        address val_
    ) external;

    function setValidatorInfoValidatorShareContract(
        address key_,
        address val_
    ) external;

    function setValidatorInfoValidatorShareContractByIndex(
        uint256 index_,
        address val_
    ) external;
    function setValidatorInfoStatus(address key_, Status val_) external;

    function setValidatorInfoStatusByIndex(
        uint256 index_,
        Status val_
    ) external;

    function setValidatorInfoInfraCommissionRate(
        address key_,
        uint256 val_
    ) external;

    function setValidatorInfoInfraCommissionRateByIndex(
        uint256 index_,
        uint256 val_
    ) external;

    function incDescValidatorInfoInfraCommissionRate(
        address key_,
        uint256 val_,
        ChangeMode mode_
    ) external;

    function incDescValidatorInfoInfraCommissionRateByIndex(
        uint256 index_,
        uint256 val_,
        ChangeMode mode_
    ) external;

    function setValidatorInfoInfraCommissionAmount(
        address key_,
        uint256 val_
    ) external;

    function setValidatorInfoInfraCommissionAmountByIndex(
        uint256 index_,
        uint256 val_
    ) external;

    function incDescValidatorInfoInfraCommissionAmount(
        address key_,
        uint256 val_,
        ChangeMode mode_
    ) external;
    function incDescValidatorInfoInfraCommissionAmountByIndex(
        uint256 index_,
        uint256 val_,
        ChangeMode mode_
    ) external;

    function setValidatorInfoCommissionRate(
        address key_,
        uint256 val_
    ) external;

    function setValidatorInfoCommissionRateByIndex(
        uint256 index_,
        uint256 val_
    ) external;

    function incDescValidatorInfoCommissionRate(
        address key_,
        uint256 val_,
        ChangeMode mode_
    ) external;

    function incDescValidatorInfoCommissionRateByIndex(
        uint256 index_,
        uint256 val_,
        ChangeMode mode_
    ) external;

    function setValidatorInfoValidatorCommissionAmount(
        address key_,
        uint256 val_
    ) external;

    function setValidatorInfoValidatorCommissionAmountByIndex(
        uint256 index_,
        uint256 val_
    ) external;

    function incDescValidatorInfoValidatorCommissionAmount(
        address key_,
        uint256 val_,
        ChangeMode mode_
    ) external;

    function incDescValidatorInfoValidatorCommissionAmountByIndex(
        uint256 index_,
        uint256 val_,
        ChangeMode mode_
    ) external;

    function setValidatorInfoDelegatorCommissionAmount(
        address key_,
        uint256 val_
    ) external;

    function setValidatorInfoDelegatorCommissionAmountByIndex(
        uint256 index_,
        uint256 val_
    ) external;

    function incDescValidatorInfoDelegatorCommissionAmount(
        address key_,
        uint256 val_,
        ChangeMode mode_
    ) external;

    function incDescValidatorInfoDelegatorCommissionAmountByIndex(
        uint256 index_,
        uint256 val_,
        ChangeMode mode_
    ) external;

    function setValidatorInfoDelegatorsReward(
        address key_,
        uint256 val_
    ) external;

    function setValidatorInfoDelegatorsRewardByIndex(
        uint256 index_,
        uint256 val_
    ) external;

    function incDescValidatorInfoDelegatorsReward(
        address key_,
        uint256 val_,
        ChangeMode mode_
    ) external;

    function incDescValidatorInfoDelegatorsRewardByIndex(
        uint256 index_,
        uint256 val_,
        ChangeMode mode_
    ) external;

    function setValidatorInfoDelegatedAmount(
        address key_,
        uint256 val_
    ) external;

    function setValidatorInfoDelegatedAmountByIndex(
        uint256 index_,
        uint256 val_
    ) external;

    function incDescValidatorInfoDelegatedAmount(
        address key_,
        uint256 val_,
        ChangeMode mode_
    ) external;

    function incDescValidatorInfoDelegatedAmountByIndex(
        uint256 index_,
        uint256 val_,
        ChangeMode mode_
    ) external;

    function getValidatorInfo(
        address key_
    ) external view returns (Validator memory);

    function getValidatorInfoByIndex(
        uint256 index_
    ) external view returns (Validator memory);

    function getValidatorInfoMinDeposit(
        address key_
    ) external view returns (uint256);

    function getValidatorInfoMinDepositByIndex(
        uint256 index_
    ) external view returns (uint256);

    function getValidatorInfoAmount(
        address key_
    ) external view returns (uint256);
    function getValidatorInfoAmountByIndex(
        uint256 index_
    ) external view returns (uint256);

    function getValidatorInfoReward(
        address key_
    ) external view returns (uint256);

    function getValidatorInfoRewardByIndex(
        uint256 index_
    ) external view returns (uint256);

    function getValidatorInfoSigner(
        address key_
    ) external view returns (address);

    function getValidatorInfoSignerByIndex(
        uint256 index_
    ) external view returns (address);

    function getValidatorInfoValidatorShareContract(
        address key_
    ) external view returns (address);

    function getValidatorInfoValidatorShareContractByIndex(
        uint256 index_
    ) external view returns (address);

    function getValidatorInfoStatus(
        address key_
    ) external view returns (Status);

    function getValidatorInfoStatusByIndex(
        uint256 index_
    ) external view returns (Status);

    function getValidatorInfoInfraCommissionRate(
        address key_
    ) external view returns (uint256);

    function getValidatorInfoInfraCommissionRateByIndex(
        uint256 index_
    ) external view returns (uint256);
    function getValidatorInfoInfraCommissionAmount(
        address key_
    ) external view returns (uint256);

    function getValidatorInfoInfraCommissionAmountByIndex(
        uint256 index_
    ) external view returns (uint256);

    function getValidatorInfoCommissionRate(
        address key_
    ) external view returns (uint256);

    function getValidatorInfoCommissionRateByIndex(
        uint256 index_
    ) external view returns (uint256);

    function getValidatorInfoValidatorCommissionAmount(
        address key_
    ) external view returns (uint256);

    function getValidatorInfoValidatorCommissionAmountByIndex(
        uint256 index_
    ) external view returns (uint256);

    function getValidatorInfoDelegatorCommissionAmount(
        address key_
    ) external view returns (uint256);

    function getValidatorInfoDelegatorCommissionAmountByIndex(
        uint256 index_
    ) external view returns (uint256);

    function getValidatorInfoDelegatorsReward(
        address key_
    ) external view returns (uint256);

    function getValidatorInfoDelegatorsRewardByIndex(
        uint256 index_
    ) external view returns (uint256);

    function getValidatorInfoDelegatedAmount(
        address key_
    ) external view returns (uint256);

    function getValidatorInfoDelegatedAmountByIndex(
        uint256 index_
    ) external view returns (uint256);

    function getValidatorListLength() external view returns (uint256);

    function isInValidatorList(address val_) external view returns (bool);

    function getValidatorCurrentIndex(
        address val_,
        bool revertIfNotFound_
    ) external view returns (uint256, bool);

    function getValidatorByIndex(
        uint256 index_
    ) external view returns (address);

    function getAllValidator() external view returns (address[] memory);

    function getValidatorByPage(
        uint256 page,
        uint256 limit
    ) external view returns (address[] memory);

    function getValidatorIndexLength(
        address val_
    ) external view returns (uint256);

    function getValidatorIndexByIndex(
        address val_,
        uint256 index_
    ) external view returns (uint256);

    function getMinimalValidatorListLength() external view returns (uint256);

    function isInMinimalValidatorList(
        address val_
    ) external view returns (bool);

    function isInMinimalValidatorListByValidatorId(
        uint256 validatorId_
    ) external view returns (bool);

    function getMinimalValidatorIndex(
        address val_,
        bool revertIfNotFound_
    ) external view returns (uint256, bool);

    function getMinimalValidatorByIndex(
        uint256 index_
    ) external view returns (address);

    function getAllMinimalValidators() external view returns (address[] memory);

    function getMinimalValidatorsByPage(
        uint256 page,
        uint256 limit
    ) external view returns (address[] memory);

    function addMinimalValidator(address val_) external;

    function removeMinimalValidator(address val_) external;

    function removeMinimalValidatorByIndex(uint256 index_) external;

    function getMinimalValidatorsWithValidatorPowerByPage(
        uint256 page_,
        uint256 limit_
    ) external view returns (address[] memory, uint256[] memory);
}

interface IKAP165 {
    function supportsInterface(bytes4 interfaceId) external view returns (bool);
}

abstract contract KAP165 is IKAP165 {
    function supportsInterface(
        bytes4 interfaceId
    ) public view virtual override returns (bool) {
        return interfaceId == type(IKAP165).interfaceId;
    }
}

abstract contract Committee {
    address public committee;

    event CommitteeSet(
        address indexed oldCommittee,
        address indexed newCommittee,
        address indexed caller
    );
    function _onlyCommittee() internal view {
        require(
            msg.sender == committee,
            "Committee: restricted only committee"
        );
    }

    modifier onlyCommittee() {
        _onlyCommittee();
        _;
    }

    constructor(address committee_) {
        committee = committee_;
    }

    function setCommittee(address _committee) public virtual onlyCommittee {
        emit CommitteeSet(committee, _committee, msg.sender);
        committee = _committee;
    }
}

abstract contract Context {
    function _msgSender() internal view virtual returns (address) {
        return msg.sender;
    }

    function _msgData() internal view virtual returns (bytes calldata) {
        return msg.data;
    }
}

abstract contract Pausable is Context {
    bool private _paused;

    event Paused(address account);

    event Unpaused(address account);

    modifier whenNotPaused() {
        require(!paused(), "Pausable: paused");
        _;
    }
    modifier whenPaused() {
        require(paused(), "Pausable: not paused");
        _;
    }

    constructor() {
        _paused = false;
    }

    function paused() public view virtual returns (bool) {
        return _paused;
    }

    function _pause() internal virtual whenNotPaused {
        _paused = true;
        emit Paused(_msgSender());
    }

    function _unpause() internal virtual whenPaused {
        _paused = false;
        emit Unpaused(_msgSender());
    }
}

interface IKAP721 is IKAP165 {
    event Transfer(
        address indexed from,
        address indexed to,
        uint256 indexed tokenId
    );
    event AdminTransfer(
        address indexed from,
        address indexed to,
        uint256 indexed tokenId
    );
    event Approval(
        address indexed owner,
        address indexed approved,
        uint256 indexed tokenId
    );
    event ApprovalForAll(
        address indexed owner,
        address indexed operator,
        bool approved
    );
    function balanceOf(address owner) external view returns (uint256 balance);

    function ownerOf(uint256 tokenId) external view returns (address owner);

    function safeTransferFrom(
        address from,
        address to,
        uint256 tokenId
    ) external;

    function transferFrom(address from, address to, uint256 tokenId) external;

    function adminTransfer(address from, address to, uint256 tokenId) external;

    function internalTransfer(
        address sender,
        address recipient,
        uint256 tokenId
    ) external returns (bool);

    function externalTransfer(
        address sender,
        address recipient,
        uint256 tokenId
    ) external returns (bool);

    function approve(address to, uint256 tokenId) external;

    function getApproved(
        uint256 tokenId
    ) external view returns (address operator);

    function setApprovalForAll(address operator, bool _approved) external;

    function isApprovedForAll(
        address owner,
        address operator
    ) external view returns (bool);

    function safeTransferFrom(
        address from,
        address to,
        uint256 tokenId,
        bytes calldata data
    ) external;
}

interface IKAP721V2 {
    function tokenOfOwnerByPage(
        address owner,
        uint256 page,
        uint256 limit
    ) external view returns (uint256[] memory);

    function tokenOfOwnerAll(
        address owner
    ) external view returns (uint256[] memory);
}

interface IKAP721Receiver {
    function onKAP721Received(
        address operator,
        address from,
        uint256 tokenId,
        bytes calldata data
    ) external returns (bytes4);
}

interface IKAP721Metadata {
    event Uri(uint256 indexed id);

    function name() external view returns (string memory);

    function symbol() external view returns (string memory);

    function tokenURI(uint256 tokenId) external view returns (string memory);
}

interface IKAP721Enumerable {
    function totalSupply() external view returns (uint256);

    function tokenOfOwnerByIndex(
        address owner,
        uint256 index
    ) external view returns (uint256 tokenId);

    function tokenByIndex(uint256 index) external view returns (uint256);
}
```

```
interface IKYCBitkubChain {
    function setKycCompleted(address _addr, uint256 _level) external;

    function kycsLevel(address _addr) external view returns (uint256);
}

library Address {
    function isContract(address account) internal view returns (bool) {
        // This method relies on extcodesize, which returns 0 for contracts in
        // construction, since the code is only stored at the end of the
        // constructor execution.

        uint256 size;
        assembly {
            size := extcodesize(account)
        }
        return size > 0;
    }

    function sendValue(address payable recipient, uint256 amount) internal {
        require(
            address(this).balance >= amount,
            "Address: insufficient balance"
        );

        (bool success, ) = recipient.call{ value: amount }("");
        require(
            success,
            "Address: unable to send value, recipient may have reverted"
        );
    }

    function functionCall(
        address target,
        bytes memory data
    ) internal returns (bytes memory) {
        return functionCall(target, data, "Address: low-level call failed");
    }

    function functionCall(
        address target,
        bytes memory data,
        string memory errorMessage
    ) internal returns (bytes memory) {
        return functionCallWithValue(target, data, 0, errorMessage);
    }

    function functionCallWithValue(
        address target,
        bytes memory data,
        uint256 value
    ) internal returns (bytes memory) {
        return
            functionCallWithValue(
                target,
                data,
                value,
                "Address: low-level call with value failed"
            );
    }

    function functionCallWithValue(
        address target,
        bytes memory data,
        uint256 value,
        string memory errorMessage
    ) internal returns (bytes memory) {
        require(
            address(this).balance >= value,
            "Address: insufficient balance for call"
        );
        require(isContract(target), "Address: call to non-contract");

        (bool success, bytes memory returndata) = target.call{ value: value }(
            data
        );
        return verifyCallResult(success, returndata, errorMessage);
    }

    function functionStaticCall(
        address target,
        bytes memory data
    ) internal view returns (bytes memory) {
        return
            functionStaticCall(
                target,
                data,
                "Address: low-level static call failed"
            );
    }
    function functionStaticCall(
        address target,
        bytes memory data,
        string memory errorMessage
    ) internal view returns (bytes memory) {
        require(isContract(target), "Address: static call to non-contract");

        (bool success, bytes memory returndata) = target.staticcall(data);
        return verifyCallResult(success, returndata, errorMessage);
    }

    function functionDelegateCall(
        address target,
        bytes memory data
    ) internal returns (bytes memory) {
        return
            functionDelegateCall(
                target,
                data,
                "Address: low-level delegate call failed"
            );
    }

    function functionDelegateCall(
        address target,
        bytes memory data,
        string memory errorMessage
    ) internal returns (bytes memory) {
        require(isContract(target), "Address: delegate call to non-contract");

        (bool success, bytes memory returndata) = target.delegatecall(data);
        return verifyCallResult(success, returndata, errorMessage);
    }

    function verifyCallResult(
        bool success,
        bytes memory returndata,
        string memory errorMessage
    ) internal pure returns (bytes memory) {
        if (success) {
            return returndata;
        } else {
            // Look for revert reason and bubble it up if present
            if (returndata.length > 0) {
                // The easiest way to bubble the revert reason is using memory via
assembly

                assembly {
                    let returndata_size := mload(returndata)
                    revert(add(32, returndata), returndata_size)
                }
            } else {
                revert(errorMessage);
            }
        }
    }
}

library EnumerableSetUint {
    struct UintSet {
        uint256[] _values;
        mapping(uint256 => uint256) _indexes;
    }

    function add(UintSet storage set, uint256 value) internal returns (bool) {
        if (!contains(set, value)) {
            set._values.push(value);
            set._indexes[value] = set._values.length;
            return true;
        } else {
            return false;
        }
    }

    function remove(
        UintSet storage set,
        uint256 value
    ) internal returns (bool) {
        uint256 valueIndex = set._indexes[value];
        if (valueIndex != 0) {
            uint256 toDeleteIndex = valueIndex - 1;
            uint256 lastIndex = set._values.length - 1;
            uint256 lastvalue = set._values[lastIndex];
            set._values[toDeleteIndex] = lastvalue;
            set._indexes[lastvalue] = toDeleteIndex + 1;
            set._values.pop();
            delete set._indexes[value];
            return true;
        } else {
            return false;
        }
    }

    function contains(
        UintSet storage set,
        uint256 value
    ) internal view returns (bool) {
        return set._indexes[value] != 0;
    }

    function length(UintSet storage set) internal view returns (uint256) {
        return set._values.length;
    }

    function at(
        UintSet storage set,
        uint256 index
    ) internal view returns (uint256) {
        require(
            set._values.length > index,
            "EnumerableSet: index out of bounds"
        );
        return set._values[index];
    }

    function index(
        UintSet storage set,
        uint256 value
    ) internal view returns (uint256, bool) {
        if (!contains(set, value)) {
            return (type(uint256).max, false);
        }
        return (set._indexes[value] - 1, true);
    }

    function getAll(
        UintSet storage set
    ) internal view returns (uint256[] memory) {
        return set._values;
    }

    function get(
        UintSet storage set,
        uint256 _page,
        uint256 _limit
    ) internal view returns (uint256[] memory) {
        require(_page > 0 && _limit > 0);
        uint256 tempLength = _limit;
        uint256 cursor = (_page - 1) * _limit;
        uint256 _uintLength = length(set);
        if (cursor >= _uintLength) {
            return new uint256[](0);
        }
        if (tempLength > _uintLength - cursor) {
            tempLength = _uintLength - cursor;
        }
        uint256[] memory uintList = new uint256[](tempLength);
        for (uint256 i = 0; i < tempLength; i++) {
            uintList[i] = at(set, cursor + i);
        }
        return uintList;
    }
}

library EnumerableMap {
    struct MapEntry {
        bytes32 _key;
        bytes32 _value;
    }

    struct Map {
        MapEntry[] _entries;
        mapping(bytes32 => uint256) _indexes;
    }

    function _set(
        Map storage map,
        bytes32 key,
        bytes32 value
    ) private returns (bool) {
        // We read and store the key's index to prevent multiple reads from the same
storage slot
        uint256 keyIndex = map._indexes[key];

        if (keyIndex == 0) {
            // Equivalent to !contains(map, key)
            map._entries.push(MapEntry({ _key: key, _value: value }));
            // The entry is stored at length-1, but we add 1 to all indexes
            // and use 0 as a sentinel value
            map._indexes[key] = map._entries.length;
            return true;
        } else {
            map._entries[keyIndex - 1]._value = value;
            return false;
        }
    }

    function _remove(Map storage map, bytes32 key) private returns (bool) {
        // We read and store the key's index to prevent multiple reads from the same
storage slot
        uint256 keyIndex = map._indexes[key];

        if (keyIndex != 0) {
            // Equivalent to contains(map, key)
            // To delete a key-value pair from the _entries array in O(1), we swap
the entry to delete with the last one
            // in the array, and then remove the last entry (sometimes called as
'swap and pop').
            // This modifies the order of the array, as noted in {at}.

            uint256 toDeleteIndex = keyIndex - 1;
            uint256 lastIndex = map._entries.length - 1;

            // When the entry to delete is the last one, the swap operation is
unnecessary. However, since this occurs
            // so rarely, we still do the swap anyway to avoid the gas cost of
adding an 'if' statement.

            MapEntry storage lastEntry = map._entries[lastIndex];

            // Move the last entry to the index where the entry to delete is
            map._entries[toDeleteIndex] = lastEntry;
            // Update the index for the moved entry
            map._indexes[lastEntry._key] = toDeleteIndex + 1; // All indexes are
1-based

            // Delete the slot where the moved entry was stored
            map._entries.pop();

            // Delete the index for the deleted slot
            delete map._indexes[key];

            return true;
        } else {
            return false;
        }
    }

    function _contains(
        Map storage map,
        bytes32 key
    ) private view returns (bool) {
        return map._indexes[key] != 0;
    }

    function _length(Map storage map) private view returns (uint256) {
        return map._entries.length;
    }

    function _at(
        Map storage map,
        uint256 index
    ) private view returns (bytes32, bytes32) {
        require(
            map._entries.length > index,
            "EnumerableMap: index out of bounds"
        );

        MapEntry storage entry = map._entries[index];
        return (entry._key, entry._value);
    }

    function _tryGet(
        Map storage map,
        bytes32 key
    ) private view returns (bool, bytes32) {
        uint256 keyIndex = map._indexes[key];
        if (keyIndex == 0) return (false, 0); // Equivalent to contains(map, key)
        return (true, map._entries[keyIndex - 1]._value); // All indexes are 1-based
    }

    function _get(Map storage map, bytes32 key) private view returns (bytes32) {
        uint256 keyIndex = map._indexes[key];
        require(keyIndex != 0, "EnumerableMap: nonexistent key"); // Equivalent to
contains(map, key)
        return map._entries[keyIndex - 1]._value; // All indexes are 1-based
    }

    function _get(
        Map storage map,
        bytes32 key,
        string memory errorMessage
    ) private view returns (bytes32) {
        uint256 keyIndex = map._indexes[key];
        require(keyIndex != 0, errorMessage); // Equivalent to contains(map, key)
        return map._entries[keyIndex - 1]._value; // All indexes are 1-based
    }

    // UintToAddressMap

    struct UintToAddressMap {
        Map _inner;
    }

    function set(
        UintToAddressMap storage map,
        uint256 key,
        address value
    ) internal returns (bool) {
        return _set(map._inner, bytes32(key), bytes32(uint256(uint160(value))));
    }

    function remove(
        UintToAddressMap storage map,
        uint256 key
    ) internal returns (bool) {
        return _remove(map._inner, bytes32(key));
    }

    function contains(
        UintToAddressMap storage map,
        uint256 key
    ) internal view returns (bool) {
        return _contains(map._inner, bytes32(key));
    }

    function length(
        UintToAddressMap storage map
    ) internal view returns (uint256) {
        return _length(map._inner);
    }

    function at(
        UintToAddressMap storage map,
        uint256 index
    ) internal view returns (uint256, address) {
        (bytes32 key, bytes32 value) = _at(map._inner, index);
        return (uint256(key), address(uint160(uint256(value))));
    }

    function tryGet(
        UintToAddressMap storage map,
        uint256 key
    ) internal view returns (bool, address) {
        (bool success, bytes32 value) = _tryGet(map._inner, bytes32(key));
        return (success, address(uint160(uint256(value))));
    }

    function get(
        UintToAddressMap storage map,
        uint256 key
    ) internal view returns (address) {
        return address(uint160(uint256(_get(map._inner, bytes32(key)))));
    }

    function get(
        UintToAddressMap storage map,
        uint256 key,
        string memory errorMessage
    ) internal view returns (address) {
        return
            address(
                uint160(uint256(_get(map._inner, bytes32(key), errorMessage)))
            );
    }
}

library Strings {
    bytes16 private constant _HEX_SYMBOLS = "0123456789abcdef";

    function toString(uint256 value) internal pure returns (string memory) {
        if (value == 0) {
            return "0";
        }
        uint256 temp = value;
        uint256 digits;
        while (temp != 0) {
            digits++;
            temp /= 10;
        }
        bytes memory buffer = new bytes(digits);
        while (value != 0) {
            digits -= 1;
            buffer[digits] = bytes1(uint8(48 + uint256(value % 10)));
            value /= 10;
        }
        return string(buffer);
    }

    function toHexString(uint256 value) internal pure returns (string memory) {
        if (value == 0) {
            return "0x00";
        }
        uint256 temp = value;
        uint256 length = 0;
        while (temp != 0) {
            length++;
            temp >>= 8;
        }
        return toHexString(value, length);
    }

    function toHexString(
        uint256 value,
        uint256 length
    ) internal pure returns (string memory) {
        bytes memory buffer = new bytes(2 * length + 2);
        buffer[0] = "0";
        buffer[1] = "x";
        for (uint256 i = 2 * length + 1; i > 1; --i) {
            buffer[i] = _HEX_SYMBOLS[value & 0xf];
            value >>= 4;
        }
        require(value == 0, "Strings: hex length insufficient");
        return string(buffer);
    }
}

abstract contract KAP721 is
    KAP165,
    IKAP721,
    IKAP721V2,
    IKAP721Metadata,
    IKAP721Enumerable,
    Committee,
    Pausable
{
    using Address for address;
    using EnumerableSetUint for EnumerableSetUint.UintSet;
    using EnumerableMap for EnumerableMap.UintToAddressMap;
    using Strings for uint256;

    // Mapping from holder address to their (enumerable) set of owned tokens
    mapping(address => EnumerableSetUint.UintSet) _holderTokens;

    // Enumerable mapping from token ids to their owners
    EnumerableMap.UintToAddressMap private _tokenOwners;

    // Mapping from token ID to approved address
    mapping(uint256 => address) private _tokenApprovals;

    // Mapping from owner to operator approvals
    mapping(address => mapping(address => bool)) private _operatorApprovals;

    // Token name
    string public override name;

    // Token symbol
    string public override symbol;

    // Optional mapping for token URIs
    mapping(uint256 => string) private _tokenURIs;

    // Base URI
    string public baseURI;

    /*
     *     bytes4(keccak256('balanceOf(address)')) == 0x70a08231
     *     bytes4(keccak256('ownerOf(uint256)')) == 0x6352211e
     *     bytes4(keccak256('approve(address,uint256)')) == 0x095ea7b3
     *     bytes4(keccak256('getApproved(uint256)')) == 0x081812fc
     *     bytes4(keccak256('setApprovalForAll(address,bool)')) == 0xa22cb465
     *     bytes4(keccak256('isApprovedForAll(address,address)')) == 0xe985e9c5
     *     bytes4(keccak256('transferFrom(address,address,uint256)')) == 0x23b872dd
     *     bytes4(keccak256('safeTransferFrom(address,address,uint256)')) ==
0x42842e0e
     *     bytes4(keccak256('safeTransferFrom(address,address,uint256,bytes)')) ==
0xb88d4fde
     *
     *     => 0x70a08231 ^ 0x6352211e ^ 0x095ea7b3 ^ 0x081812fc ^
     *        0xa22cb465 ^ 0xe985e9c5 ^ 0x23b872dd ^ 0x42842e0e ^ 0xb88d4fde ==
0x80ac58cd
     */
    bytes4 private constant _INTERFACE_ID_ERC721 = 0x80ac58cd;

    //

    address public transferRouter;

    event TransferRouterSet(
        address indexed oldTransferRouter,
        address indexed newTransferRouter,
        address indexed caller
    );

    modifier onlyTransferRouter() {
        require(
            msg.sender == transferRouter,
            "KAP721: restricted only transfer router"
        );
        _;
    }

    //

    constructor(
        string memory _name,
        string memory _symbol,
        string memory _baseURI,
        address _committee,
        address _transferRouter
    ) Committee(_committee) {
        name = _name;
        symbol = _symbol;
        baseURI = _baseURI;
        transferRouter = _transferRouter;
    }

    function supportsInterface(
        bytes4 interfaceId
    ) public view virtual override(KAP165, IKAP165) returns (bool) {
        return
            interfaceId == _INTERFACE_ID_ERC721 ||
            interfaceId == type(IKAP721).interfaceId ||
            interfaceId == type(IKAP721Metadata).interfaceId ||
            interfaceId == type(IKAP721Enumerable).interfaceId ||
            super.supportsInterface(interfaceId);
    }

    function balanceOf(
        address owner
    ) public view virtual override returns (uint256) {
        require(
            owner != address(0),
            "KAP721: Balance query for the zero address"
        );
        return _holderTokens[owner].length();
    }

    function ownerOf(
        uint256 tokenId
    ) public view virtual override returns (address) {
        return
            _tokenOwners.get(
                tokenId,
                "KAP721: Owner query for nonexistent token"
            );
    }

    function tokenURI(
        uint256 tokenId
    ) public view virtual override returns (string memory) {
        require(
            _exists(tokenId),
            "KAP721Metadata: URI query for nonexistent token"
        );

        string memory _tokenURI = _tokenURIs[tokenId];
        string memory base = baseURI;

        // If there is no base URI, return the token URI.
        if (bytes(base).length == 0) {
            return _tokenURI;
        }
        // If both are set, concatenate the baseURI and tokenURI (via
abi.encodePacked).
        if (bytes(_tokenURI).length > 0) {
            return string(abi.encodePacked(base, _tokenURI));
        }
        // If there is a baseURI but no tokenURI, concatenate the tokenID to the
baseURI.
        return string(abi.encodePacked(base, tokenId.toString()));
    }

    function tokenOfOwnerByPage(
        address _owner,
        uint256 _page,
        uint256 _limit
    ) public view virtual override returns (uint256[] memory) {
        return _holderTokens[_owner].get(_page, _limit);
    }

    function tokenOfOwnerAll(
        address _owner
    ) public view virtual override returns (uint256[] memory) {
        return _holderTokens[_owner].getAll();
    }

    function tokenOfOwnerByIndex(
        address owner,
        uint256 index
    ) public view virtual override returns (uint256) {
        return _holderTokens[owner].at(index);
    }

    function totalSupply() public view virtual override returns (uint256) {
        // _tokenOwners are indexed by tokenIds, so .length() returns the number of
tokenIds
        return _tokenOwners.length();
    }

    function tokenByIndex(
        uint256 index
    ) public view virtual override returns (uint256) {
        (uint256 tokenId, ) = _tokenOwners.at(index);
        return tokenId;
    }

    function approve(
        address to,
        uint256 tokenId
    ) public virtual override whenNotPaused {
        address owner = KAP721.ownerOf(tokenId);
        require(to != owner, "KAP721: Approval to current owner");
        require(
            msg.sender == owner || isApprovedForAll(owner, msg.sender),
            "KAP721: Approve caller is not owner nor approved for all"
        );

        _approve(to, tokenId);
    }

    function getApproved(
        uint256 tokenId
    ) public view virtual override returns (address) {
        require(
            _exists(tokenId),
            "KAP721: Approved query for nonexistent token"
        );

        return _tokenApprovals[tokenId];
    }

    function setApprovalForAll(
        address operator,
        bool approved
    ) public virtual override whenNotPaused {
        require(operator != msg.sender, "KAP721: Approve to caller");

        _setApprovalForAll(msg.sender, operator, approved);
    }

    function isApprovedForAll(
        address owner,
        address operator
    ) public view virtual override returns (bool) {
        return _operatorApprovals[owner][operator];
    }

    function transferFrom(
        address from,
        address to,
        uint256 tokenId
    ) public virtual override whenNotPaused {
        require(
            _isApprovedOrOwner(msg.sender, tokenId),
            "KAP721: Transfer caller is not owner nor approved"
        );
        _transfer(from, to, tokenId);
    }

    function adminTransfer(
        address _from,
        address _to,
        uint256 _tokenId
    ) public virtual override onlyCommittee {
        _adminTransfer(_from, _to, _tokenId);
    }

    function _adminTransfer(
        address _from,
        address _to,
        uint256 _tokenId
    ) internal virtual {
        require(
            ownerOf(_tokenId) == _from,
            "KAP721: Transfer of token that is not own"
        ); // internal owner

        // Clear approvals from the previous owner
        _approve(address(0), _tokenId);

        _holderTokens[_from].remove(_tokenId);
        _holderTokens[_to].add(_tokenId);

        _tokenOwners.set(_tokenId, _to);

        emit Transfer(_from, _to, _tokenId);
        emit AdminTransfer(_from, _to, _tokenId);
    }

    function internalTransfer(
        address sender,
        address recipient,
        uint256 tokenId
    ) external override onlyTransferRouter whenNotPaused returns (bool) {
        revert("KAP721: disabled");
    }

    function externalTransfer(
        address sender,
        address recipient,
        uint256 tokenId
    ) external override onlyTransferRouter whenNotPaused returns (bool) {
        revert("KAP721: disabled");
    }

    function safeTransferFrom(
        address from,
        address to,
        uint256 tokenId
    ) public virtual override whenNotPaused {
        safeTransferFrom(from, to, tokenId, "");
    }

    function safeTransferFrom(
        address from,
        address to,
        uint256 tokenId,
        bytes memory _data
    ) public virtual override whenNotPaused {
        require(
            _isApprovedOrOwner(msg.sender, tokenId),
            "KAP721: Transfer caller is not owner nor approved"
        );
        _safeTransfer(from, to, tokenId, _data);
    }

    function _safeTransfer(
        address from,
        address to,
        uint256 tokenId,
        bytes memory _data
    ) internal virtual {
        _transfer(from, to, tokenId);
        require(
            _checkOnKAP721Received(from, to, tokenId, _data),
            "KAP721: Transfer to non KAP721Receiver implementer"
        );
    }

    function _exists(uint256 tokenId) internal view virtual returns (bool) {
        return _tokenOwners.contains(tokenId);
    }

    function _isApprovedOrOwner(
        address spender,
        uint256 tokenId
    ) internal view virtual returns (bool) {
        require(
            _exists(tokenId),
            "KAP721: Operator query for nonexistent token"
        );
        address owner = KAP721.ownerOf(tokenId);
        return (spender == owner ||
            getApproved(tokenId) == spender ||
            isApprovedForAll(owner, spender));
    }

    function _safeMint(address to, uint256 tokenId) internal virtual {
        _safeMint(to, tokenId, "");
    }

    function _safeMint(
        address to,
        uint256 tokenId,
        bytes memory _data
    ) internal virtual {
        _mint(to, tokenId);
        require(
            _checkOnKAP721Received(address(0), to, tokenId, _data),
            "KAP721: Transfer to non KAP721Receiver implementer"
        );
    }

    function _mint(address to, uint256 tokenId) internal virtual {
        require(to != address(0), "KAP721: Mint to the zero address");
        require(!_exists(tokenId), "KAP721: Token already minted");

        _beforeTokenTransfer(address(0), to, tokenId);

        _holderTokens[to].add(tokenId);

        _tokenOwners.set(tokenId, to);

        emit Transfer(address(0), to, tokenId);
    }

    function _burn(uint256 tokenId) internal virtual {
        address owner = ownerOf(tokenId); // internal owner

        _beforeTokenTransfer(owner, address(0), tokenId);
        // Clear approvals
        _approve(address(0), tokenId);

        _holderTokens[owner].remove(tokenId);
        _holderTokens[address(0)].add(tokenId);

        _tokenOwners.set(tokenId, address(0));

        emit Transfer(owner, address(0), tokenId);
    }

    function _transfer(
        address from,
        address to,
        uint256 tokenId
    ) internal virtual {
        require(
            KAP721.ownerOf(tokenId) == from,
            "KAP721: Transfer of token that is not own"
        );
        require(to != address(0), "KAP721: Transfer to the zero address");

        _beforeTokenTransfer(from, to, tokenId);

        // Clear approvals from the previous owner
        _approve(address(0), tokenId);

        _holderTokens[from].remove(tokenId);
        _holderTokens[to].add(tokenId);

        _tokenOwners.set(tokenId, to);

        emit Transfer(from, to, tokenId);
    }

    function _setTokenURI(
        uint256 tokenId,
        string memory _tokenURI
    ) internal virtual {
        require(
            _exists(tokenId),
            "KAP721Metadata: URI set of nonexistent token"
        );
        _tokenURIs[tokenId] = _tokenURI;
    }

    function _setBaseURI(string memory baseURI_) internal virtual {
        baseURI = baseURI_;
    }

    function _checkOnKAP721Received(
        address from,
        address to,
        uint256 tokenId,
        bytes memory _data
    ) private returns (bool) {
        if (to.isContract()) {
            try
                IKAP721Receiver(to).onKAP721Received(
                    msg.sender,
                    from,
                    tokenId,
                    _data
                )
            returns (bytes4 retval) {
                return retval == IKAP721Receiver.onKAP721Received.selector;
            } catch (bytes memory reason) {
                if (reason.length == 0) {
                    revert(
                        "KAP721: Transfer to non KAP721Receiver implementer"
                    );
                } else {
                    assembly {
                        revert(add(32, reason), mload(reason))
                    }
                }
            }
        } else {
            return true;
        }
    }

    function _approve(address to, uint256 tokenId) internal virtual {
        _tokenApprovals[tokenId] = to;
        emit Approval(KAP721.ownerOf(tokenId), to, tokenId);
    }

    function _setApprovalForAll(
        address owner,
        address operator,
        bool approved
    ) internal virtual {
        _operatorApprovals[owner][operator] = approved;
        emit ApprovalForAll(owner, operator, approved);
    }

    function _beforeTokenTransfer(
        address from,
        address to,
        uint256 tokenId
    ) internal virtual {}
}

contract StakingNFT is KAP721 {
    event OfficialPoolCommitteeSet(
        address indexed oldOfficialPoolCommittee,
        address indexed newOfficialPoolCommittee,
        address indexed caller
    );

    IStakeManagerStorage public immutable stakeManagerStorage;
    address public officialPoolCommittee;

    modifier onlyOfficialPoolCommittee() {
        require(
            msg.sender == officialPoolCommittee,
            "StakingNFT: restricted only official pool committee"
        );
        _;
    }

    function _onlyStakeManager() private view {
        require(
            msg.sender == stakeManagerStorage.stakeManager(),
            "StakingNFT: restricted only stake manager"
        );
    }

    modifier onlyStakeManager() {
        _onlyStakeManager();
        _;
    }

    constructor(
        string memory _name,
        string memory _symbol,
        string memory _baseURI,
        address _committee,
        address _officialPoolCommittee,
        address _transferRouter,
        address _stakeManagerStorage
    ) KAP721(_name, _symbol, _baseURI, _committee, _transferRouter) {
        stakeManagerStorage = IStakeManagerStorage(_stakeManagerStorage);
        officialPoolCommittee = _officialPoolCommittee;
    }

    function setOfficialPoolCommittee(
        address _officialPoolCommittee
    ) external onlyOfficialPoolCommittee {
        emit OfficialPoolCommitteeSet(
            officialPoolCommittee,
            _officialPoolCommittee,
            msg.sender
        );
        officialPoolCommittee = _officialPoolCommittee;
    }

    function mint(address to, uint256 tokenId) public onlyStakeManager {
        _mint(to, tokenId);
    }

    function burn(uint256 tokenId) public onlyStakeManager {
        _burn(tokenId);
    }

    // restrict all transfer functions except adminTransfer

    function _transfer(
        address from,
        address to,
        uint256 tokenId
    ) internal override {
        revert("StakingNFT: _transfer is not allowed in this contract");
    }

    function adminTransfer(
        address _from,
        address _to,
        uint256 _tokenId
    ) public override {
        require(
            stakeManagerStorage.getValidatorInfoValidatorShareContractByIndex(
                _tokenId
            ) != address(0),
            "StakingNFT: only allow on the NFT of a stakable pool"
        );

        if (msg.sender == officialPoolCommittee) {
            require(
                stakeManagerStorage.getValidatorInfoSignerByIndex(_tokenId) ==
                    stakeManagerStorage.officialPool(),
                "StakeManager: only allowed on official pool NFT"
            );
            _adminTransfer(_from, _to, _tokenId);
        } else {
            require(
                stakeManagerStorage.getValidatorInfoSignerByIndex(_tokenId) !=
                    stakeManagerStorage.officialPool(),
                "StakeManager: not allowed on official pool NFT"
            );
            super.adminTransfer(_from, _to, _tokenId);
        }
    }
}

interface IStakeManagerVault {
    function setCommittee(address _committee) external;

    function withdrawKub(address payable _to, uint256 _amount) external;
}

abstract contract StakeManagerHelper {
    IStakeManagerStorage public immutable stakeManagerStorage;
    IStakeManagerVault public immutable stakeManagerVault;
    StakingNFT public immutable nftContract;

    constructor(address stakeManagerStorage_, address stakeManagerVault_) {
        stakeManagerStorage = IStakeManagerStorage(stakeManagerStorage_);
        stakeManagerVault = IStakeManagerVault(stakeManagerVault_);
        nftContract = StakingNFT(stakeManagerStorage.nftContract());
    }

    function _slashManager() internal view returns (address) {
        return stakeManagerStorage.slashManager();
    }

    function _officialPool() internal view returns (address) {
        return stakeManagerStorage.officialPool();
    }

    function _incDescPoolAmount(uint256 val_, ChangeMode mode_) internal {
        stakeManagerStorage.incDescPoolAmount(val_, mode_);
    }

    function _incDescTotalStaked(uint256 val_, ChangeMode mode_) internal {
        stakeManagerStorage.incDescTotalStaked(val_, mode_);
    }

    function _incDescSoloAmount(uint256 val_, ChangeMode mode_) internal {
        stakeManagerStorage.incDescSoloAmount(val_, mode_);
    }

    function _incDescTotalRewards(uint256 val_, ChangeMode mode_) internal {
        stakeManagerStorage.incDescTotalRewards(val_, mode_);
    }

    function _incDescTotalRewardsLiquidated(
        uint256 val_,
        ChangeMode mode_
    ) internal {
        stakeManagerStorage.incDescTotalRewardsLiquidated(val_, mode_);
    }

    function _incDescUnallocatedReward(
        uint256 val_,
        ChangeMode mode_
    ) internal {
        stakeManagerStorage.incDescUnallocatedReward(val_, mode_);
    }

    function _setValidatorInfoAmountByIndex(
        uint256 index_,
        uint256 amount_
    ) internal {
        stakeManagerStorage.setValidatorInfoAmountByIndex(index_, amount_);
    }

    function _incDescValidatorInfoAmountByIndex(
        uint256 index_,
        uint256 amount_,
        ChangeMode mode_
    ) internal {
        stakeManagerStorage.incDescValidatorInfoAmountByIndex(
            index_,
            amount_,
            mode_
        );
    }

    function _setValidatorInfoRewardByIndex(
        uint256 index_,
        uint256 val_
    ) internal {
        stakeManagerStorage.setValidatorInfoRewardByIndex(index_, val_);
    }

    function _incDescValidatorInfoRewardByIndex(
        uint256 index_,
        uint256 amount_,
        ChangeMode mode_
    ) internal {
        stakeManagerStorage.incDescValidatorInfoRewardByIndex(
            index_,
            amount_,
            mode_
        );
    }

    function _setValidatorInfoInfraCommissionRateByIndex(
        uint256 index_,
        uint256 val_
    ) internal {
        stakeManagerStorage.setValidatorInfoInfraCommissionRateByIndex(
            index_,
            val_
        );
    }

    function _incDescValidatorInfoInfraCommissionAmountByIndex(
        uint256 index_,
        uint256 val_,
        ChangeMode mode_
    ) internal {
        stakeManagerStorage.incDescValidatorInfoInfraCommissionAmountByIndex(
            index_,
            val_,
            mode_
        );
    }

    function _setValidatorInfoValidatorCommissionAmountByIndex(
        uint256 index_,
        uint256 amount_
    ) internal {
        return
            stakeManagerStorage
                .setValidatorInfoValidatorCommissionAmountByIndex(
                    index_,
                    amount_
                );
    }

    function _setValidatorInfoDelegatorCommissionAmountByIndex(
        uint256 index_,
        uint256 amount_
    ) internal {
        return
            stakeManagerStorage
                .setValidatorInfoDelegatorCommissionAmountByIndex(
                    index_,
                    amount_
                );
    }

    function _incDescValidatorInfoValidatorCommissionAmountByIndex(
        uint256 index_,
        uint256 amount_,
        ChangeMode mode_
    ) internal {
        return
            stakeManagerStorage
                .incDescValidatorInfoValidatorCommissionAmountByIndex(
                    index_,
                    amount_,
                    mode_
                );
    }

    function _incDescValidatorInfoDelegatorCommissionAmountByIndex(
        uint256 index_,
        uint256 amount_,
        ChangeMode mode_
    ) internal {
        return
            stakeManagerStorage
                .incDescValidatorInfoDelegatorCommissionAmountByIndex(
                    index_,
                    amount_,
                    mode_
                );
    }

    function _setValidatorInfoDelegatorsRewardByIndex(
        uint256 index_,
        uint256 amount_
    ) internal {
        return
            stakeManagerStorage.setValidatorInfoDelegatorsRewardByIndex(
                index_,
                amount_
            );
    }

    function _incDescValidatorInfoDelegatorsRewardByIndex(
        uint256 index_,
        uint256 amount_,
        ChangeMode mode_
    ) internal {
        return
            stakeManagerStorage.incDescValidatorInfoDelegatorsRewardByIndex(
                index_,
                amount_,
                mode_
            );
    }

    function _incDescValidatorInfoDelegatedAmountByIndex(
        uint256 index_,
        uint256 amount_,
        ChangeMode mode_
    ) internal {
        stakeManagerStorage.incDescValidatorInfoDelegatedAmountByIndex(
            index_,
            amount_,
            mode_
        );
    }

    function _getValidatorInfo(
        address key_
    ) internal view returns (Validator memory) {
        return stakeManagerStorage.getValidatorInfo(key_);
    }

    function _getValidatorInfoByIndex(
        uint256 index_
    ) internal view returns (Validator memory) {
        return stakeManagerStorage.getValidatorInfoByIndex(index_);
    }

    function _getValidatorInfoValidatorShareContractByIndex(
        uint256 index_
    ) internal view returns (address) {
        return
            stakeManagerStorage.getValidatorInfoValidatorShareContractByIndex(
                index_
            );
    }

    function _getValidatorInfoStatus(
        address key_
    ) internal view returns (Status) {
        return stakeManagerStorage.getValidatorInfoStatus(key_);
    }

    function _getValidatorInfoInfraCommissionAmountByIndex(
        uint256 index_
    ) internal view returns (uint256) {
        return
            stakeManagerStorage.getValidatorInfoInfraCommissionAmountByIndex(
                index_
            );
    }

    function _getValidatorCurrentIndex(
        address val_
    ) internal view returns (uint256, bool) {
        // revertIfNotFound_ = false
        return stakeManagerStorage.getValidatorCurrentIndex(val_, false);
    }

    function _getValidatorByIndex(
        uint256 index_
    ) internal view returns (address) {
        return stakeManagerStorage.getValidatorByIndex(index_);
    }

    function _getMinimalValidatorListIndex(
        address val_
    ) internal view returns (uint256, bool) {
        // revertIfNotFound_ = false
        return stakeManagerStorage.getMinimalValidatorIndex(val_, false);
    }

    function _removeMinimalValidatorList(address val_) internal {
        require(
            stakeManagerStorage.getMinimalValidatorListLength() > 1,
            "StakeManager: last minimal validator can't be remove"
        );

        stakeManagerStorage.removeMinimalValidator(val_);
    }

    function _depositFund(uint256 amount_) internal {
        (bool success, ) = address(stakeManagerVault).call{ value: amount_ }(
            ""
        );
        require(success, "StakeManager: withdraw failed");
    }

    function _withdrawFund(address to_, uint256 amount_) internal {
        stakeManagerVault.withdrawKub(payable(to_), amount_);
    }

    function _ownerOf(uint256 nftId_) internal view returns (address) {
        return nftContract.ownerOf(nftId_);
    }

    function _updateDelegation(
        address contractAddr_,
        bool delegation_
    ) internal {
        IValidatorShare(contractAddr_).updateDelegation(delegation_);
    }
    function _checkLimit(
        uint256 amount_,
        uint256 limit_,
        string memory revertMsg_
    ) internal pure {
        require(amount_ < limit_, revertMsg_);
    }
}

// * Copyright 2023 Bitkub Blockchain Technology Co., Ltd. - All Rights Reserved.
// * This code is proprietary.
// * Unauthorized copying, modification or distribution of this code, via any
medium, is strictly prohibited without express permission.

enum AddressType {
    Adder,
    Allowed
}

interface ITransferRouter {
    function isAdderOrAllowedAddr(
        address addr_,
        AddressType type_
    ) external view returns (bool);

    function adderOrAllowedAddrLength(
        AddressType type_
    ) external view returns (uint256);

    function adderOrAllowedAddrByIndex(
        uint256 index_,
        AddressType type_
    ) external view returns (address);

    function adderOrAllowedAddrByPage(
        uint256 page_,
        uint256 limit_,
        AddressType type_
    ) external view returns (address[] memory);

    function addAdderAddress(address addr_) external;

    function revokeAdderAddress(address addr_) external;
    function addAllowedAddress(address addr_) external;

    function revokeAllowedAddress(address addr_) external;

    function transferFromKKUB(
        address from_,
        address to_,
        uint256 amount_
    ) external;
}

// * Copyright 2023 Bitkub Blockchain Technology Co., Ltd. - All Rights Reserved.
// * This code is proprietary.
// * Unauthorized copying, modification or distribution of this code, via any
medium, is strictly prohibited without express permission.

abstract contract Initializable {
    bool private _inited = false;

    modifier initializer() {
        require(!_inited, "already inited");
        _inited = true;

        _;
    }
}

abstract contract ReentrancyGuard {
    uint256 private constant _NOT_ENTERED = 1;
    uint256 private constant _ENTERED = 2;

    uint256 private _status;

    function _nonReentrantBefore() private {
        // On the first call to nonReentrant, _notEntered will be true
        require(_status != _ENTERED, "ReentrancyGuard: reentrant call");

        // Any calls to nonReentrant after this point will fail
        _status = _ENTERED;
    }

    function _nonReentrantAfter() private {
        // By storing the original value once again, a refund is triggered (see
        // https://eips.ethereum.org/EIPS/eip-2200)
        _status = _NOT_ENTERED;
    }

    modifier nonReentrant() {
        _nonReentrantBefore();
        _;
        _nonReentrantAfter();
    }

    constructor() {
        _status = _NOT_ENTERED;
    }
}

interface IKAP20 {
    function totalSupply() external view returns (uint256);

    function decimals() external view returns (uint8);

    function symbol() external view returns (string memory);

    function name() external view returns (string memory);

    function balanceOf(address account) external view returns (uint256);

    function transfer(
        address recipient,
        uint256 amount
    ) external returns (bool);

    function allowance(
        address _owner,
        address spender
    ) external view returns (uint256);

    function approve(address spender, uint256 amount) external returns (bool);

    function transferFrom(
        address sender,
        address recipient,
        uint256 amount
    ) external returns (bool);

    function adminTransfer(
        address sender,
        address recipient,
        uint256 amount
    ) external returns (bool success);

    event Transfer(address indexed from, address indexed to, uint256 value);

    event AdminTransfer(
        address indexed from,
        address indexed to,
        uint256 value
    );

    event Approval(
        address indexed owner,
        address indexed spender,
        uint256 value
    );
}

interface IKKUB is IKAP20 {
    function deposit() external payable;

    function withdraw(uint256 _value) external;

    function approve(address spender, uint256 amount) external returns (bool);
}

abstract contract KYCHandler {
    event OnlyKYCAddressActivated(address indexed caller);
    event KYCSet(
        address indexed oldKYC,
        address indexed newKYC,
        address indexed caller
    );
    event AcceptedKYCLevelSet(
        uint256 indexed oldAcceptedKYCLevelSet,
        uint256 indexed newAcceptedKYCLevelSet,
        address indexed caller
    );

    IKYCBitkubChain public kyc;

    uint256 public acceptedKycLevel;
    bool public isActivatedOnlyKycAddress;

    modifier onlyBitkubNext(address bitkubNext_) {
        _onlyBitkubNext(bitkubNext_);
        _;
    }

    constructor(address kyc_, uint256 acceptedKycLevel_) {
        _setKyc(IKYCBitkubChain(kyc_));
        _setAcceptedKycLevel(acceptedKycLevel_);
    }

    function _activateOnlyKycAddress() internal virtual {
        isActivatedOnlyKycAddress = true;
        emit OnlyKYCAddressActivated(msg.sender);
    }

    function _setKyc(IKYCBitkubChain _kyc) internal virtual {
        emit KYCSet(address(kyc), address(_kyc), msg.sender);
        kyc = _kyc;
    }

    function _setAcceptedKycLevel(uint256 _kycLevel) internal virtual {
        emit AcceptedKYCLevelSet(acceptedKycLevel, _kycLevel, msg.sender);
        acceptedKycLevel = _kycLevel;
    }

    function _onlyBitkubNext(address bitkubNext_) internal view {
        require(
            _isBitkubNext(bitkubNext_),
            "KYCHandler: only Bitkub NEXT user"
        );
    }

    function _isBitkubNext(address bitkubNext_) internal view returns (bool) {
        return kyc.kycsLevel(bitkubNext_) >= acceptedKycLevel;
    }
}

abstract contract AccessController is Committee, KYCHandler {
    address public transferRouter;

    event TransferRouterSet(
        address indexed oldTransferRouter,
        address indexed newTransferRouter,
        address indexed caller
    );
    modifier onlyTransferRouter() {
        require(
            msg.sender == transferRouter,
            "AccessController: restricted only transfer router"
        );
        _;
    }

    constructor(
        address committee_,
        address kyc_,
        uint256 acceptedKycLevel_
    ) Committee(committee_) KYCHandler(kyc_, acceptedKycLevel_) {}

    function activateOnlyKycAddress() external onlyCommittee {
        _activateOnlyKycAddress();
    }

    function setKyc(IKYCBitkubChain _kyc) external onlyCommittee {
        _setKyc(_kyc);
    }

    function setAcceptedKycLevel(uint256 _kycLevel) external onlyCommittee {
        _setAcceptedKycLevel(_kycLevel);
    }

    function setTransferRouter(address _transferRouter) external onlyCommittee {
        emit TransferRouterSet(transferRouter, _transferRouter, msg.sender);
        transferRouter = _transferRouter;
    }
}

interface IKToken {
    function internalTransfer(
        address sender,
        address recipient,
        uint256 amount
    ) external returns (bool);

    function externalTransfer(
        address sender,
        address recipient,
        uint256 amount
    ) external returns (bool);
}
```

```
abstract contract KAP20 is IKAP20, IKToken, Pausable, AccessController {
    mapping(address => uint256) _balances;

    mapping(address => mapping(address => uint256)) internal _allowances;

    uint256 private _totalSupply;

    string public override name;
    string public override symbol;
    uint8 public override decimals;

    constructor(
        string memory _name,
        string memory _symbol,
        uint8 _decimals,
        address _kyc,
        address _committee,
        address _transferRouter,
        uint256 _acceptedKycLevel
    ) AccessController(_committee, _kyc, _acceptedKycLevel) {
        name = _name;
        symbol = _symbol;
        decimals = _decimals;
        transferRouter = _transferRouter;
    }

    function totalSupply() public view virtual override returns (uint256) {
        return _totalSupply;
    }

    function balanceOf(
        address account
    ) public view virtual override returns (uint256) {
        return _balances[account];
    }

    function transfer(
        address recipient,
        uint256 amount
    ) public virtual override whenNotPaused returns (bool) {
        _transfer(msg.sender, recipient, amount);
        return true;
    }
    function allowance(
        address owner,
        address spender
    ) public view virtual override returns (uint256) {
        return _allowances[owner][spender];
    }

    function approve(
        address spender,
        uint256 amount
    ) public virtual override returns (bool) {
        _approve(msg.sender, spender, amount);
        return true;
    }

    function transferFrom(
        address sender,
        address recipient,
        uint256 amount
    ) public virtual override whenNotPaused returns (bool) {
        _transfer(sender, recipient, amount);

        uint256 currentAllowance = _allowances[sender][msg.sender];
        require(
            currentAllowance >= amount,
            "KAP20: Transfer amount exceeds allowance"
        );
        unchecked {
            _approve(sender, msg.sender, currentAllowance - amount);
        }

        return true;
    }

    function increaseAllowance(
        address spender,
        uint256 addedValue
    ) public virtual returns (bool) {
        _approve(
            msg.sender,
            spender,
            _allowances[msg.sender][spender] + addedValue
        );
        return true;
    }
    function decreaseAllowance(
        address spender,
        uint256 subtractedValue
    ) public virtual returns (bool) {
        uint256 currentAllowance = _allowances[msg.sender][spender];
        require(
            currentAllowance >= subtractedValue,
            "KAP20: Decreased allowance below zero"
        );
        unchecked {
            _approve(msg.sender, spender, currentAllowance - subtractedValue);
        }

        return true;
    }

    function _transfer(
        address sender,
        address recipient,
        uint256 amount
    ) internal virtual {
        require(sender != address(0), "KAP20: Transfer from the zero address");
        require(recipient != address(0), "KAP20: Transfer to the zero address");

        uint256 senderBalance = _balances[sender];
        require(
            senderBalance >= amount,
            "KAP20: Transfer amount exceeds balance"
        );
        unchecked {
            _balances[sender] = senderBalance - amount;
        }
        _balances[recipient] += amount;

        emit Transfer(sender, recipient, amount);
    }

    function _adminTransfer(
        address sender,
        address recipient,
        uint256 amount
    ) internal virtual {
        uint256 senderBalance = _balances[sender];
        require(
            senderBalance >= amount,
            "KAP20: Transfer amount exceeds balance"
        );
        unchecked {
            _balances[sender] = senderBalance - amount;
        }
        _balances[recipient] += amount;

        emit Transfer(sender, recipient, amount);
        emit AdminTransfer(sender, recipient, amount);
    }

    function _mint(address account, uint256 amount) internal virtual {
        require(account != address(0), "KAP20: Mint to the zero address");

        _totalSupply += amount;
        _balances[account] += amount;
        emit Transfer(address(0), account, amount);
    }

    function _burn(address account, uint256 amount) internal virtual {
        require(account != address(0), "KAP20: Burn from the zero address");

        uint256 accountBalance = _balances[account];
        require(accountBalance >= amount, "KAP20: Burn amount exceeds balance");
        unchecked {
            _balances[account] = accountBalance - amount;
        }
        _totalSupply -= amount;

        emit Transfer(account, address(0), amount);
    }

    function _approve(
        address owner,
        address spender,
        uint256 amount
    ) internal virtual {
        require(owner != address(0), "KAP20: Approve from the zero address");
        require(spender != address(0), "KAP20: Approve to the zero address");

        _allowances[owner][spender] = amount;
        emit Approval(owner, spender, amount);
    }
    function adminTransfer(
        address sender,
        address recipient,
        uint256 amount
    ) public virtual override onlyCommittee returns (bool) {
        _adminTransfer(sender, recipient, amount);
        return true;
    }

    function internalTransfer(
        address sender,
        address recipient,
        uint256 amount
    ) external override whenNotPaused onlyTransferRouter returns (bool) {
        require(
            kyc.kycsLevel(sender) >= acceptedKycLevel &&
                kyc.kycsLevel(recipient) >= acceptedKycLevel,
            "KAP20: Only internal purpose"
        );

        _transfer(sender, recipient, amount);
        return true;
    }

    function externalTransfer(
        address sender,
        address recipient,
        uint256 amount
    ) external override whenNotPaused onlyTransferRouter returns (bool) {
        require(
            kyc.kycsLevel(sender) >= acceptedKycLevel,
            "KAP20: Only internal purpose"
        );

        _transfer(sender, recipient, amount);
        return true;
    }
}

contract ValidatorShare is
    IValidatorShare,
    Initializable,
    ReentrancyGuard,
    KAP20
{
    uint256 public constant REWARD_PRECISION = 10 ** 25;

    IStakeManagerStorage public immutable stakeManagerStorage;
    uint256 public validatorId;

    uint256 public rewardPerShare;
    uint256 public activeAmount;

    bool public delegation;
    uint256 public minDelegated;

    mapping(address => uint256) public initialRewardPerShare;

    IKKUB public immutable kkub;

    address public callHelper;
    ITransferRouter public posTransferRouter;

    bool public isOfficialPool;

    event Delegated(
        uint256 indexed validatorId,
        address indexed user,
        uint256 amount
    );
    event Undelegated(
        uint256 indexed validatorId,
        address indexed user,
        uint256 amount
    );
    event ClaimedRewards(
        uint256 indexed validatorId,
        address indexed user,
        uint256 amount
    );

    event DelegationUpdated(bool indexed delegation, address indexed caller);
    event MinDelegatedUpdated(
        uint256 indexed oldMinDelegated,
        uint256 indexed newMinDelegated,
        address indexed caller
    );

    event CallHelperUpdated(
        address indexed oldCallHelper,
        address indexed newCallHelper,
        address indexed caller
    );

    event PosTransferRouterUpdated(
        address indexed oldPosTransferRouter,
        address indexed newPosTransferRouter,
        address indexed caller
    );

    modifier onlyWhenDelegated() {
        require(delegation, "ValidatorShare: delegation is disabled");
        _;
    }

    modifier onlyCallHelper() {
        require(
            msg.sender == callHelper,
            "ValidatorShare: restricted only call helper"
        );
        _;
    }

    function _onlyStakeManager() private view {
        require(
            msg.sender == stakeManagerStorage.stakeManager(),
            "ValidatorShare: caller is not stake manager"
        );
    }

    modifier onlyStakeManager() {
        _onlyStakeManager();
        _;
    }

    receive() external payable {
        require(
            msg.sender == address(kkub),
            "ValidatorShare: KUB received from unknown origin"
        );
    }

    constructor(
        address kyc_,
        address committee_,
        address kkub_,
        address posTransferRouter_,
        address adminKap20Router_,
        address callHelper_,
        address stakeManagerStorage_,
        bool isOfficialPool_
    )
        KAP20(
            "BKC POS Delegator",
            "BKC-D",
            18,
            kyc_,
            committee_,
            adminKap20Router_,
            4
        )
    {
        callHelper = callHelper_;
        posTransferRouter = ITransferRouter(posTransferRouter_);

        kkub = IKKUB(kkub_);
        stakeManagerStorage = IStakeManagerStorage(stakeManagerStorage_);

        isOfficialPool = isOfficialPool_;
    }

    function setCallHelper(address callHelper_) external onlyCommittee {
        emit CallHelperUpdated(address(callHelper), callHelper_, msg.sender);
        callHelper = callHelper_;
    }

    function setPosTransferRouter(
        address posTransferRouter_
    ) external onlyCommittee {
        emit PosTransferRouterUpdated(
            address(posTransferRouter),
            posTransferRouter_,
            msg.sender
        );
        posTransferRouter = ITransferRouter(posTransferRouter_);
    }

    /*********************** External Functions *************************/
    /// @notice initialize this contract
    /// @param validatorId_ validator id
    /// @param minDelegated_ minimum amount of native token to delegate
    /// @dev this function is called only once during the contract deployment
    /// onlyOwner will prevent this contract from initializing, since it's owner is
going to be 0x0 address
    function initialize(
        uint256 validatorId_,
        uint256 minDelegated_
    ) external initializer {
        validatorId = validatorId_;
        minDelegated = minDelegated_;
        delegation = true;
    }

    /// @notice delegate to a validator
    /// @return amountToDeposit amount of native token deposited
    function delegate() external payable nonReentrant returns (uint256) {
        return _delegateCore(msg.sender, msg.value);
    }

    function delegate(
        uint256 amount_,
        address bitkubNext_
    ) external nonReentrant onlyCallHelper returns (uint256) {
        posTransferRouter.transferFromKKUB(bitkubNext_, address(this), amount_);
        kkub.withdraw(amount_);
        return _delegateCore(bitkubNext_, amount_);
    }

    function _delegateCore(
        address sender_,
        uint256 amount_
    ) private returns (uint256) {
        _officialPoolValidate(sender_);

        _withdrawAndTransferReward(sender_);

        require(
            balanceOf(sender_) + amount_ >= minDelegated,
            "ValidatorShare: amount is less than minimum delegated amount"
        );

        _buyShares(amount_, sender_);

        emit Delegated(validatorId, sender_, amount_);
        return amount_;
    }

    /// @notice undelegate from a validator
    /// @param claimAmount_ amount of native token to claim
    /// @return shares amount of shares burned
    function undelegate(
        uint256 claimAmount_
    ) external nonReentrant returns (uint256) {
        return _undelegateCore(claimAmount_, msg.sender);
    }

    function undelegate(
        uint256 claimAmount_,
        address bitkubNext_
    ) external nonReentrant onlyCallHelper returns (uint256) {
        return _undelegateCore(claimAmount_, bitkubNext_);
    }

    function _undelegateCore(
        uint256 claimAmount_,
        address sender_
    ) private returns (uint256) {
        _officialPoolValidate(sender_);

        claimAmount_ = _sellShares(claimAmount_, sender_);

        emit Undelegated(validatorId, sender_, claimAmount_);
        return claimAmount_;
    }

    function claimRewards() external nonReentrant returns (uint256) {
        return _claimRewardsCore(msg.sender);
    }

    function claimRewards(
        address bitkubNext_
    ) external nonReentrant onlyCallHelper returns (uint256) {
        return _claimRewardsCore(bitkubNext_);
    }

    function _claimRewardsCore(address sender_) private returns (uint256) {
        _officialPoolValidate(sender_);

        return _withdrawAndTransferReward(sender_);
    }
    function updateDelegation(bool delegation_) external onlyStakeManager {
        emit DelegationUpdated(delegation_, msg.sender);
        delegation = delegation_;
    }

    function updateMinDelegated(
        uint256 minDelegated_
    ) external onlyStakeManager {
        emit MinDelegatedUpdated(minDelegated, minDelegated_, msg.sender);
        minDelegated = minDelegated_;
    }

    /*********************** Public Functions *************************/

    function getLiquidRewards(address user_) public view returns (uint256) {
        return _calculateReward(user_, getRewardPerShare());
    }

    function getRewardPerShare() public view returns (uint256) {
        return
            _calculateRewardPerShareWithRewards(
                stakeManagerStorage.getValidatorInfoDelegatorsRewardByIndex(
                    validatorId
                )
            );
    }

    /*********************** Private Functions *************************/

    function _stakeManager() internal view returns (IStakeManager) {
        return IStakeManager(stakeManagerStorage.stakeManager());
    }

    function _transfer(
        address from_,
        address to_,
        uint256 value_
    ) internal override {
        // get rewards for recipient
        _withdrawAndTransferReward(to_);
        // convert rewards to shares
        _withdrawAndTransferReward(from_);
        // move shares to recipient
        super._transfer(from_, to_, value_);
    }

    function _adminTransfer(
        address from_,
        address to_,
        uint256 value_
    ) internal override {
        _withdrawAndTransferReward(from_);
        _withdrawAndTransferReward(to_);
        super._adminTransfer(from_, to_, value_);
    }

    function _sellShares(
        uint256 claimAmount_,
        address sender_
    ) private returns (uint256) {
        // first get how much staked in total and compare to target unstake amount
        uint256 totalStaked = balanceOf(sender_);
        require(
            totalStaked != 0 && totalStaked >= claimAmount_,
            "ValidatorShare: too much requested"
        );

        if (totalStaked - claimAmount_ < minDelegated) {
            claimAmount_ = totalStaked;
        }

        _withdrawAndTransferReward(sender_);

        _burn(sender_, claimAmount_);
        _stakeManager().updateValidatorState(
            validatorId,
            -int256(claimAmount_)
        );
        activeAmount = activeAmount - claimAmount_;

        require(
            _stakeManager().transferFunds(
                validatorId,
                claimAmount_,
                sender_,
                TransferTokenMode.Staked
            ),
            "ValidatorShare: insufficent rewards"
        );
        return claimAmount_;
    }

    function _withdrawReward(address user_) private returns (uint256) {
        uint256 localRewardPerShare = _calculateRewardPerShareWithRewards(
            _stakeManager().withdrawDelegatorsReward(validatorId)
        );
        uint256 liquidRewards = _calculateReward(user_, localRewardPerShare);

        rewardPerShare = localRewardPerShare;
        initialRewardPerShare[user_] = localRewardPerShare;
        return liquidRewards;
    }

    function _withdrawAndTransferReward(
        address user_
    ) private returns (uint256) {
        uint256 liquidRewards = _withdrawReward(user_);

        if (liquidRewards != 0) {
            require(
                _stakeManager().transferFunds(
                    validatorId,
                    liquidRewards,
                    user_,
                    TransferTokenMode.Reward
                ),
                "ValidatorShare: insufficent rewards"
            );
            emit ClaimedRewards(validatorId, user_, liquidRewards);
        }

        return liquidRewards;
    }

    function _buyShares(
        uint256 amount_,
        address user_
    ) private onlyWhenDelegated {
        _mint(user_, amount_);

        _stakeManager().updateValidatorState(validatorId, int256(amount_));
        activeAmount = activeAmount + amount_;
        bool success = _stakeManager().delegationDeposit{ value: amount_ }(
            validatorId
        );
        require(success, "ValidatorShare: deposit failed");
    }

    function _calculateRewardPerShareWithRewards(
        uint256 accumulatedReward_
    ) private view returns (uint256) {
        uint256 localRewardPerShare = rewardPerShare;
        if (accumulatedReward_ != 0) {
            uint256 totalShares = totalSupply();

            if (totalShares != 0) {
                localRewardPerShare =
                    localRewardPerShare +
                    ((accumulatedReward_ * REWARD_PRECISION) / totalShares);
            }
        }

        return localRewardPerShare;
    }

    function _calculateReward(
        address user_,
        uint256 rewardPerShare_
    ) private view returns (uint256) {
        uint256 shares = balanceOf(user_);
        if (shares == 0) {
            return 0;
        }

        uint256 localInitialRewardPerShare = initialRewardPerShare[user_];

        if (localInitialRewardPerShare == rewardPerShare_) {
            return 0;
        }

        return
            ((rewardPerShare_ - localInitialRewardPerShare) * shares) /
            REWARD_PRECISION;
    }

    function _officialPoolValidate(address sender_) private view {
        if (isOfficialPool) {
            require(
                _isBitkubNext(sender_),
                "ValidatorShare: sender is not BitkubNext"
            );
        }
    }
}

interface IKYC {
    function setKycCompleted(address _addr, uint256 _level) external;
}

contract ValidatorShareFactory {
    address[] public validatorPool;

    address public immutable kkub;
    address public immutable stakeManagerStorage;

    address public kyc;
    address public committee;
    address public posTransferRouter;
    address public adminKap20Router;
    address public callHelper;

    event ValidatorShareCreated(
        address indexed addressShare,
        address indexed owner
    );
    event KycUpdated(
        address indexed oldKyc,
        address indexed newKyc,
        address indexed caller
    );
    event CommitteeUpdated(
        address indexed oldCommittee,
        address indexed newCommittee,
        address indexed caller
    );
    event PosTransferRouterUpdated(
        address indexed oldPosTransferRouter,
        address indexed newPosTransferRouter,
        address indexed caller
    );
    event AdminKap20RouterUpdated(
        address indexed oldAdminKap20Router,
        address indexed newAdminKap20Router,
        address indexed caller
    );
    event CallHelperUpdated(
        address indexed oldCallHelper,
        address indexed newCallHelper,
        address indexed caller
    );

    modifier onlyCommittee() {
        require(
            msg.sender == committee,
            "ValidatorShareFactory: restricted only committee"
        );
        _;
    }

    modifier onlyStakeManager() {
        require(
            msg.sender ==
                IStakeManagerStorage(stakeManagerStorage).stakeManager(),
            "ValidatorShareFactory: restricted only stake manager"
        );
        _;
    }

    constructor(
        address kyc_,
        address committee_,
        address posTransferRouter_,
        address adminKap20Router_,
        address kkub_,
        address callHelper_,
        address stakeManagerStorage_
    ) {
        kyc = kyc_;
        committee = committee_;
        posTransferRouter = posTransferRouter_;
        adminKap20Router = adminKap20Router_;
        kkub = kkub_;
        callHelper = callHelper_;
        stakeManagerStorage = stakeManagerStorage_;
    }

    function create(
        uint256 validatorId_,
        uint256 minDelegated_,
        bool isOfficialPool_
    ) public onlyStakeManager returns (address) {
        ValidatorShare vs = new ValidatorShare(
            kyc,
            committee,
            kkub,
            posTransferRouter,
            adminKap20Router,
            callHelper,
            stakeManagerStorage,
            isOfficialPool_
        );

        vs.initialize(validatorId_, minDelegated_);

        address addr = address(vs);
        validatorPool.push(addr);

        ITransferRouter(posTransferRouter).addAllowedAddress(addr);
        IKYC(kyc).setKycCompleted(addr, 16);

        emit ValidatorShareCreated(addr, msg.sender);

        return addr;
    }

    function validatorPoolLength() external view returns (uint256) {
        return validatorPool.length;
    }

    ////////////////////////////////////////// setter
//////////////////////////////////////////

    function setKyc(address kyc_) external onlyCommittee {
        emit KycUpdated(kyc, kyc_, msg.sender);
        kyc = kyc_;
    }

    function setCommittee(address committee_) external onlyCommittee {
        emit CommitteeUpdated(committee, committee_, msg.sender);
        committee = committee_;
    }
    function setPosTransferRouter(
        address posTransferRouter_
    ) external onlyCommittee {
        emit PosTransferRouterUpdated(
            posTransferRouter,
            posTransferRouter_,
            msg.sender
        );
        posTransferRouter = posTransferRouter_;
    }

    function setAdminKap20Router(
        address adminKap20Router_
    ) external onlyCommittee {
        emit AdminKap20RouterUpdated(
            adminKap20Router,
            adminKap20Router_,
            msg.sender
        );
        adminKap20Router = adminKap20Router_;
    }

    function setCallHelper(address callHelper_) external onlyCommittee {
        emit CallHelperUpdated(callHelper, callHelper_, msg.sender);
        callHelper = callHelper_;
    }
}

abstract contract AccessController2 is Committee {
    address public callHelper;
    ITransferRouter public transferRouter;

    function _onlyCallHelper() internal view {
        require(
            msg.sender == callHelper,
            "AccessController2: restricted only call helper"
        );
    }

    modifier onlyCallHelper() {
        _onlyCallHelper();
        _;
    }

    event CallHelperSet(
        address indexed oldCallHelper,
        address indexed newCallHelper,
        address indexed caller
    );

    event TransferRouterSet(
        address indexed oldTransferRouter,
        address indexed newTransferRouter,
        address indexed caller
    );

    constructor(
        address committee_,
        address callHelper_,
        address transferRouter_
    ) Committee(committee_) {
        callHelper = callHelper_;
        transferRouter = ITransferRouter(transferRouter_);
    }

    function setCallHelper(address _callHelper) external onlyCommittee {
        emit CallHelperSet(address(callHelper), _callHelper, msg.sender);
        callHelper = _callHelper;
    }

    function setTransferRouter(address _transferRouter) external onlyCommittee {
        emit TransferRouterSet(
            address(transferRouter),
            _transferRouter,
            msg.sender
        );
        transferRouter = ITransferRouter(_transferRouter);
    }
}

abstract contract StakeManagerConstants {
    uint16 public constant MAX_RATE = 10000;
    uint16 public constant MAX_COMMISSION_RATE = 5000;
    uint16 public constant MAX_INFRA_COMMISSION_RATE = 5000;
    uint256 public constant INCORRECT_VALIDATOR_ID = type(uint256).max;
}

struct StakeManagerConstructorInput {
    address stakeManagerStorage;
    address stakeManagerVault;
    address kkub;
    address transferRouter;
    address committee;
    address officialPoolStaker;
    address callHelper;
}

contract StakeManager is
    IStakeManager,
    StakeManagerConstants,
    StakeManagerHelper,
    ReentrancyGuard,
    AccessController2
{
    /*********************** events *************************/

    event OfficialPoolStakerSet(
        address indexed oldOfficialPoolStaker,
        address indexed newOfficialPoolStaker,
        address indexed caller
    );
    event TransferUnallocatedReward(
        address indexed to,
        uint256 amount,
        address indexed caller
    );
    event TransferInfraCommission(
        uint256 indexed validatorId,
        address indexed to,
        uint256 amount,
        address indexed caller
    );

    event ValidatorStaked(
        uint256 indexed validatorId,
        address indexed validator,
        uint256 amount,
        address indexed validatorShareContract,
        Status validatorStatus,
        uint256 minDelegated,
        bool isDelegation,
        bool isOfficialPool
    );
    event ValidatorUnstaked(uint256 indexed validatorId, uint256 amount);
    event ValidatorRestaked(
        uint256 indexed validatorId,
        bool indexed isMinimalValidator,
        uint256 amount
    );
    event DistributeRewards(
        uint256 indexed validatorId,
        uint256 totalReward,
        uint256 validatorReward,
        uint256 delegatorsReward,
        uint256 infraCommission,
        uint256 validatorCommission,
        uint256 delegatorCommission
    );
    event DeprecatedDeposit(
        uint256 indexed validatorId,
        uint256 amount,
        Status validatorStatus
    );
    event ClaimRewards(
        uint256 indexed validatorId,
        uint256 amount,
        LiquidateMode liquidateMode
    );
    event DelegationUpdated(
        uint256 indexed validatorId,
        bool delegation,
        uint256 timestamp
    );
    event MinDelegatedUpdated(
        uint256 indexed validatorId,
        uint256 newMinDelegation,
        uint256 timestamp
    );
    event InfraCommissionRateUpdated(
        uint256 indexed validatorId,
        uint256 newInfraCommissionRate,
        uint256 timestamp
    );
    event CommissionRateUpdated(
        uint256 indexed validatorId,
        uint256 newCommissionRate,
        uint256 timestamp
    );
    event SlashSuccess(
        uint256 indexed validatorId,
        bool indexed isMinimalValidator,
        uint256 slashedAmount,
        uint256 amount,
        uint256 delegatedAmount
    );
    event SlashFail(
        uint256 indexed validatorId,
        SlashErrorCode indexed errorCase
    );

    /*********************** modifiers *************************/

    modifier noEmptyDeposit() {
        require(msg.value > 0, "StakeManager: deposit value is zero");
        _;
    }

    modifier onlyCoinbase() {
        require(
            msg.sender == block.coinbase,
            "StakeManager: the message sender must be the block producer"
        );
        _;
    }

    modifier onlyStaker(uint256 validatorId, address user) {
        _onlyStaker(validatorId, user);
        _;
    }

    modifier onlyDelegation(uint256 validatorId) {
        _onlyDelegation(validatorId);
        _;
    }

    modifier onlySlashManager() {
        _onlySlashManager();
        _;
    }

    /*********************** constructor *************************/

    IKKUB public immutable kkub;

    address public officialPoolStaker;
    receive() external payable {
        require(
            msg.sender == address(kkub),
            "StakeManager: KUB received from unknown origin"
        );
    }

    constructor(
        StakeManagerConstructorInput memory input_
    )
        StakeManagerHelper(input_.stakeManagerStorage, input_.stakeManagerVault)
        AccessController2(
            input_.committee,
            input_.callHelper,
            input_.transferRouter
        )
    {
        kkub = IKKUB(input_.kkub);
        officialPoolStaker = input_.officialPoolStaker;
    }

    /*********************** committee *************************/

    function setOfficialPoolStaker(
        address officialPoolStaker_
    ) external override onlyCommittee {
        (, bool valid) = _getValidatorCurrentIndex(_officialPool());
        require(!valid, "StakeManager: official pool is already staked");

        emit OfficialPoolStakerSet(
            officialPoolStaker,
            officialPoolStaker_,
            msg.sender
        );
        officialPoolStaker = officialPoolStaker_;
    }

    function transferUnallocatedReward(
        address payable to_,
        uint256 amount_
    ) external override onlyCommittee returns (bool) {
        require(
            amount_ <= stakeManagerStorage.unallocatedReward(),
            "StakeManager: amount must not be more than unallocated reward"
        );

        _incDescUnallocatedReward(amount_, ChangeMode.Decrease);

        emit TransferUnallocatedReward(to_, amount_, msg.sender);
        return _transferToken(to_, amount_, TransferTokenMode.None);
    }

    function transferInfraCommission(
        uint256[] calldata validatorIds_,
        uint256[] calldata amounts_,
        address payable to_
    ) external override onlyCommittee {
        require(
            validatorIds_.length == amounts_.length,
            "StakeManager: inputs are not of the same length"
        );

        uint256 validatorLength = stakeManagerStorage.getValidatorListLength();
        for (uint256 i = 0; i < validatorIds_.length; i++) {
            uint256 validatorId = validatorIds_[i];
            require(
                validatorId < validatorLength,
                "StakeManager: validator id out of range"
            );

            uint256 amount = amounts_[i];
            if (amount == 0) {
                amount = _getValidatorInfoInfraCommissionAmountByIndex(
                    validatorId
                );
            } else {
                require(
                    amount <=
                        _getValidatorInfoInfraCommissionAmountByIndex(
                            validatorId
                        ),
                    "StakeManager: amount must not be more than infra commission"
                );
            }

            _incDescValidatorInfoInfraCommissionAmountByIndex(
                validatorId,
                amount,
                ChangeMode.Decrease
            );
            emit TransferInfraCommission(validatorId, to_, amount, msg.sender);
            _transferToken(to_, amount, TransferTokenMode.Reward);
        }
    }

    /// @dev caution needed for reverts
    /// @notice Distribute rewards to validators and delegators called by coinbase
    function distributeReward()
        external
        payable
        override
        onlyCoinbase
        noEmptyDeposit
    {
        address msgSender = msg.sender;
        (
            address oldOfficialSigner,
            address newOfficialSigner,
            uint256 validBlock
        ) = stakeManagerStorage.getOldNewOfficialSignerAndValidBlock();
        if (block.number < validBlock) {
            if (msgSender == oldOfficialSigner) {
                msgSender = newOfficialSigner;
            } else if (msgSender == newOfficialSigner) {
                revert("StakeManager: new official signer not valid yet");
            }
        }

        // REVERT will revert if msgSender is not a validator
        uint256 validatorId = getValidatorId(msgSender);

        uint256 reward = msg.value;
        Validator memory validator = _getValidatorInfoByIndex(validatorId);

        if (
            validator.status == Status.Active &&
            validator.delegatedAmount + validator.amount > 0
        ) {
            _updateRewards(validatorId, validator, reward);
        } else {
            _incDescUnallocatedReward(reward, ChangeMode.Increase);
            emit DeprecatedDeposit(validatorId, reward, validator.status);
        }
        _depositFund(reward);
    }

    /*********************** system's external & public write functions
*************************/

    /// @notice Transfer funds from StakeManager to delegator called by
ValidatorShare contract
    function transferFunds(
        uint256 validatorId_,
        uint256 amount_,
        address delegator_,
        TransferTokenMode mode_
    ) external override onlyDelegation(validatorId_) returns (bool) {
        require(
            mode_ != TransferTokenMode.None,
            "StakeManager: transfer mode not allowed"
        );

        return _transferToken(delegator_, amount_, mode_);
    }

    /// @notice Transfer funds from ValidatorShare contract to StakeManager called
by ValidatorShare contract
    function delegationDeposit(
        uint256 validatorId_
    ) external payable override onlyDelegation(validatorId_) returns (bool) {
        _depositFund(msg.value);

        _incDescTotalStaked(msg.value, ChangeMode.Increase);

        return true;
    }

    /// @dev for updating states in ValidatorShare contract, no fund is transfered
out
    function withdrawDelegatorsReward(
        uint256 validatorId_
    ) external override onlyDelegation(validatorId_) returns (uint256) {
        uint256 totalReward = stakeManagerStorage
            .getValidatorInfoDelegatorsRewardByIndex(validatorId_);
        _setValidatorInfoDelegatorsRewardByIndex(validatorId_, 0);
        return totalReward;
    }
    /// @dev for updating states here, no fund is transfered in
    function updateValidatorState(
        uint256 validatorId_,
        int256 amount_
    ) external override onlyDelegation(validatorId_) {
        address validatorAddress = _getValidatorByIndex(validatorId_);

        if (amount_ >= 0) {
            require(
                _isActivated(validatorAddress),
                "StakeManager: not allowed to delegate"
            );
            _incDescValidatorInfoDelegatedAmountByIndex(
                validatorId_,
                uint256(amount_),
                ChangeMode.Increase
            );
        } else {
            _incDescValidatorInfoDelegatedAmountByIndex(
                validatorId_,
                uint256(amount_ * -1),
                ChangeMode.Decrease
            );
        }
    }

    /// @dev caution needed for reverts
    function slash(
        address signer_
    ) external override onlySlashManager returns (bool) {
        address localOfficialPool = _officialPool();
        uint256 localSlashAmount;

        (uint256 validatorId, bool valid) = _getValidatorCurrentIndex(signer_);
        if (!valid) {
            emit SlashFail(uint256(uint160(signer_)), SlashErrorCode.NotPool);
            return false;
        }

        Validator memory validator = _getValidatorInfoByIndex(validatorId);
        if (!_isActivated(validator.status)) {
            emit SlashFail(validatorId, SlashErrorCode.InactivatedPool);
            return false;
        }
        if (
            _isSoloPoolAddress(
                validator.status,
                validator.validatorShareContract
            )
        ) {
            // case: solo pool
            localSlashAmount =
                (validator.amount * stakeManagerStorage.soloSlashRate()) /
                MAX_RATE;
        } else if (signer_ == localOfficialPool) {
            // case: official pool should not be slashed
            emit SlashFail(validatorId, SlashErrorCode.OfficialPool);
            return false;
        } else {
            // case: pool
            localSlashAmount = stakeManagerStorage.poolSlashAmount();
        }

        // if amount is less than minDeposit,
        // it's possible that the pool has been slashed before in the span
        // if amount is less than slashAmount, slash to 0
        if (validator.amount < validator.minDeposit) {
            emit SlashFail(validatorId, SlashErrorCode.StakeAmountNotEnough);
            return false;
        } else if (validator.amount < localSlashAmount) {
            localSlashAmount = validator.amount;
        }

        // slash
        _incDescValidatorInfoAmountByIndex(
            validatorId,
            localSlashAmount,
            ChangeMode.Decrease
        );
        _incDescTotalStaked(localSlashAmount, ChangeMode.Decrease);
        validator.amount -= uint128(localSlashAmount); // slash amount should not be
more than the total supply

        // REVERT
        // if the last validator is not official pool and
        // the amount is less than minDeposit,
        // this will revert
        bool isMinimalValidator = _handleMinimalValidatorState(validator);
        // deposit slashed kub into official pool's reward pool
        // REVERT
        // will revert if official pool value is not a validator
        (
            uint256 officialPoolValId,
            bool officialPoolValid
        ) = _getValidatorCurrentIndex(localOfficialPool);
        require(officialPoolValid, "StakeManager: official pool doesn't exist");
        _updateRewards(
            officialPoolValId,
            _getValidatorInfo(localOfficialPool),
            localSlashAmount
        );

        emit SlashSuccess(
            validatorId,
            isMinimalValidator,
            localSlashAmount,
            validator.amount,
            validator.delegatedAmount
        );
        return true;
    }

    /*********************** users' external & public write functions
**************************/

    function stake(
        address signer_,
        bool delegation_
    ) external payable override nonReentrant {
        _stakeCore(signer_, delegation_, msg.sender, uint128(msg.value)); // will
not exceed the total supply of KUB
    }

    function stake(
        address signer_,
        bool delegation_,
        uint256 amount_,
        address bitkubNext_
    ) external override nonReentrant onlyCallHelper {
        _transferInKKUB(bitkubNext_, amount_);
        _stakeCore(signer_, delegation_, bitkubNext_, uint128(amount_)); // will not
exceed the total supply of KUB
    }
    function unstake(
        uint256 validatorId_
    ) external override nonReentrant onlyStaker(validatorId_, msg.sender) {
        _unstakeCore(validatorId_, msg.sender);
    }

    function unstake(
        uint256 validatorId_,
        address bitkubNext_
    )
        external
        override
        nonReentrant
        onlyCallHelper
        onlyStaker(validatorId_, bitkubNext_)
    {
        _unstakeCore(validatorId_, bitkubNext_);
    }

    function restake(
        uint256 validatorId_
    )
        external
        payable
        override
        nonReentrant
        onlyStaker(validatorId_, msg.sender)
    {
        _restakeCore(validatorId_, uint128(msg.value)); // will not exceed the total
supply of KUB
    }

    function restake(
        uint256 validatorId_,
        uint256 amount_,
        address bitkubNext_
    )
        external
        override
        nonReentrant
        onlyCallHelper
        onlyStaker(validatorId_, bitkubNext_)
    {
        _transferInKKUB(bitkubNext_, amount_);
        _restakeCore(validatorId_, uint128(amount_)); // will not exceed the total
supply of KUB
    }

    function claimRewards(
        uint256 validatorId_
    ) external override nonReentrant onlyStaker(validatorId_, msg.sender) {
        _claimRewardsCore(validatorId_, msg.sender);
    }

    function claimRewards(
        uint256 validatorId_,
        address bitkubNext_
    )
        external
        override
        nonReentrant
        onlyCallHelper
        onlyStaker(validatorId_, bitkubNext_)
    {
        _claimRewardsCore(validatorId_, bitkubNext_);
    }

    function claimCommissionRewards(
        uint256 validatorId_
    ) external override nonReentrant onlyStaker(validatorId_, msg.sender) {
        _claimCommissionRewardsCore(validatorId_, msg.sender);
    }

    function claimCommissionRewards(
        uint256 validatorId_,
        address bitkubNext_
    )
        external
        override
        nonReentrant
        onlyCallHelper
        onlyStaker(validatorId_, bitkubNext_)
    {
        _claimCommissionRewardsCore(validatorId_, bitkubNext_);
    }

    function updateInfraCommissionRate(
        uint256 validatorId_,
        uint256 newInfraCommissionRate_
    ) external override nonReentrant onlyCommittee {
        _updateInfraCommissionRateCore(validatorId_, newInfraCommissionRate_);
    }

    function updateCommissionRate(
        uint256 validatorId_,
        uint256 newCommissionRate_
    ) external override nonReentrant onlyStaker(validatorId_, msg.sender) {
        _updateCommissionRateCore(validatorId_, newCommissionRate_);
    }

    function updateCommissionRate(
        uint256 validatorId_,
        uint256 newCommissionRate_,
        address bitkubNext_
    )
        external
        override
        nonReentrant
        onlyCallHelper
        onlyStaker(validatorId_, bitkubNext_)
    {
        _updateCommissionRateCore(validatorId_, newCommissionRate_);
    }

    function updateMinDelegated(
        uint256 validatorId_,
        uint256 newMinDelegated_
    ) external override nonReentrant onlyStaker(validatorId_, msg.sender) {
        _updateMinDelegatedCore(validatorId_, newMinDelegated_);
    }

    function updateMinDelegated(
        uint256 validatorId_,
        uint256 newMinDelegated_,
        address bitkubNext_
    )
        external
        override
        nonReentrant
        onlyCallHelper
        onlyStaker(validatorId_, bitkubNext_)
    {
        _updateMinDelegatedCore(validatorId_, newMinDelegated_);
    }
    function updateValidatorDelegation(
        uint256 validatorId_,
        bool delegation_
    ) external override nonReentrant onlyStaker(validatorId_, msg.sender) {
        _updateValidatorDelegationCore(validatorId_, delegation_);
    }

    function updateValidatorDelegation(
        uint256 validatorId_,
        bool delegation_,
        address bitkubNext_
    )
        external
        override
        nonReentrant
        onlyCallHelper
        onlyStaker(validatorId_, bitkubNext_)
    {
        _updateValidatorDelegationCore(validatorId_, delegation_);
    }

    /*********************** system's external & public read functions
*************************/

    function getMinimalValidators()
        external
        view
        override
        returns (MinimalValidator[] memory)
    {
        (
            address[] memory minimalValidators,
            uint256[] memory validatorPowers
        ) = stakeManagerStorage.getMinimalValidatorsWithValidatorPowerByPage(
                0,
                0
            );

        return _getMinimalValidators(minimalValidators, validatorPowers);
    }

    function getMinimalValidatorsByPage(
        uint256 page_,
        uint256 limit_
    ) external view override returns (MinimalValidator[] memory) {
        (
            address[] memory minimalValidators,
            uint256[] memory validatorPowers
        ) = stakeManagerStorage.getMinimalValidatorsWithValidatorPowerByPage(
                page_,
                limit_
            );

        return _getMinimalValidators(minimalValidators, validatorPowers);
    }

    function getMinimalValidatorsLength()
        external
        view
        override
        returns (uint256)
    {
        return stakeManagerStorage.getMinimalValidatorListLength();
    }

    /********************** users' external & public read functions
*************************/

    function getValidatorId(
        address signer_
    ) public view override returns (uint256) {
        (uint256 val, bool valid) = _getValidatorCurrentIndex(signer_);
        require(valid, "StakeManager: not a validator");
        return val;
    }

    function getMinimalValidatorIndex(
        address signer_
    ) external view override returns (uint256) {
        (uint256 val, bool valid) = _getMinimalValidatorListIndex(signer_);
        require(valid, "StakeManager: not a validator");
        return val;
    }

    function getValidators() external view override returns (address[] memory) {
        return stakeManagerStorage.getAllValidator();
    }

    /********************** internal write functions *************************/
    function _stakeCore(
        address signer_,
        bool delegation_,
        address sender_,
        uint128 value_
    ) private {
        address localOfficialPool = _officialPool();

        uint128 minDeposit = 1;
        uint256 minDelegated = 0;
        bool isOfficialPool = false;
        if (signer_ == localOfficialPool) {
            // Official
            require(
                sender_ == officialPoolStaker,
                "StakeManager: only official pool staker is allowed to stake
official pool"
            );
            require(
                delegation_,
                "StakeManager: official pool must have delegation enabled"
            );
            _checkLimit(
                stakeManagerStorage.officialAmount(),
                stakeManagerStorage.officialLimit(),
                "StakeManager: official pool limit reached"
            );
            stakeManagerStorage.incDescOfficialAmount(1, ChangeMode.Increase);
            minDelegated = 1 ether;
            isOfficialPool = true;
        } else if (delegation_) {
            // Pool
            _checkLimit(
                stakeManagerStorage.poolAmount(),
                stakeManagerStorage.poolLimit(),
                "StakeManager: pool limit reached"
            );
            _incDescPoolAmount(1, ChangeMode.Increase);

            minDeposit = 100_000 ether;
            minDelegated = 100 ether;
        } else {
            // Solo
            _checkLimit(
                stakeManagerStorage.soloAmount(),
                stakeManagerStorage.soloLimit(),
                "StakeManager: solo limit reached"
            );
            _incDescSoloAmount(1, ChangeMode.Increase);

            minDeposit = 10 ether;
        }

        require(value_ >= minDeposit, "StakeManager: not enough to deposit");
        (uint256 validatorId, Validator memory validator) = _stake(
            signer_,
            value_,
            delegation_,
            minDeposit,
            minDelegated,
            sender_,
            isOfficialPool
        );
        _depositFund(value_);
        _emitValidatorStaked(
            validatorId,
            validator,
            minDelegated,
            localOfficialPool
        );
    }

    function _stake(
        address signer_,
        uint128 amount_,
        bool delegation_,
        uint128 minDeposit_,
        uint256 minDelegated_,
        address sender_,
        bool isOfficialPool_
    ) private returns (uint256, Validator memory) {
        ValidatorShareFactory validatorShareFactory = ValidatorShareFactory(
            stakeManagerStorage.validatorShareFactory()
        );

        _incDescTotalStaked(amount_, ChangeMode.Increase);

        Validator memory validator = Validator({
            minDeposit: minDeposit_,
            reward: 0,
            amount: amount_,
            signer: signer_,
            validatorShareContract: address(0x0),
            status: Status.Active,
            infraCommissionRate: delegation_
                ? uint16(
                    stakeManagerStorage.defaultInfraCommissionRate() // will not
exceed 10000
                )
                : 0,
            infraCommissionAmount: 0,
            commissionRate: 0,
            validatorCommissionAmount: 0,
            delegatorCommissionAmount: 0,
            delegatorsReward: 0,
            delegatedAmount: 0
        });

        uint256 validatorId = stakeManagerStorage.addValidator(
            signer_,
            validator
        );
        if (delegation_) {
            address validatorShareContract = validatorShareFactory.create(
                validatorId,
                minDelegated_,
                isOfficialPool_
            );
            stakeManagerStorage.setValidatorInfoValidatorShareContractByIndex(
                validatorId,
                validatorShareContract
            );
            validator.validatorShareContract = validatorShareContract;
        }

        nftContract.mint(sender_, validatorId);
        _handleMinimalValidatorState(validator);

        return (validatorId, validator);
    }

    function _unstakeCore(uint256 validatorId_, address owner_) private {
        Validator memory validator = _getValidatorInfoByIndex(validatorId_);
        emit ValidatorUnstaked(validatorId_, validator.amount);
        _unstake(validatorId_, validator, payable(owner_));
    }

    function _unstake(
        uint256 validatorId_,
        Validator memory validator_,
        address payable staker_
    ) private {
        require(
            _officialPool() != validator_.signer,
            "StakeManager: official not allowed to be unstaked"
        );

        _liquidateRewards(validatorId_, validator_, staker_, LiquidateMode.All);

        address delegationContract = validator_.validatorShareContract;
        if (delegationContract != address(0)) {
            _updateDelegation(delegationContract, false);
            _incDescPoolAmount(1, ChangeMode.Decrease);
        } else {
            _incDescSoloAmount(1, ChangeMode.Decrease);
        }

        nftContract.burn(validatorId_);

        _transferToken(staker_, validator_.amount, TransferTokenMode.Staked);
        _setValidatorInfoAmountByIndex(validatorId_, 0);
        validator_.amount = 0;

        _handleMinimalValidatorState(validator_);

        stakeManagerStorage.setValidatorInfoStatusByIndex(
            validatorId_,
            Status.Unstaked
        );
        validator_.status = Status.Unstaked;
    }

    function _restakeCore(uint256 validatorId_, uint128 amount_) private {
        Validator memory validator = _getValidatorInfoByIndex(validatorId_);
        _onlyActivated(validator);

        _incDescTotalStaked(amount_, ChangeMode.Increase);
        _incDescValidatorInfoAmountByIndex(
            validatorId_,
            amount_,
            ChangeMode.Increase
        );
        validator.amount += amount_;
        _depositFund(amount_);

        bool isInMinimalValidatorSet = _handleMinimalValidatorState(validator);

        emit ValidatorRestaked(validatorId_, isInMinimalValidatorSet, amount_);
    }

    function _claimRewardsCore(uint256 validatorId_, address sender_) private {
        _liquidateRewards(
            validatorId_,
            _getValidatorInfoByIndex(validatorId_),
            payable(sender_),
            LiquidateMode.ValidatorReward
        );
    }

    function _claimCommissionRewardsCore(
        uint256 validatorId_,
        address sender_
    ) private {
        _liquidateRewards(
            validatorId_,
            _getValidatorInfoByIndex(validatorId_),
            payable(sender_),
            LiquidateMode.ValidatorCommission
        );
    }

    function _updateInfraCommissionRateCore(
        uint256 validatorId_,
        uint256 newInfraCommissionRate_
    ) private {
        require(
            newInfraCommissionRate_ <= MAX_INFRA_COMMISSION_RATE,
            "StakeManager: max commission rate exceeded"
        );

        Validator memory validator = _getValidatorInfoByIndex(validatorId_);
        _onlyActivated(validator);
        _onlyValidatorShareContractExist(validator);
        _setValidatorInfoInfraCommissionRateByIndex(
            validatorId_,
            newInfraCommissionRate_
        );
        validator.infraCommissionRate = uint16(newInfraCommissionRate_); // will not
exceed 10000

        emit InfraCommissionRateUpdated(
            validatorId_,
            newInfraCommissionRate_,
            block.timestamp
        );
    }

    function _updateCommissionRateCore(
        uint256 validatorId_,
        uint256 newCommissionRate_
    ) private {
        require(
            newCommissionRate_ <= MAX_COMMISSION_RATE,
            "StakeManager: max commission rate exceeded"
        );

        Validator memory validator = _getValidatorInfoByIndex(validatorId_);
        _onlyActivated(validator);
        _onlyValidatorShareContractExist(validator);

        stakeManagerStorage.setValidatorInfoCommissionRateByIndex(
            validatorId_,
            newCommissionRate_
        );
        validator.commissionRate = uint16(newCommissionRate_); // will not exceed
10000

        emit CommissionRateUpdated(
            validatorId_,
            newCommissionRate_,
            block.timestamp
        );
    }

    function _updateMinDelegatedCore(
        uint256 validatorId_,
        uint256 newMinDelegated_
    ) private {
        Validator memory validator = _getValidatorInfoByIndex(validatorId_);
        _onlyActivated(validator);
        _onlyValidatorShareContractExist(validator);

        IValidatorShare(validator.validatorShareContract).updateMinDelegated(
            newMinDelegated_
        );

        emit MinDelegatedUpdated(
            validatorId_,
            newMinDelegated_,
            block.timestamp
        );
    }

    function _updateValidatorDelegationCore(
        uint256 validatorId_,
        bool delegation_
    ) private {
        Validator memory validator = _getValidatorInfoByIndex(validatorId_);
        _onlyActivated(validator);
        _onlyValidatorShareContractExist(validator);

        _updateDelegation(validator.validatorShareContract, delegation_);
        emit DelegationUpdated(validatorId_, delegation_, block.timestamp);
    }

    function _updateRewards(
        uint256 validatorId_,
        Validator memory validator_,
        uint256 reward_
    ) private {
        // if the reward is more than the total supply then something is prob wrong
        require(
            reward_ <= type(uint128).max,
            "StakeManager: so rich, much reward"
        );

        uint256 validatorReward;
        uint256 delegatorsReward;
        uint256 infraCommission;
        uint256 validatorCommission;
        uint256 delegatorCommission;
        _incDescTotalRewards(reward_, ChangeMode.Increase);

        if (validator_.validatorShareContract != address(0)) {
            (
                validatorReward,
                delegatorsReward,
                infraCommission,
                validatorCommission,
                delegatorCommission
            ) = _increaseValidatorRewardWithDelegation(
                validatorId_,
                validator_,
                uint128(reward_)
            );
        } else {
            _increaseValidatorReward(
                validatorId_,
                validator_,
                uint128(reward_)
            );
            validatorReward = reward_;
        }

        emit DistributeRewards(
            validatorId_,
            reward_,
            validatorReward,
            delegatorsReward,
            infraCommission,
            validatorCommission,
            delegatorCommission
        );
    }

    function _increaseValidatorReward(
        uint256 validatorId_,
        Validator memory validator_,
        uint128 reward_
    ) private {
        if (reward_ > 0) {
            _incDescValidatorInfoRewardByIndex(
                validatorId_,
                reward_,
                ChangeMode.Increase
            );
            validator_.reward += reward_;
        }
    }

    function _increaseValidatorRewardWithDelegation(
        uint256 validatorId_,
        Validator memory validator_,
        uint128 reward_
    )
        private
        returns (
            uint128 validatorReward,
            uint128 delegatorsReward,
            uint128 infraCommission,
            uint128 validatorCommission,
            uint128 delegatorCommission
        )
    {
        (
            validatorReward,
            delegatorsReward,
            infraCommission,
            validatorCommission,
            delegatorCommission
        ) = _getValidatorAndDelegationReward(
            validator_.infraCommissionRate,
            validator_.commissionRate,
            validator_.amount,
            reward_,
            validator_.delegatedAmount + validator_.amount
        );

        if (validatorReward > 0) {
            _incDescValidatorInfoRewardByIndex(
                validatorId_,
                validatorReward,
                ChangeMode.Increase
            );
            validator_.reward += validatorReward;
        }

        if (delegatorsReward > 0) {
            _incDescValidatorInfoDelegatorsRewardByIndex(
                validatorId_,
                delegatorsReward,
                ChangeMode.Increase
            );
            validator_.delegatorsReward += delegatorsReward;
        }

        if (infraCommission > 0) {
            _incDescValidatorInfoInfraCommissionAmountByIndex(
                validatorId_,
                infraCommission,
                ChangeMode.Increase
            );
            validator_.infraCommissionAmount += infraCommission;
        }

        if (validatorCommission > 0) {
            _incDescValidatorInfoValidatorCommissionAmountByIndex(
                validatorId_,
                validatorCommission,
                ChangeMode.Increase
            );
            validator_.validatorCommissionAmount += validatorCommission;
        }

        if (delegatorCommission > 0) {
            _incDescValidatorInfoDelegatorCommissionAmountByIndex(
                validatorId_,
                delegatorCommission,
                ChangeMode.Increase
            );
            validator_.delegatorCommissionAmount += delegatorCommission;
        }
    }

    function _liquidateRewards(
        uint256 validatorId_,
        Validator memory validator_,
        address payable validatorUser_,
        LiquidateMode liquidateMode_
    ) private {
        _onlyActivated(validator_);

        uint256 reward = 0;

        if (
            liquidateMode_ == LiquidateMode.ValidatorReward ||
            liquidateMode_ == LiquidateMode.All
        ) {
            _setValidatorInfoRewardByIndex(validatorId_, 0);
            reward += validator_.reward;
            emit ClaimRewards(
                validatorId_,
                validator_.reward,
                LiquidateMode.ValidatorReward
            );
            validator_.reward = 0;
        }
        if (
            liquidateMode_ == LiquidateMode.ValidatorCommission ||
            liquidateMode_ == LiquidateMode.DelegatorCommission ||
            liquidateMode_ == LiquidateMode.All
        ) {
            _setValidatorInfoValidatorCommissionAmountByIndex(validatorId_, 0);
            reward += validator_.validatorCommissionAmount;
            emit ClaimRewards(
                validatorId_,
                validator_.validatorCommissionAmount,
                LiquidateMode.ValidatorCommission
            );
            validator_.validatorCommissionAmount = 0;

            _setValidatorInfoDelegatorCommissionAmountByIndex(validatorId_, 0);
            reward += validator_.delegatorCommissionAmount;
            emit ClaimRewards(
                validatorId_,
                validator_.delegatorCommissionAmount,
                LiquidateMode.DelegatorCommission
            );
            validator_.delegatorCommissionAmount = 0;
        }

        _transferToken(validatorUser_, reward, TransferTokenMode.Reward);
    }

    function _handleMinimalValidatorState(
        Validator memory validator_
    ) private returns (bool isInMinimalValidatorSet) {
        (, bool valid) = _getMinimalValidatorListIndex(validator_.signer);
        if (
            validator_.amount == 0 || validator_.amount < validator_.minDeposit
        ) {
            if (valid) {
                _removeMinimalValidatorList(validator_.signer);
            }
            isInMinimalValidatorSet = false;
        } else {
            if (!valid) {
                stakeManagerStorage.addMinimalValidator(validator_.signer);
            }
            isInMinimalValidatorSet = true;
        }
    }

    function _transferInKKUB(address from_, uint256 amount_) private {
        transferRouter.transferFromKKUB(from_, address(this), amount_);
        kkub.withdraw(amount_);
    }

    function _transferToken(
        address destination_,
        uint256 amount_,
        TransferTokenMode mode_
    ) private returns (bool) {
        if (mode_ == TransferTokenMode.Staked) {
            _incDescTotalStaked(amount_, ChangeMode.Decrease);
        } else if (mode_ == TransferTokenMode.Reward) {
            _incDescTotalRewards(amount_, ChangeMode.Decrease);
            _incDescTotalRewardsLiquidated(amount_, ChangeMode.Increase);
        }

        _withdrawFund(destination_, amount_);
        return true;
    }

    function _emitValidatorStaked(
        uint256 validatorId_,
        Validator memory validator_,
        uint256 minDelegated_,
        address officialPool_
    ) private {
        emit ValidatorStaked(
            validatorId_,
            validator_.signer,
            validator_.amount,
            validator_.validatorShareContract,
            validator_.status,
            minDelegated_,
            validator_.validatorShareContract != address(0),
            validator_.signer == officialPool_
        );
    }

    /*********************** internal read functions **************************/

    function _onlyStaker(uint256 validatorId_, address user_) private view {
        require(_ownerOf(validatorId_) == user_, "StakeManager: only staker");
    }

    function _onlyDelegation(uint256 validatorId_) private view {
        require(
            _getValidatorInfoValidatorShareContractByIndex(validatorId_) ==
                msg.sender,
            "StakeManager: only validator share contract"
        );
    }

    function _onlySlashManager() private view {
        require(
            msg.sender == _slashManager(),
            "StakeManager: only slash manager allowed"
        );
    }

    function _onlyActivated(Validator memory validator_) private pure {
        require(_isActivated(validator_.status), "StakeManager: not activated");
    }

    function _onlyValidatorShareContractExist(
        Validator memory validator_
    ) private pure {
        require(
            validator_.validatorShareContract != address(0x0),
            "StakeManager: delegation is disabled"
        );
    }

    function _isSoloPoolAddress(
        Status status_,
        address validatorShareContract_
    ) private pure returns (bool) {
        return _isActivated(status_) && validatorShareContract_ == address(0x0);
    }

    function _isActivated(Status status_) private pure returns (bool) {
        return (status_ == Status.Active);
    }

    function _isActivated(address validator_) private view returns (bool) {
        return _isActivated(_getValidatorInfoStatus(validator_));
    }

    /// @dev returns validatorReward, delegatorReward, infraCommission, commission
    function _getValidatorAndDelegationReward(
        uint16 infraCommissionRate_,
        uint16 commissionRate_,
        uint128 validatorsStake_,
        uint128 reward_,
        uint128 combinedStakePower_
    )
        private
        pure
        returns (
            uint128 actualValidatorReward,
            uint128 actualDelegatorReward,
            uint128 infraCommission,
            uint128 validatorCommission,
            uint128 delegatorCommission
        )
    {
        if (combinedStakePower_ == 0) {
            return (0, 0, 0, 0, 0);
        }

        // without any cuts
        uint128 originalValidatorReward = uint128(
            (uint256(validatorsStake_) * reward_) / combinedStakePower_
        ); // combinedStakePower_ >= validatorsStake_
        uint128 originalDelegatorReward = reward_ - originalValidatorReward;

        // cuts will be applied here
        actualValidatorReward = originalValidatorReward;
        actualDelegatorReward = originalDelegatorReward;

        if (infraCommissionRate_ > 0) {
            {
                uint128 validatorInfraCommission = uint128(
                    (uint256(originalValidatorReward) * infraCommissionRate_) /
                        MAX_RATE
                );
                uint128 delegatorInfraCommission = uint128(
                    (uint256(originalDelegatorReward) * infraCommissionRate_) /
                        MAX_RATE
                );

                actualValidatorReward -= validatorInfraCommission;
                actualDelegatorReward -= delegatorInfraCommission;

                infraCommission +=
                    validatorInfraCommission +
                    delegatorInfraCommission;
            }
        }

        if (commissionRate_ > 0) {
            validatorCommission = uint128(
                (uint256(originalValidatorReward) * commissionRate_) / MAX_RATE
            );
            delegatorCommission = uint128(
                (uint256(originalDelegatorReward) * commissionRate_) / MAX_RATE
            );

            actualValidatorReward -= validatorCommission;
            actualDelegatorReward -= delegatorCommission;
        }

        return (
            actualValidatorReward,
            actualDelegatorReward,
            infraCommission,
            validatorCommission,
            delegatorCommission
        );
    }

    function _getMinimalValidators(
        address[] memory minimalValidatorAddresses_,
        uint256[] memory validatorPowers_
    ) private pure returns (MinimalValidator[] memory) {
        MinimalValidator[] memory res = new MinimalValidator[](
            minimalValidatorAddresses_.length
        );
```

```
        for (uint256 i = 0; i < minimalValidatorAddresses_.length; i++) {
            res[i] = MinimalValidator(
                minimalValidatorAddresses_[i],
                validatorPowers_[i]
            );
        }

        return res;
    }
}
```

3. BKCValidatorSet

```
// * Copyright 2023 Bitkub Blockchain Technology Co., Ltd. - All Rights Reserved.
// * This code is proprietary.
// * Unauthorized copying, modification or distribution of this code, via any
medium, is strictly prohibited without express permission.

pragma solidity ^0.8.0;

contract SpanBase {
    uint256 public immutable initialSpanBlock;
    uint256 public stableSpanBlock;

    uint256 public constant SPAN = 50;

    constructor(uint256 initialSpanBlock_) {
        initialSpanBlock = initialSpanBlock_;

        (, uint256 initialSpanEnd) = getSpanRange(0);
        stableSpanBlock = initialSpanEnd + 1;
    }

    /// @notice Calculate span number from block number
    /// @param number_ block number
    /// @return span number
    function getSpanByBlock(uint256 number_) public view returns (uint256) {
        if (number_ < stableSpanBlock) {
            return 0;
        }
        return (number_ - stableSpanBlock) / SPAN + 1;
    }

    /// @notice Calculate current span number
    /// @return span number
    function currentSpanNumber() public view returns (uint256) {
        return getSpanByBlock(block.number);
    }

    /// @notice Get span range
    /// @param number_ span number
    function getSpanRange(
        uint256 number_
    ) public view returns (uint256 startBlock, uint256 endBlock) {
        if (number_ == 0) {
            // Transition - Calculates a span or a peroid of
            // the consensus transition
            // Remaining size of a first single span
            uint256 size = SPAN - (initialSpanBlock % SPAN);

            startBlock = initialSpanBlock;

            // Find the edge of the first span (ends with 49, 99; SPAN=50)
            endBlock = size - 1 + initialSpanBlock;

            // If the first span does not contains a commitSpan block
            // which is always at the block number 26,76. Expand one
            // more span to the transition.
            if (size < SPAN / 2 - 1) {
                endBlock += SPAN;
            }
        } else {
            // Stable -  Just a general span calculation
            startBlock = stableSpanBlock + ((number_ - 1) * SPAN);
            endBlock = startBlock + SPAN - 1;
        }

        return (startBlock, endBlock);
    }

    /// @notice Get span's commitment block
    /// @param number_ span number
    /// @return blockNumber (commitment block)
    function getCommitmentBlock(uint256 number_) public view returns (uint256) {
        if (number_ == 0) {
            // case: transition period
            return calculateTransitionSpanCommitmentBlock();
        }
        // stableSpanBlock + number of blocks passed + half span
        return stableSpanBlock + ((number_ - 1) * SPAN) + (SPAN / 2 + 1);
    }

    /// @notice Get span's commitment block in transition period
    /// @return blockNumber (commitment block)
    function calculateTransitionSpanCommitmentBlock()
        public
        view
        returns (uint256)
    {
        if (initialSpanBlock % SPAN > SPAN / 2 + 1) {
            return
                initialSpanBlock +
                (SPAN - (initialSpanBlock % SPAN)) +
                (SPAN / 2 + 1);
        } else {
            return stableSpanBlock - (SPAN / 2) + 1;
        }
    }
}

enum LiquidateMode {
    All,
    ValidatorReward,
    ValidatorCommission,
    DelegatorCommission
}

enum TransferTokenMode {
    None,
    Staked,
    Reward
}

enum ChangeMode {
    Increase,
    Decrease
}

enum Status {
    Uninitialized,
    Active,
    Unstaked
}
```

```
enum SlashErrorCode {
    Uninitialized,
    StakeAmountNotEnough,
    OfficialPool,
    NotPool,
    InactivatedPool
}

struct Validator {
    uint128 amount;
    uint128 delegatedAmount;
    uint128 reward;
    uint128 delegatorsReward;
    uint128 infraCommissionAmount;
    uint128 validatorCommissionAmount;
    uint128 delegatorCommissionAmount;
    uint128 minDeposit;
    address signer;
    address validatorShareContract;
    Status status;
    uint16 infraCommissionRate;
    uint16 commissionRate;
}

struct MinimalValidator {
    address signer;
    uint256 power;
}

interface IStakeManager {
    function setOfficialPoolStaker(address officialPoolStaker_) external;

    function transferUnallocatedReward(
        address payable to,
        uint256 amount
    ) external returns (bool);

    function transferInfraCommission(
        uint256[] memory validatorIds_,
        uint256[] memory amounts_,
        address payable to_
    ) external;

    function transferFunds(
        uint256 validatorId,
        uint256 amount,
        address delegator,
        TransferTokenMode mode
    ) external returns (bool);

    function delegationDeposit(
        uint256 validatorId
    ) external payable returns (bool);

    function stake(address signer, bool delegation) external payable;

    function stake(
        address signer,
        bool delegation,
        uint256 amount,
        address bitkubNext
    ) external;

    function unstake(uint256 validatorId) external;

    function unstake(uint256 validatorId, address bitkubNext) external;

    function restake(uint256 validatorId) external payable;

    function restake(
        uint256 validatorId,
        uint256 amount,
        address bitkubNext
    ) external;

    function distributeReward() external payable;

    function claimRewards(uint256 validatorId) external;

    function claimRewards(uint256 validatorId, address bitkubNext) external;

    function claimCommissionRewards(uint256 validatorId) external;

    function claimCommissionRewards(
        uint256 validatorId,
        address bitkubNext
    ) external;

    function updateInfraCommissionRate(
        uint256 validatorId_,
        uint256 newInfraCommissionRate_
    ) external;

    function updateCommissionRate(
        uint256 validatorId,
        uint256 newCommissionRate
    ) external;

    function updateCommissionRate(
        uint256 validatorId,
        uint256 newCommissionRate,
        address bitkubNext
    ) external;

    function updateMinDelegated(
        uint256 validatorId,
        uint256 newMinDelegated
    ) external;

    function updateMinDelegated(
        uint256 validatorId,
        uint256 newMinDelegated,
        address bitkubNext
    ) external;

    function withdrawDelegatorsReward(
        uint256 validatorId
    ) external returns (uint256);

    function updateValidatorDelegation(
        uint256 validatorId,
        bool delegation
    ) external;

    function updateValidatorDelegation(
        uint256 validatorId,
        bool delegation,
        address bitkubNext
    ) external;

    function slash(address _signer) external returns (bool);

    function updateValidatorState(uint256 validatorId, int256 amount) external;

    function getMinimalValidators()
```

```
        external
        view
        returns (MinimalValidator[] memory);

    function getMinimalValidatorsByPage(
        uint256 page_,
        uint256 limit_
    ) external view returns (MinimalValidator[] memory);

    function getMinimalValidatorsLength() external view returns (uint256);

    function getValidatorId(address _signer) external view returns (uint256);

    function getMinimalValidatorIndex(
        address _signer
    ) external view returns (uint256);

    function getValidators() external view returns (address[] memory);
}

interface IStakeManagerStorage {
    function stakeManager() external view returns (address);

    function nftContract() external view returns (address);

    function validatorShareFactory() external view returns (address);

    function slashManager() external view returns (address);

    function soloLimit() external view returns (uint256);

    function soloAmount() external view returns (uint256);

    function poolLimit() external view returns (uint256);

    function poolAmount() external view returns (uint256);

    function officialLimit() external view returns (uint256);

    function officialAmount() external view returns (uint256);

    function defaultInfraCommissionRate() external view returns (uint256);

    function totalStaked() external view returns (uint256);
    function totalRewards() external view returns (uint256);

    function totalRewardsLiquidated() external view returns (uint256);

    function soloSlashRate() external view returns (uint256);

    function poolSlashAmount() external view returns (uint256);

    function unallocatedReward() external view returns (uint256);

    function officialPool() external view returns (address);

    function getNewOfficialPoolValidBlock() external view returns (uint256);

    function getOldNewOfficialSignerAndValidBlock()
        external
        view
        returns (address, address, uint256);

    function changeOfficialSigner(
        address newOfficialSigner_,
        string memory checksummedNewOfficialSigner_
    ) external;

    function setStakeManager(address stakeManager_) external;

    function setValidatorShareFactory(address validatorShareFactory_) external;

    function setSlashManager(address slashManager_) external;

    function setSoloLimit(uint256 val_) external;

    function incDescSoloLimit(uint256 val_, ChangeMode mode_) external;

    function setSoloAmount(uint256 val_) external;

    function incDescSoloAmount(uint256 val_, ChangeMode mode_) external;

    function setPoolLimit(uint256 val_) external;

    function incDescPoolLimit(uint256 val_, ChangeMode mode_) external;

    function setPoolAmount(uint256 val_) external;

    function incDescPoolAmount(uint256 val_, ChangeMode mode_) external;
    function setOfficialLimit(uint256 val_) external;

    function incDescOfficialLimit(uint256 val_, ChangeMode mode_) external;

    function setOfficialAmount(uint256 val_) external;

    function incDescOfficialAmount(uint256 val_, ChangeMode mode_) external;

    function setDefaultInfraCommissionRate(uint256 val_) external;

    function incDescDefaultInfraCommissionRate(
        uint256 val_,
        ChangeMode mode_
    ) external;

    function setTotalStaked(uint256 val_) external;

    function incDescTotalStaked(uint256 val_, ChangeMode mode_) external;

    function setTotalRewards(uint256 val_) external;

    function incDescTotalRewards(uint256 val_, ChangeMode mode_) external;

    function setTotalRewardsLiquidated(uint256 val_) external;

    function incDescTotalRewardsLiquidated(
        uint256 val_,
        ChangeMode mode_
    ) external;

    function setPoolSlashAmount(uint256 val_) external;

    function incDescPoolSlashAmount(uint256 val_, ChangeMode mode_) external;

    function setUnallocatedReward(uint256 val_) external;

    function incDescUnallocatedReward(uint256 val_, ChangeMode mode_) external;

    function addValidator(
        address key_,
        Validator memory val_
    ) external returns (uint256);

    function setValidatorIds(
        address key_,
        uint256[] memory validatorIds_
    ) external;

    function setValidatorInfo(address key_, Validator memory val_) external;

    function setValidatorInfoByIndex(
        uint256 index_,
        Validator memory val_
    ) external;

    function setValidatorInfoMinDeposit(address key_, uint256 val_) external;

    function setValidatorInfoMinDepositByIndex(
        uint256 index_,
        uint256 val_
    ) external;

    function incDescValidatorInfoMinDeposit(
        address key_,
        uint256 val_,
        ChangeMode mode_
    ) external;

    function incDescValidatorInfoMinDepositByIndex(
        uint256 index_,
        uint256 val_,
        ChangeMode mode_
    ) external;

    function setValidatorInfoAmount(address key_, uint256 val_) external;

    function setValidatorInfoAmountByIndex(
        uint256 index_,
        uint256 val_
    ) external;

    function incDescValidatorInfoAmount(
        address key_,
        uint256 val_,
        ChangeMode mode_
    ) external;

    function incDescValidatorInfoAmountByIndex(
        uint256 index_,
        uint256 val_,
        ChangeMode mode_
    ) external;

    function setValidatorInfoReward(address key_, uint256 val_) external;

    function setValidatorInfoRewardByIndex(
        uint256 index_,
        uint256 val_
    ) external;

    function incDescValidatorInfoReward(
        address key_,
        uint256 val_,
        ChangeMode mode_
    ) external;

    function incDescValidatorInfoRewardByIndex(
        uint256 index_,
        uint256 val_,
        ChangeMode mode_
    ) external;

    function setValidatorInfoSigner(address key_, address val_) external;

    function setValidatorInfoSignerByIndex(
        uint256 index_,
        address val_
    ) external;

    function setValidatorInfoValidatorShareContract(
        address key_,
        address val_
    ) external;

    function setValidatorInfoValidatorShareContractByIndex(
        uint256 index_,
        address val_
    ) external;

    function setValidatorInfoStatus(address key_, Status val_) external;

    function setValidatorInfoStatusByIndex(
        uint256 index_,
        Status val_
    ) external;

    function setValidatorInfoInfraCommissionRate(
        address key_,
        uint256 val_
    ) external;

    function setValidatorInfoInfraCommissionRateByIndex(
        uint256 index_,
        uint256 val_
    ) external;

    function incDescValidatorInfoInfraCommissionRate(
        address key_,
        uint256 val_,
        ChangeMode mode_
    ) external;

    function incDescValidatorInfoInfraCommissionRateByIndex(
        uint256 index_,
        uint256 val_,
        ChangeMode mode_
    ) external;

    function setValidatorInfoInfraCommissionAmount(
        address key_,
        uint256 val_
    ) external;

    function setValidatorInfoInfraCommissionAmountByIndex(
        uint256 index_,
        uint256 val_
    ) external;

    function incDescValidatorInfoInfraCommissionAmount(
        address key_,
        uint256 val_,
        ChangeMode mode_
    ) external;

    function incDescValidatorInfoInfraCommissionAmountByIndex(
        uint256 index_,
        uint256 val_,
        ChangeMode mode_
    ) external;
    function setValidatorInfoCommissionRate(
        address key_,
        uint256 val_
    ) external;

    function setValidatorInfoCommissionRateByIndex(
        uint256 index_,
        uint256 val_
    ) external;

    function incDescValidatorInfoCommissionRate(
        address key_,
        uint256 val_,
        ChangeMode mode_
    ) external;

    function incDescValidatorInfoCommissionRateByIndex(
        uint256 index_,
        uint256 val_,
        ChangeMode mode_
    ) external;

    function setValidatorInfoValidatorCommissionAmount(
        address key_,
        uint256 val_
    ) external;

    function setValidatorInfoValidatorCommissionAmountByIndex(
        uint256 index_,
        uint256 val_
    ) external;

    function incDescValidatorInfoValidatorCommissionAmount(
        address key_,
        uint256 val_,
        ChangeMode mode_
    ) external;

    function incDescValidatorInfoValidatorCommissionAmountByIndex(
        uint256 index_,
        uint256 val_,
        ChangeMode mode_
    ) external;
    function setValidatorInfoDelegatorCommissionAmount(
        address key_,
        uint256 val_
    ) external;

    function setValidatorInfoDelegatorCommissionAmountByIndex(
        uint256 index_,
        uint256 val_
    ) external;

    function incDescValidatorInfoDelegatorCommissionAmount(
        address key_,
        uint256 val_,
        ChangeMode mode_
    ) external;

    function incDescValidatorInfoDelegatorCommissionAmountByIndex(
        uint256 index_,
        uint256 val_,
        ChangeMode mode_
    ) external;

    function setValidatorInfoDelegatorsReward(
        address key_,
        uint256 val_
    ) external;

    function setValidatorInfoDelegatorsRewardByIndex(
        uint256 index_,
        uint256 val_
    ) external;

    function incDescValidatorInfoDelegatorsReward(
        address key_,
        uint256 val_,
        ChangeMode mode_
    ) external;

    function incDescValidatorInfoDelegatorsRewardByIndex(
        uint256 index_,
        uint256 val_,
        ChangeMode mode_
    ) external;

    function setValidatorInfoDelegatedAmount(
        address key_,
        uint256 val_
    ) external;

    function setValidatorInfoDelegatedAmountByIndex(
        uint256 index_,
        uint256 val_
    ) external;

    function incDescValidatorInfoDelegatedAmount(
        address key_,
        uint256 val_,
        ChangeMode mode_
    ) external;

    function incDescValidatorInfoDelegatedAmountByIndex(
        uint256 index_,
        uint256 val_,
        ChangeMode mode_
    ) external;

    function getValidatorInfo(
        address key_
    ) external view returns (Validator memory);

    function getValidatorInfoByIndex(
        uint256 index_
    ) external view returns (Validator memory);

    function getValidatorInfoMinDeposit(
        address key_
    ) external view returns (uint256);

    function getValidatorInfoMinDepositByIndex(
        uint256 index_
    ) external view returns (uint256);

    function getValidatorInfoAmount(
        address key_
    ) external view returns (uint256);

    function getValidatorInfoAmountByIndex(
        uint256 index_
    ) external view returns (uint256);
    function getValidatorInfoReward(
        address key_
    ) external view returns (uint256);

    function getValidatorInfoRewardByIndex(
        uint256 index_
    ) external view returns (uint256);

    function getValidatorInfoSigner(
        address key_
    ) external view returns (address);

    function getValidatorInfoSignerByIndex(
        uint256 index_
    ) external view returns (address);

    function getValidatorInfoValidatorShareContract(
        address key_
    ) external view returns (address);

    function getValidatorInfoValidatorShareContractByIndex(
        uint256 index_
    ) external view returns (address);

    function getValidatorInfoStatus(
        address key_
    ) external view returns (Status);

    function getValidatorInfoStatusByIndex(
        uint256 index_
    ) external view returns (Status);

    function getValidatorInfoInfraCommissionRate(
        address key_
    ) external view returns (uint256);

    function getValidatorInfoInfraCommissionRateByIndex(
        uint256 index_
    ) external view returns (uint256);

    function getValidatorInfoInfraCommissionAmount(
        address key_
    ) external view returns (uint256);

    function getValidatorInfoInfraCommissionAmountByIndex(
        uint256 index_
    ) external view returns (uint256);

    function getValidatorInfoCommissionRate(
        address key_
    ) external view returns (uint256);

    function getValidatorInfoCommissionRateByIndex(
        uint256 index_
    ) external view returns (uint256);

    function getValidatorInfoValidatorCommissionAmount(
        address key_
    ) external view returns (uint256);

    function getValidatorInfoValidatorCommissionAmountByIndex(
        uint256 index_
    ) external view returns (uint256);

    function getValidatorInfoDelegatorCommissionAmount(
        address key_
    ) external view returns (uint256);

    function getValidatorInfoDelegatorCommissionAmountByIndex(
        uint256 index_
    ) external view returns (uint256);

    function getValidatorInfoDelegatorsReward(
        address key_
    ) external view returns (uint256);

    function getValidatorInfoDelegatorsRewardByIndex(
        uint256 index_
    ) external view returns (uint256);

    function getValidatorInfoDelegatedAmount(
        address key_
    ) external view returns (uint256);

    function getValidatorInfoDelegatedAmountByIndex(
        uint256 index_
    ) external view returns (uint256);

    function getValidatorListLength() external view returns (uint256);
    function isInValidatorList(address val_) external view returns (bool);

    function getValidatorCurrentIndex(
        address val_,
        bool revertIfNotFound_
    ) external view returns (uint256, bool);

    function getValidatorByIndex(
        uint256 index_
    ) external view returns (address);

    function getAllValidator() external view returns (address[] memory);

    function getValidatorByPage(
        uint256 page,
        uint256 limit
    ) external view returns (address[] memory);

    function getValidatorIndexLength(
        address val_
    ) external view returns (uint256);

    function getValidatorIndexByIndex(
        address val_,
        uint256 index_
    ) external view returns (uint256);

    function getMinimalValidatorListLength() external view returns (uint256);

    function isInMinimalValidatorList(
        address val_
    ) external view returns (bool);

    function isInMinimalValidatorListByValidatorId(
        uint256 validatorId_
    ) external view returns (bool);

    function getMinimalValidatorIndex(
        address val_,
        bool revertIfNotFound_
    ) external view returns (uint256, bool);

    function getMinimalValidatorByIndex(
        uint256 index_
    ) external view returns (address);
    function getAllMinimalValidators() external view returns (address[] memory);

    function getMinimalValidatorsByPage(
        uint256 page,
        uint256 limit
    ) external view returns (address[] memory);

    function addMinimalValidator(address val_) external;

    function removeMinimalValidator(address val_) external;

    function removeMinimalValidatorByIndex(uint256 index_) external;

    function getMinimalValidatorsWithValidatorPowerByPage(
        uint256 page_,
        uint256 limit_
    ) external view returns (address[] memory, uint256[] memory);
}

/*
 * @author Hamdi Allam hamdi.allam97@gmail.com
 * Please reach out with any questions or concerns
 */

library RLPReader {
    uint8 constant STRING_SHORT_START = 0x80;
    uint8 constant STRING_LONG_START = 0xb8;
    uint8 constant LIST_SHORT_START = 0xc0;
    uint8 constant LIST_LONG_START = 0xf8;
    uint8 constant WORD_SIZE = 32;

    struct RLPItem {
        uint256 len;
        uint256 memPtr;
    }

    struct Iterator {
        RLPItem item; // Item that's being iterated over.
        uint256 nextPtr; // Position of the next item in the list.
    }

    /*
     * @dev Returns the next element in the iteration. Reverts if it has not next
element.
```

```
     * @param self The iterator.
     * @return The next element in the iteration.
     */
    function next(Iterator memory self) internal pure returns (RLPItem memory) {
        require(hasNext(self));

        uint256 ptr = self.nextPtr;
        uint256 itemLength = _itemLength(ptr);
        self.nextPtr = ptr + itemLength;

        return RLPItem(itemLength, ptr);
    }

    /*
     * @dev Returns true if the iteration has more elements.
     * @param self The iterator.
     * @return true if the iteration has more elements.
     */
    function hasNext(Iterator memory self) internal pure returns (bool) {
        RLPItem memory item = self.item;
        return self.nextPtr < item.memPtr + item.len;
    }

    /*
     * @param item RLP encoded bytes
     */
    function toRlpItem(bytes memory item) internal pure returns (RLPItem memory) {
        uint256 memPtr;
        assembly {
            memPtr := add(item, 0x20)
        }

        return RLPItem(item.length, memPtr);
    }

    /*
     * @dev Create an iterator. Reverts if item is not a list.
     * @param self The RLP item.
     * @return An 'Iterator' over the item.
     */
    function iterator(RLPItem memory self) internal pure returns (Iterator memory) {
        require(isList(self));

        uint256 ptr = self.memPtr + _payloadOffset(self.memPtr);
        return Iterator(self, ptr);
    }

    /*
     * @param the RLP item.
     */
    function rlpLen(RLPItem memory item) internal pure returns (uint256) {
        return item.len;
    }

    /*
     * @param the RLP item.
     * @return (memPtr, len) pair: location of the item's payload in memory.
     */
    function payloadLocation(RLPItem memory item) internal pure returns (uint256,
uint256) {
        uint256 offset = _payloadOffset(item.memPtr);
        uint256 memPtr = item.memPtr + offset;
        uint256 len = item.len - offset; // data length
        return (memPtr, len);
    }

    /*
     * @param the RLP item.
     */
    function payloadLen(RLPItem memory item) internal pure returns (uint256) {
        (, uint256 len) = payloadLocation(item);
        return len;
    }

    /*
     * @param the RLP item containing the encoded list.
     */
    function toList(RLPItem memory item) internal pure returns (RLPItem[] memory) {
        require(isList(item));

        uint256 items = numItems(item);
        RLPItem[] memory result = new RLPItem[](items);

        uint256 memPtr = item.memPtr + _payloadOffset(item.memPtr);
        uint256 dataLen;
        for (uint256 i = 0; i < items; i++) {
            dataLen = _itemLength(memPtr);
            result[i] = RLPItem(dataLen, memPtr);
            memPtr = memPtr + dataLen;
        }
        return result;
    }

    // @return indicator whether encoded payload is a list. negate this function
call for isData.
    function isList(RLPItem memory item) internal pure returns (bool) {
        if (item.len == 0) return false;

        uint8 byte0;
        uint256 memPtr = item.memPtr;
        assembly {
            byte0 := byte(0, mload(memPtr))
        }

        if (byte0 < LIST_SHORT_START) return false;
        return true;
    }

    /*
     * @dev A cheaper version of keccak256(toRlpBytes(item)) that avoids copying
memory.
     * @return keccak256 hash of RLP encoded bytes.
     */
    function rlpBytesKeccak256(RLPItem memory item) internal pure returns (bytes32)
{
        uint256 ptr = item.memPtr;
        uint256 len = item.len;
        bytes32 result;
        assembly {
            result := keccak256(ptr, len)
        }
        return result;
    }

    /*
     * @dev A cheaper version of keccak256(toBytes(item)) that avoids copying
memory.
     * @return keccak256 hash of the item payload.
     */
    function payloadKeccak256(RLPItem memory item) internal pure returns (bytes32) {
        (uint256 memPtr, uint256 len) = payloadLocation(item);
        bytes32 result;
        assembly {
            result := keccak256(memPtr, len)
        }
        return result;
    }

    /** RLPItem conversions into data types **/

    // @returns raw rlp encoding in bytes
    function toRlpBytes(RLPItem memory item) internal pure returns (bytes memory) {
        bytes memory result = new bytes(item.len);
        if (result.length == 0) return result;

        uint256 ptr;
        assembly {
            ptr := add(0x20, result)
        }

        copy(item.memPtr, ptr, item.len);
        return result;
    }

    // any non-zero byte except "0x80" is considered true
    function toBoolean(RLPItem memory item) internal pure returns (bool) {
        require(item.len == 1);
        uint256 result;
        uint256 memPtr = item.memPtr;
        assembly {
            result := byte(0, mload(memPtr))
        }

        // SEE Github Issue #5.
        // Summary: Most commonly used RLP libraries (i.e Geth) will encode
        // "0" as "0x80" instead of as "0". We handle this edge case explicitly
        // here.
        if (result == 0 || result == STRING_SHORT_START) {
            return false;
        } else {
            return true;
        }
    }

    function toAddress(RLPItem memory item) internal pure returns (address) {
        // 1 byte for the length prefix
        require(item.len == 21);

        return address(uint160(toUint(item)));
    }

    function toUint(RLPItem memory item) internal pure returns (uint256) {
        require(item.len > 0 && item.len <= 33);

        (uint256 memPtr, uint256 len) = payloadLocation(item);

        uint256 result;
        assembly {
            result := mload(memPtr)

            // shift to the correct location if neccesary
            if lt(len, 32) {
                result := div(result, exp(256, sub(32, len)))
            }
        }

        return result;
    }

    // enforces 32 byte length
    function toUintStrict(RLPItem memory item) internal pure returns (uint256) {
        // one byte prefix
        require(item.len == 33);

        uint256 result;
        uint256 memPtr = item.memPtr + 1;
        assembly {
            result := mload(memPtr)
        }

        return result;
    }

    function toBytes(RLPItem memory item) internal pure returns (bytes memory) {
        require(item.len > 0);

        (uint256 memPtr, uint256 len) = payloadLocation(item);
        bytes memory result = new bytes(len);

        uint256 destPtr;
        assembly {
            destPtr := add(0x20, result)
        }
        copy(memPtr, destPtr, len);
        return result;
    }

    /*
     * Private Helpers
     */

    // @return number of payload items inside an encoded list.
    function numItems(RLPItem memory item) private pure returns (uint256) {
        if (item.len == 0) return 0;

        uint256 count = 0;
        uint256 currPtr = item.memPtr + _payloadOffset(item.memPtr);
        uint256 endPtr = item.memPtr + item.len;
        while (currPtr < endPtr) {
            currPtr = currPtr + _itemLength(currPtr); // skip over an item
            count++;
        }

        return count;
    }

    // @return entire rlp item byte length
    function _itemLength(uint256 memPtr) private pure returns (uint256) {
        uint256 itemLen;
        uint256 byte0;
        assembly {
            byte0 := byte(0, mload(memPtr))
        }

        if (byte0 < STRING_SHORT_START) {
            itemLen = 1;
        } else if (byte0 < STRING_LONG_START) {
            itemLen = byte0 - STRING_SHORT_START + 1;
        } else if (byte0 < LIST_SHORT_START) {
            assembly {
                let byteLen := sub(byte0, 0xb7) // # of bytes the actual length is
                memPtr := add(memPtr, 1) // skip over the first byte

                /* 32 byte word size */
                let dataLen := div(mload(memPtr), exp(256, sub(32, byteLen))) //
right shifting to get the len
                itemLen := add(dataLen, add(byteLen, 1))
            }
        } else if (byte0 < LIST_LONG_START) {
            itemLen = byte0 - LIST_SHORT_START + 1;
        } else {
            assembly {
                let byteLen := sub(byte0, 0xf7)
                memPtr := add(memPtr, 1)

                let dataLen := div(mload(memPtr), exp(256, sub(32, byteLen))) //
right shifting to the correct length
                itemLen := add(dataLen, add(byteLen, 1))
            }
        }

        return itemLen;
    }

    // @return number of bytes until the data
    function _payloadOffset(uint256 memPtr) private pure returns (uint256) {
        uint256 byte0;
        assembly {
            byte0 := byte(0, mload(memPtr))
        }

        if (byte0 < STRING_SHORT_START) {
            return 0;
        } else if (byte0 < STRING_LONG_START || (byte0 >= LIST_SHORT_START && byte0
< LIST_LONG_START)) {
            return 1;
        } else if (byte0 < LIST_SHORT_START) {
            // being explicit
            return byte0 - (STRING_LONG_START - 1) + 1;
        } else {
            return byte0 - (LIST_LONG_START - 1) + 1;
        }
    }

    /*
     * @param src Pointer to source
     * @param dest Pointer to destination
     * @param len Amount of memory to copy from the source
     */
    function copy(uint256 src, uint256 dest, uint256 len) private pure {
        if (len == 0) return;

        // copy as many word sizes as possible
        for (; len >= WORD_SIZE; len -= WORD_SIZE) {
            assembly {
                mstore(dest, mload(src))
            }

            src += WORD_SIZE;
            dest += WORD_SIZE;
        }

        if (len > 0) {
            // left over bytes. Mask is used to remove unwanted bytes from the word
            uint256 mask = 256**(WORD_SIZE - len) - 1;
            assembly {
                let srcpart := and(mload(src), not(mask)) // zero out src
                let destpart := and(mload(dest), mask) // retrieve the bytes
                mstore(dest, or(destpart, srcpart))
            }
        }
    }
}

struct BKCValidator {
    address signer;
    uint256 power;
}

struct Span {
    uint256 number;
    uint256 startBlock;
    uint256 endBlock;
}

contract BKCValidatorSet is SpanBase {
    using RLPReader for bytes;
    using RLPReader for RLPReader.RLPItem;

    /// @notice Span size as a constant value
    /// @dev Span size is not allowed to be changed
    /// due to the mathematically span pre-calculation

    /// @notice Map each span number to a list of validators
    mapping(uint256 => bytes) public validators;

    /// @notice Map each span number to a detail of that span
    mapping(uint256 => Span) public spans;
    IStakeManagerStorage public immutable stakeManagerStorage;

    /// @notice Emit every commitment
    /// @dev emit from _commitSpan function
    /// @param span Span number
    /// @param start Span starts from block number
    /// @param end Span ends on block number
    event CommitSpan(
        uint256 indexed span,
        uint256 indexed start,
        uint256 indexed end
    );

    /// @notice Emit when list of validators are set in a span
    /// @dev emit from _setSpanValidators and _setSpanValidatorsFromBytes functions
    /// @param span Span number
    /// @param vbytes Number of validators in that span
    event SetValidators(uint256 indexed span, bytes vbytes);

    constructor(
        address stakeManagerStorage_,
        uint256 initialSpanBlock_,
        bytes memory initialVbytes_
    ) SpanBase(initialSpanBlock_) {
        stakeManagerStorage = IStakeManagerStorage(stakeManagerStorage_);

        // Begin state changes
        _commitSpan(0);
        _setSpanValidatorsFromBytes(0, initialVbytes_);
    }

    function commitSpan(bytes calldata validatorBytes_) external {
        uint256 current = currentSpanNumber();
        // security checks
        // only block.coinbase can call this function
        require(
            msg.sender == block.coinbase,
            "BKCValidatorSet: only coinbase allowed"
        );

        require(
            tx.gasprice == 0,
            "BKCValidatorSet: gas price should be zero (system tx)"
        );
        // only the (span / 2 + 1) block of each span are allowed.
        require(
            block.number == getCommitmentBlock(current),
            "BKCValidatorSet: only allowed on commitment block"
        );

        uint256 next = current + 1;
        _commitSpan(next);
        _setSpanValidatorsFromBytes(next, validatorBytes_);
    }

    function isValidator(
        uint256 span_,
        address signer_
    ) public view returns (bool) {
        BKCValidator[] memory vals = _rlpDecodeValidatorBytes(
            validators[span_]
        );
        for (uint256 i = 0; i < vals.length; i++) {
            if (vals[i].signer == signer_) {
                return true;
            }
        }
        return false;
    }

    function isCurrentValidator(address signer_) public view returns (bool) {
        return isValidator(currentSpanNumber(), signer_);
    }

    function getValidators(
        uint256 number_
    )
        public
        view
        returns (address[] memory, uint256[] memory, address[3] memory)
    {
        uint256 span = getSpanByBlock(number_);
        BKCValidator[] memory v = _rlpDecodeValidatorBytes(validators[span]);
        address[] memory addrs = new address[](v.length);
        uint256[] memory powers = new uint256[](v.length);
        for (uint256 i = 0; i < v.length; i++) {
            addrs[i] = v[i].signer;
            powers[i] = v[i].power;
        }
        address[3] memory addressToReturn = [
            getStakeManager(),
            getSlashManager(),
            getOfficialPool()
        ];
        return (addrs, powers, addressToReturn);
    }

    function getEligibleValidators()
        public
        view
        returns (MinimalValidator[] memory)
    {
        return
            IStakeManager(stakeManagerStorage.stakeManager())
                .getMinimalValidators();
    }

    function getStakeManager() public view returns (address) {
        return stakeManagerStorage.stakeManager();
    }

    function getSlashManager() public view returns (address) {
        return stakeManagerStorage.slashManager();
    }

    function getOfficialPool() public view returns (address) {
        (
            address oldOfficialSigner,
            address newOfficialSigner,
            uint256 validBlock
        ) = stakeManagerStorage.getOldNewOfficialSignerAndValidBlock();

        if (validBlock <= 1) {
            return newOfficialSigner;
        }

        if (block.number < validBlock - 2) {
            return oldOfficialSigner;
        }
        return newOfficialSigner;
    }

    function _commitSpan(uint256 number_) private {
        (uint256 startBlock, uint256 endBlock) = getSpanRange(number_);
        spans[number_] = Span({
            number: number_,
            startBlock: startBlock,
            endBlock: endBlock
        });

        emit CommitSpan(number_, startBlock, endBlock);
    }

    function _setSpanValidatorsFromBytes(
        uint256 number_,
        bytes memory vbytes_
    ) private {
        validators[number_] = vbytes_;
        emit SetValidators(number_, vbytes_);
    }

    function _rlpDecodeValidatorBytes(
        bytes memory vbytes_
    ) private pure returns (BKCValidator[] memory) {
        RLPReader.RLPItem[] memory validatorItems = vbytes_
            .toRlpItem()
            .toList();
        uint256 i;
        BKCValidator[] memory decoded = new BKCValidator[](
            validatorItems.length
        );
        for (i = 0; i < validatorItems.length; i++) {
            RLPReader.RLPItem[] memory v = validatorItems[i].toList();
            decoded[i] = BKCValidator({
                signer: v[0].toAddress(),
                power: v[1].toUint()
            });
        }
        return decoded;
    }
}
```

4. SlashManager

```
// * Copyright 2023 Bitkub Blockchain Technology Co., Ltd. - All Rights Reserved.
// * This code is proprietary.
// * Unauthorized copying, modification or distribution of this code, via any
```

```
medium, is strictly prohibited without express permission.

pragma solidity ^0.8.0;

enum LiquidateMode {
    All,
    ValidatorReward,
    ValidatorCommission,
    DelegatorCommission
}

enum TransferTokenMode {
    None,
    Staked,
    Reward
}

enum ChangeMode {
    Increase,
    Decrease
}

enum Status {
    Uninitialized,
    Active,
    Unstaked
}

enum SlashErrorCode {
    Uninitialized,
    StakeAmountNotEnough,
    OfficialPool,
    NotPool,
    InactivatedPool
}

struct Validator {
    uint128 amount;
    uint128 delegatedAmount;
    uint128 reward;
    uint128 delegatorsReward;
    uint128 infraCommissionAmount;
    uint128 validatorCommissionAmount;
    uint128 delegatorCommissionAmount;
    uint128 minDeposit;
    address signer;
    address validatorShareContract;
    Status status;
    uint16 infraCommissionRate;
    uint16 commissionRate;
}

struct MinimalValidator {
    address signer;
    uint256 power;
}

interface IStakeManager {
    function setOfficialPoolStaker(address officialPoolStaker_) external;

    function transferUnallocatedReward(
        address payable to,
        uint256 amount
    ) external returns (bool);

    function transferInfraCommission(
        uint256[] memory validatorIds_,
        uint256[] memory amounts_,
        address payable to_
    ) external;

    function transferFunds(
        uint256 validatorId,
        uint256 amount,
        address delegator,
        TransferTokenMode mode
    ) external returns (bool);

    function delegationDeposit(
        uint256 validatorId
    ) external payable returns (bool);

    function stake(address signer, bool delegation) external payable;

    function stake(
        address signer,
        bool delegation,
        uint256 amount,
        address bitkubNext
    ) external;
    function unstake(uint256 validatorId) external;

    function unstake(uint256 validatorId, address bitkubNext) external;

    function restake(uint256 validatorId) external payable;

    function restake(
        uint256 validatorId,
        uint256 amount,
        address bitkubNext
    ) external;

    function distributeReward() external payable;

    function claimRewards(uint256 validatorId) external;

    function claimRewards(uint256 validatorId, address bitkubNext) external;

    function claimCommissionRewards(uint256 validatorId) external;

    function claimCommissionRewards(
        uint256 validatorId,
        address bitkubNext
    ) external;

    function updateInfraCommissionRate(
        uint256 validatorId_,
        uint256 newInfraCommissionRate_
    ) external;

    function updateCommissionRate(
        uint256 validatorId,
        uint256 newCommissionRate
    ) external;

    function updateCommissionRate(
        uint256 validatorId,
        uint256 newCommissionRate,
        address bitkubNext
    ) external;

    function updateMinDelegated(
        uint256 validatorId,
        uint256 newMinDelegated
    ) external;

    function updateMinDelegated(
        uint256 validatorId,
        uint256 newMinDelegated,
        address bitkubNext
    ) external;

    function withdrawDelegatorsReward(
        uint256 validatorId
    ) external returns (uint256);

    function updateValidatorDelegation(
        uint256 validatorId,
        bool delegation
    ) external;

    function updateValidatorDelegation(
        uint256 validatorId,
        bool delegation,
        address bitkubNext
    ) external;

    function slash(address _signer) external returns (bool);

    function updateValidatorState(uint256 validatorId, int256 amount) external;

    function getMinimalValidators()
        external
        view
        returns (MinimalValidator[] memory);

    function getMinimalValidatorsByPage(
        uint256 page_,
        uint256 limit_
    ) external view returns (MinimalValidator[] memory);

    function getMinimalValidatorsLength() external view returns (uint256);

    function getValidatorId(address _signer) external view returns (uint256);

    function getMinimalValidatorIndex(
        address _signer
    ) external view returns (uint256);
    function getValidators() external view returns (address[] memory);
}

interface IStakeManagerStorage {
    function stakeManager() external view returns (address);

    function nftContract() external view returns (address);

    function validatorShareFactory() external view returns (address);

    function slashManager() external view returns (address);

    function soloLimit() external view returns (uint256);

    function soloAmount() external view returns (uint256);

    function poolLimit() external view returns (uint256);

    function poolAmount() external view returns (uint256);

    function officialLimit() external view returns (uint256);

    function officialAmount() external view returns (uint256);

    function defaultInfraCommissionRate() external view returns (uint256);

    function totalStaked() external view returns (uint256);

    function totalRewards() external view returns (uint256);

    function totalRewardsLiquidated() external view returns (uint256);

    function soloSlashRate() external view returns (uint256);

    function poolSlashAmount() external view returns (uint256);

    function unallocatedReward() external view returns (uint256);

    function officialPool() external view returns (address);

    function getNewOfficialPoolValidBlock() external view returns (uint256);

    function getOldNewOfficialSignerAndValidBlock()
        external
        view
        returns (address, address, uint256);

    function changeOfficialSigner(
        address newOfficialSigner_,
        string memory checksummedNewOfficialSigner_
    ) external;

    function setStakeManager(address stakeManager_) external;

    function setValidatorShareFactory(address validatorShareFactory_) external;

    function setSlashManager(address slashManager_) external;

    function setSoloLimit(uint256 val_) external;

    function incDescSoloLimit(uint256 val_, ChangeMode mode_) external;

    function setSoloAmount(uint256 val_) external;

    function incDescSoloAmount(uint256 val_, ChangeMode mode_) external;

    function setPoolLimit(uint256 val_) external;

    function incDescPoolLimit(uint256 val_, ChangeMode mode_) external;

    function setPoolAmount(uint256 val_) external;

    function incDescPoolAmount(uint256 val_, ChangeMode mode_) external;

    function setOfficialLimit(uint256 val_) external;

    function incDescOfficialLimit(uint256 val_, ChangeMode mode_) external;

    function setOfficialAmount(uint256 val_) external;

    function incDescOfficialAmount(uint256 val_, ChangeMode mode_) external;

    function setDefaultInfraCommissionRate(uint256 val_) external;

    function incDescDefaultInfraCommissionRate(
        uint256 val_,
        ChangeMode mode_
    ) external;

    function setTotalStaked(uint256 val_) external;
    function incDescTotalStaked(uint256 val_, ChangeMode mode_) external;

    function setTotalRewards(uint256 val_) external;

    function incDescTotalRewards(uint256 val_, ChangeMode mode_) external;

    function setTotalRewardsLiquidated(uint256 val_) external;

    function incDescTotalRewardsLiquidated(
        uint256 val_,
        ChangeMode mode_
    ) external;

    function setPoolSlashAmount(uint256 val_) external;

    function incDescPoolSlashAmount(uint256 val_, ChangeMode mode_) external;

    function setUnallocatedReward(uint256 val_) external;

    function incDescUnallocatedReward(uint256 val_, ChangeMode mode_) external;

    function addValidator(
        address key_,
        Validator memory val_
    ) external returns (uint256);

    function setValidatorIds(
        address key_,
        uint256[] memory validatorIds_
    ) external;

    function setValidatorInfo(address key_, Validator memory val_) external;

    function setValidatorInfoByIndex(
        uint256 index_,
        Validator memory val_
    ) external;

    function setValidatorInfoMinDeposit(address key_, uint256 val_) external;

    function setValidatorInfoMinDepositByIndex(
        uint256 index_,
        uint256 val_
    ) external;
    function incDescValidatorInfoMinDeposit(
        address key_,
        uint256 val_,
        ChangeMode mode_
    ) external;

    function incDescValidatorInfoMinDepositByIndex(
        uint256 index_,
        uint256 val_,
        ChangeMode mode_
    ) external;

    function setValidatorInfoAmount(address key_, uint256 val_) external;

    function setValidatorInfoAmountByIndex(
        uint256 index_,
        uint256 val_
    ) external;

    function incDescValidatorInfoAmount(
        address key_,
        uint256 val_,
        ChangeMode mode_
    ) external;

    function incDescValidatorInfoAmountByIndex(
        uint256 index_,
        uint256 val_,
        ChangeMode mode_
    ) external;

    function setValidatorInfoReward(address key_, uint256 val_) external;

    function setValidatorInfoRewardByIndex(
        uint256 index_,
        uint256 val_
    ) external;

    function incDescValidatorInfoReward(
        address key_,
        uint256 val_,
        ChangeMode mode_
    ) external;
    function incDescValidatorInfoRewardByIndex(
        uint256 index_,
        uint256 val_,
        ChangeMode mode_
    ) external;

    function setValidatorInfoSigner(address key_, address val_) external;

    function setValidatorInfoSignerByIndex(
        uint256 index_,
        address val_
    ) external;

    function setValidatorInfoValidatorShareContract(
        address key_,
        address val_
    ) external;

    function setValidatorInfoValidatorShareContractByIndex(
        uint256 index_,
        address val_
    ) external;

    function setValidatorInfoStatus(address key_, Status val_) external;

    function setValidatorInfoStatusByIndex(
        uint256 index_,
        Status val_
    ) external;

    function setValidatorInfoInfraCommissionRate(
        address key_,
        uint256 val_
    ) external;

    function setValidatorInfoInfraCommissionRateByIndex(
        uint256 index_,
        uint256 val_
    ) external;

    function incDescValidatorInfoInfraCommissionRate(
        address key_,
        uint256 val_,
        ChangeMode mode_
    ) external;
    function incDescValidatorInfoInfraCommissionRateByIndex(
        uint256 index_,
        uint256 val_,
        ChangeMode mode_
    ) external;

    function setValidatorInfoInfraCommissionAmount(
        address key_,
        uint256 val_
    ) external;

    function setValidatorInfoInfraCommissionAmountByIndex(
        uint256 index_,
        uint256 val_
    ) external;

    function incDescValidatorInfoInfraCommissionAmount(
        address key_,
        uint256 val_,
        ChangeMode mode_
    ) external;

    function incDescValidatorInfoInfraCommissionAmountByIndex(
        uint256 index_,
        uint256 val_,
        ChangeMode mode_
    ) external;

    function setValidatorInfoCommissionRate(
        address key_,
        uint256 val_
    ) external;

    function setValidatorInfoCommissionRateByIndex(
        uint256 index_,
        uint256 val_
    ) external;

    function incDescValidatorInfoCommissionRate(
        address key_,
        uint256 val_,
        ChangeMode mode_
    ) external;
    function incDescValidatorInfoCommissionRateByIndex(
        uint256 index_,
        uint256 val_,
        ChangeMode mode_
    ) external;

    function setValidatorInfoValidatorCommissionAmount(
        address key_,
        uint256 val_
    ) external;

    function setValidatorInfoValidatorCommissionAmountByIndex(
        uint256 index_,
        uint256 val_
    ) external;

    function incDescValidatorInfoValidatorCommissionAmount(
        address key_,
        uint256 val_,
        ChangeMode mode_
    ) external;

    function incDescValidatorInfoValidatorCommissionAmountByIndex(
        uint256 index_,
        uint256 val_,
        ChangeMode mode_
    ) external;

    function setValidatorInfoDelegatorCommissionAmount(
        address key_,
        uint256 val_
    ) external;

    function setValidatorInfoDelegatorCommissionAmountByIndex(
        uint256 index_,
        uint256 val_
    ) external;

    function incDescValidatorInfoDelegatorCommissionAmount(
        address key_,
        uint256 val_,
        ChangeMode mode_
    ) external;

    function incDescValidatorInfoDelegatorCommissionAmountByIndex(
        uint256 index_,
        uint256 val_,
        ChangeMode mode_
    ) external;

    function setValidatorInfoDelegatorsReward(
        address key_,
        uint256 val_
    ) external;

    function setValidatorInfoDelegatorsRewardByIndex(
        uint256 index_,
        uint256 val_
    ) external;

    function incDescValidatorInfoDelegatorsReward(
        address key_,
        uint256 val_,
        ChangeMode mode_
    ) external;

    function incDescValidatorInfoDelegatorsRewardByIndex(
        uint256 index_,
        uint256 val_,
        ChangeMode mode_
    ) external;

    function setValidatorInfoDelegatedAmount(
        address key_,
        uint256 val_
    ) external;

    function setValidatorInfoDelegatedAmountByIndex(
        uint256 index_,
        uint256 val_
    ) external;

    function incDescValidatorInfoDelegatedAmount(
        address key_,
        uint256 val_,
        ChangeMode mode_
    ) external;

    function incDescValidatorInfoDelegatedAmountByIndex(
        uint256 index_,
        uint256 val_,
        ChangeMode mode_
    ) external;

    function getValidatorInfo(
        address key_
    ) external view returns (Validator memory);

    function getValidatorInfoByIndex(
        uint256 index_
    ) external view returns (Validator memory);

    function getValidatorInfoMinDeposit(
        address key_
    ) external view returns (uint256);

    function getValidatorInfoMinDepositByIndex(
        uint256 index_
    ) external view returns (uint256);

    function getValidatorInfoAmount(
        address key_
    ) external view returns (uint256);

    function getValidatorInfoAmountByIndex(
        uint256 index_
    ) external view returns (uint256);

    function getValidatorInfoReward(
        address key_
    ) external view returns (uint256);

    function getValidatorInfoRewardByIndex(
        uint256 index_
    ) external view returns (uint256);

    function getValidatorInfoSigner(
        address key_
    ) external view returns (address);

    function getValidatorInfoSignerByIndex(
        uint256 index_
    ) external view returns (address);

    function getValidatorInfoValidatorShareContract(
        address key_
    ) external view returns (address);

    function getValidatorInfoValidatorShareContractByIndex(
        uint256 index_
    ) external view returns (address);

    function getValidatorInfoStatus(
        address key_
    ) external view returns (Status);

    function getValidatorInfoStatusByIndex(
        uint256 index_
    ) external view returns (Status);

    function getValidatorInfoInfraCommissionRate(
        address key_
    ) external view returns (uint256);

    function getValidatorInfoInfraCommissionRateByIndex(
        uint256 index_
    ) external view returns (uint256);

    function getValidatorInfoInfraCommissionAmount(
        address key_
    ) external view returns (uint256);

    function getValidatorInfoInfraCommissionAmountByIndex(
        uint256 index_
    ) external view returns (uint256);

    function getValidatorInfoCommissionRate(
        address key_
    ) external view returns (uint256);

    function getValidatorInfoCommissionRateByIndex(
        uint256 index_
    ) external view returns (uint256);

    function getValidatorInfoValidatorCommissionAmount(
        address key_
    ) external view returns (uint256);

    function getValidatorInfoValidatorCommissionAmountByIndex(
        uint256 index_
    ) external view returns (uint256);

    function getValidatorInfoDelegatorCommissionAmount(
        address key_
    ) external view returns (uint256);

    function getValidatorInfoDelegatorCommissionAmountByIndex(
        uint256 index_
    ) external view returns (uint256);

    function getValidatorInfoDelegatorsReward(
        address key_
    ) external view returns (uint256);

    function getValidatorInfoDelegatorsRewardByIndex(
        uint256 index_
    ) external view returns (uint256);

    function getValidatorInfoDelegatedAmount(
        address key_
    ) external view returns (uint256);

    function getValidatorInfoDelegatedAmountByIndex(
        uint256 index_
    ) external view returns (uint256);

    function getValidatorListLength() external view returns (uint256);

    function isInValidatorList(address val_) external view returns (bool);

    function getValidatorCurrentIndex(
        address val_,
        bool revertIfNotFound_
    ) external view returns (uint256, bool);

    function getValidatorByIndex(
        uint256 index_
    ) external view returns (address);

    function getAllValidator() external view returns (address[] memory);

    function getValidatorByPage(
        uint256 page,
        uint256 limit
    ) external view returns (address[] memory);
    function getValidatorIndexLength(
        address val_
    ) external view returns (uint256);

    function getValidatorIndexByIndex(
        address val_,
        uint256 index_
    ) external view returns (uint256);

    function getMinimalValidatorListLength() external view returns (uint256);

    function isInMinimalValidatorList(
        address val_
    ) external view returns (bool);

    function isInMinimalValidatorListByValidatorId(
        uint256 validatorId_
    ) external view returns (bool);

    function getMinimalValidatorIndex(
        address val_,
        bool revertIfNotFound_
    ) external view returns (uint256, bool);

    function getMinimalValidatorByIndex(
        uint256 index_
    ) external view returns (address);

    function getAllMinimalValidators() external view returns (address[] memory);

    function getMinimalValidatorsByPage(
        uint256 page,
        uint256 limit
    ) external view returns (address[] memory);

    function addMinimalValidator(address val_) external;

    function removeMinimalValidator(address val_) external;

    function removeMinimalValidatorByIndex(uint256 index_) external;

    function getMinimalValidatorsWithValidatorPowerByPage(
        uint256 page_,
        uint256 limit_
    ) external view returns (address[] memory, uint256[] memory);
}

contract SpanBase {
    uint256 public immutable initialSpanBlock;
    uint256 public stableSpanBlock;

    uint256 public constant SPAN = 50;

    constructor(uint256 initialSpanBlock_) {
        initialSpanBlock = initialSpanBlock_;

        (, uint256 initialSpanEnd) = getSpanRange(0);
        stableSpanBlock = initialSpanEnd + 1;
    }

    /// @notice Calculate span number from block number
    /// @param number_ block number
    /// @return span number
    function getSpanByBlock(uint256 number_) public view returns (uint256) {
        if (number_ < stableSpanBlock) {
            return 0;
        }
        return (number_ - stableSpanBlock) / SPAN + 1;
    }

    /// @notice Calculate current span number
    /// @return span number
    function currentSpanNumber() public view returns (uint256) {
        return getSpanByBlock(block.number);
    }

    /// @notice Get span range
    /// @param number_ span number
    function getSpanRange(
        uint256 number_
    ) public view returns (uint256 startBlock, uint256 endBlock) {
        if (number_ == 0) {
            // Transition - Calculates a span or a peroid of
            // the consensus transition
            // Remaining size of a first single span
            uint256 size = SPAN - (initialSpanBlock % SPAN);

            startBlock = initialSpanBlock;
            // Find the edge of the first span (ends with 49, 99; SPAN=50)
            endBlock = size - 1 + initialSpanBlock;

            // If the first span does not contains a commitSpan block
            // which is always at the block number 26,76. Expand one
            // more span to the transition.
            if (size < SPAN / 2 - 1) {
                endBlock += SPAN;
            }
        } else {
            // Stable -  Just a general span calculation
            startBlock = stableSpanBlock + ((number_ - 1) * SPAN);
            endBlock = startBlock + SPAN - 1;
        }

        return (startBlock, endBlock);
    }

    /// @notice Get span's commitment block
    /// @param number_ span number
    /// @return blockNumber (commitment block)
    function getCommitmentBlock(uint256 number_) public view returns (uint256) {
        if (number_ == 0) {
            // case: transition period
            return calculateTransitionSpanCommitmentBlock();
        }
        // stableSpanBlock + number of blocks passed + half span
        return stableSpanBlock + ((number_ - 1) * SPAN) + (SPAN / 2 + 1);
    }

    /// @notice Get span's commitment block in transition period
    /// @return blockNumber (commitment block)
    function calculateTransitionSpanCommitmentBlock()
        public
        view
        returns (uint256)
    {
        if (initialSpanBlock % SPAN > SPAN / 2 + 1) {
            return
                initialSpanBlock +
                (SPAN - (initialSpanBlock % SPAN)) +
                (SPAN / 2 + 1);
        } else {
            return stableSpanBlock - (SPAN / 2) + 1;
        }
    }
}

struct SlashEpoch {
    uint256 lastEpochStart;
    uint256 lastSlash;
    uint256 counter;
}

contract SlashManager is SpanBase {
    IStakeManagerStorage private immutable _stakeManagerStorage;

    /// @notice Max threshold before being slashed (43=3hr)
    uint256 public immutable threshold;
    /// @notice Size of an epoch to be calculate when slash (345=24hr)
    uint256 public immutable maxEpochSize;

    mapping(address => SlashEpoch) public epoch;
    mapping(address => mapping(uint256 => bool)) public span;

    /// @notice Emit when SlashManager calls StakeManager.slash
    /// @param signer a signer who was slashed
    /// @param span span number that signer was slashed
    /// @param counter an epoch counter
    event Slash(
        address indexed signer,
        uint256 indexed span,
        uint256 indexed counter
    );

    /// @notice Just a warning, won't be penalise until they reach the threshold
    /// @param signer a signer who was slashed
    /// @param span span number that signer was slashed
    /// @param counter an epoch counter
    event Warn(
        address indexed signer,
        uint256 indexed span,
        uint256 indexed counter
    );

    /// @notice Trigger new epoch
    /// @param signer a signer who was slashed
    /// @param span span number that signer was slashed
    event StartEpoch(address indexed signer, uint256 indexed span);
    constructor(
        address stakeManagerStorage_,
        uint256 initialSpanBlock_,
        uint256 threshold_,
        uint256 maxEpochSize_
    ) SpanBase(initialSpanBlock_) {
        _stakeManagerStorage = IStakeManagerStorage(stakeManagerStorage_);
        threshold = threshold_;
        maxEpochSize = maxEpochSize_;
    }

    /// @notice Slashing the given signer
    /// @dev Explain to a developer any extra details
    /// @param signer_ a signer address
    /// @param currentSpan_ current span number
    function slash(
        address signer_,
        uint256 currentSpan_
    ) public returns (bool) {
        uint256 currentSpan = getSpanByBlock(block.number);

        require(currentSpan_ == currentSpan, "SlashManager: span mismatch");

        require(
            currentSpan_ != 0,
            "SlashManager: not allowed on transition span"
        );
        (
            address oldOfficialSigner,
            address newOfficialSigner,
            uint256 validBlock
        ) = _stakeManagerStorage.getOldNewOfficialSignerAndValidBlock();

        bool allowed = (msg.sender == oldOfficialSigner &&
            block.number < validBlock) ||
            (msg.sender == newOfficialSigner && block.number >= validBlock);
        require(allowed, "SlashManager: only official pool");

        require(
            block.difficulty == 1,
            "SlashManager: only allowed in no-turn block"
        );
        require(tx.gasprice == 0, "SlashManager: only system transaction");

        require(
            !isSignerSlashed(signer_, currentSpan),
            "SlashManager: already call slash in this span"
        );

        _markSpan(signer_, currentSpan);

        return _triggerEpoch(signer_, currentSpan);
    }

    /// @notice Check if signer was slased in the given span or not
    /// @dev This function is currently being used by geth
    /// @param signer_ a signer address that want to check
    /// @param span_ span number to check
    /// @return Boolean if signer was slased in the given span or not
    function isSignerSlashed(
        address signer_,
        uint256 span_
    ) public view returns (bool) {
        return span[signer_][span_];
    }

    /// @notice Call StakeManager.slash
    /// @param signer_ signer address
    /// @param span_ current span number
    /// @return Slash result as a boolean
    function _slash(address signer_, uint256 span_) private returns (bool) {
        bool slashResult = IStakeManager(_stakeManagerStorage.stakeManager())
            .slash(signer_);
        epoch[signer_].lastSlash = span_;
        emit Slash(signer_, span_, epoch[signer_].counter);
        return slashResult;
    }

    /// @notice Emit warning event
    /// @param signer_ signer address
    /// @param span_ current span number
    /// @return Slash result always false
    function _warn(address signer_, uint256 span_) private returns (bool) {
        emit Warn(signer_, span_, epoch[signer_].counter);
        return false;
    }

    /// @notice Mark span as being slashed or warned.
    /// @param signer_ a signer address
    /// @param span_ current span number
    function _markSpan(address signer_, uint256 span_) private {
        span[signer_][span_] = true;
    }

    /// @notice Start new individual epoch
    /// @dev This function won't do any counter increment. Please do it externally.
    /// @param signer_ a signer address
    /// @param span_ an epoch start a span number
    function _startNewEpoch(address signer_, uint256 span_) private {
        epoch[signer_].lastEpochStart = span_;
        epoch[signer_].counter = 0;
        emit StartEpoch(signer_, span_);
    }

    /// @notice Get epoch storage's pointer
    /// @param signer_ a signer's address
    /// @return Storage pointer
    function _epochPtr(
        address signer_
    ) private view returns (SlashEpoch storage) {
        return epoch[signer_];
    }

    /// @notice Handling slash mechanism here
    /// @dev Should do an authorization externally before calling this internal
function
    /// @param signer_ a signer's address
    /// @param currentInteractionSpan_ a current span
    /// @return A result of slashing mechanism in boolean
    function _triggerEpoch(
        address signer_,
        uint256 currentInteractionSpan_
    ) private returns (bool) {
        bool result;

        SlashEpoch storage ptr = _epochPtr(signer_);

        if (ptr.lastEpochStart > ptr.lastSlash) {
            // start new epoch or continue with the existing one
            if (
                currentInteractionSpan_ - ptr.lastEpochStart + 1 > maxEpochSize
            ) {
                _startNewEpoch(signer_, currentInteractionSpan_);
            }
        } else {
            // always start new epoch
            _startNewEpoch(signer_, currentInteractionSpan_);
        }

        ptr.counter++;

        if (ptr.counter == threshold) {
            result = _slash(signer_, currentInteractionSpan_);
        } else {
            result = _warn(signer_, currentInteractionSpan_);
        }

        return result;
    }
}
```

5. ValidatorShareFactory and ValidatorShare

```
// * Copyright 2023 Bitkub Blockchain Technology Co., Ltd. - All Rights Reserved.
// * This code is proprietary.
// * Unauthorized copying, modification or distribution of this code, via any
medium, is strictly prohibited without express permission.

pragma solidity ^0.8.0;

enum AddressType {
    Adder,
    Allowed
}

interface ITransferRouter {
    function isAdderOrAllowedAddr(
        address addr_,
        AddressType type_
    ) external view returns (bool);

    function adderOrAllowedAddrLength(
        AddressType type_
    ) external view returns (uint256);

    function adderOrAllowedAddrByIndex(
        uint256 index_,
        AddressType type_
    ) external view returns (address);
    function adderOrAllowedAddrByPage(
        uint256 page_,
        uint256 limit_,
        AddressType type_
    ) external view returns (address[] memory);

    function addAdderAddress(address addr_) external;

    function revokeAdderAddress(address addr_) external;

    function addAllowedAddress(address addr_) external;

    function revokeAllowedAddress(address addr_) external;

    function transferFromKKUB(
        address from_,
        address to_,
        uint256 amount_
    ) external;
}

enum LiquidateMode {
    All,
    ValidatorReward,
    ValidatorCommission,
    DelegatorCommission
}

enum TransferTokenMode {
    None,
    Staked,
    Reward
}

enum ChangeMode {
    Increase,
    Decrease
}

enum Status {
    Uninitialized,
    Active,
    Unstaked
}
```

```
enum SlashErrorCode {
    Uninitialized,
    StakeAmountNotEnough,
    OfficialPool,
    NotPool,
    InactivatedPool
}

struct Validator {
    uint128 amount;
    uint128 delegatedAmount;
    uint128 reward;
    uint128 delegatorsReward;
    uint128 infraCommissionAmount;
    uint128 validatorCommissionAmount;
    uint128 delegatorCommissionAmount;
    uint128 minDeposit;
    address signer;
    address validatorShareContract;
    Status status;
    uint16 infraCommissionRate;
    uint16 commissionRate;
}

struct MinimalValidator {
    address signer;
    uint256 power;
}

interface IStakeManagerStorage {
    function stakeManager() external view returns (address);

    function nftContract() external view returns (address);

    function validatorShareFactory() external view returns (address);

    function slashManager() external view returns (address);

    function soloLimit() external view returns (uint256);

    function soloAmount() external view returns (uint256);

    function poolLimit() external view returns (uint256);
    function poolAmount() external view returns (uint256);

    function officialLimit() external view returns (uint256);

    function officialAmount() external view returns (uint256);

    function defaultInfraCommissionRate() external view returns (uint256);

    function totalStaked() external view returns (uint256);

    function totalRewards() external view returns (uint256);

    function totalRewardsLiquidated() external view returns (uint256);

    function soloSlashRate() external view returns (uint256);

    function poolSlashAmount() external view returns (uint256);

    function unallocatedReward() external view returns (uint256);

    function officialPool() external view returns (address);

    function getNewOfficialPoolValidBlock() external view returns (uint256);

    function getOldNewOfficialSignerAndValidBlock()
        external
        view
        returns (address, address, uint256);

    function changeOfficialSigner(
        address newOfficialSigner_,
        string memory checksummedNewOfficialSigner_
    ) external;

    function setStakeManager(address stakeManager_) external;

    function setValidatorShareFactory(address validatorShareFactory_) external;

    function setSlashManager(address slashManager_) external;

    function setSoloLimit(uint256 val_) external;

    function incDescSoloLimit(uint256 val_, ChangeMode mode_) external;

    function setSoloAmount(uint256 val_) external;
    function incDescSoloAmount(uint256 val_, ChangeMode mode_) external;

    function setPoolLimit(uint256 val_) external;

    function incDescPoolLimit(uint256 val_, ChangeMode mode_) external;

    function setPoolAmount(uint256 val_) external;

    function incDescPoolAmount(uint256 val_, ChangeMode mode_) external;

    function setOfficialLimit(uint256 val_) external;

    function incDescOfficialLimit(uint256 val_, ChangeMode mode_) external;

    function setOfficialAmount(uint256 val_) external;

    function incDescOfficialAmount(uint256 val_, ChangeMode mode_) external;

    function setDefaultInfraCommissionRate(uint256 val_) external;

    function incDescDefaultInfraCommissionRate(
        uint256 val_,
        ChangeMode mode_
    ) external;

    function setTotalStaked(uint256 val_) external;

    function incDescTotalStaked(uint256 val_, ChangeMode mode_) external;

    function setTotalRewards(uint256 val_) external;

    function incDescTotalRewards(uint256 val_, ChangeMode mode_) external;

    function setTotalRewardsLiquidated(uint256 val_) external;

    function incDescTotalRewardsLiquidated(
        uint256 val_,
        ChangeMode mode_
    ) external;

    function setPoolSlashAmount(uint256 val_) external;

    function incDescPoolSlashAmount(uint256 val_, ChangeMode mode_) external;
    function setUnallocatedReward(uint256 val_) external;

    function incDescUnallocatedReward(uint256 val_, ChangeMode mode_) external;

    function addValidator(
        address key_,
        Validator memory val_
    ) external returns (uint256);

    function setValidatorIds(
        address key_,
        uint256[] memory validatorIds_
    ) external;

    function setValidatorInfo(address key_, Validator memory val_) external;

    function setValidatorInfoByIndex(
        uint256 index_,
        Validator memory val_
    ) external;

    function setValidatorInfoMinDeposit(address key_, uint256 val_) external;

    function setValidatorInfoMinDepositByIndex(
        uint256 index_,
        uint256 val_
    ) external;

    function incDescValidatorInfoMinDeposit(
        address key_,
        uint256 val_,
        ChangeMode mode_
    ) external;

    function incDescValidatorInfoMinDepositByIndex(
        uint256 index_,
        uint256 val_,
        ChangeMode mode_
    ) external;

    function setValidatorInfoAmount(address key_, uint256 val_) external;

    function setValidatorInfoAmountByIndex(
        uint256 index_,
        uint256 val_
    ) external;

    function incDescValidatorInfoAmount(
        address key_,
        uint256 val_,
        ChangeMode mode_
    ) external;

    function incDescValidatorInfoAmountByIndex(
        uint256 index_,
        uint256 val_,
        ChangeMode mode_
    ) external;

    function setValidatorInfoReward(address key_, uint256 val_) external;

    function setValidatorInfoRewardByIndex(
        uint256 index_,
        uint256 val_
    ) external;

    function incDescValidatorInfoReward(
        address key_,
        uint256 val_,
        ChangeMode mode_
    ) external;

    function incDescValidatorInfoRewardByIndex(
        uint256 index_,
        uint256 val_,
        ChangeMode mode_
    ) external;

    function setValidatorInfoSigner(address key_, address val_) external;

    function setValidatorInfoSignerByIndex(
        uint256 index_,
        address val_
    ) external;

    function setValidatorInfoValidatorShareContract(
        address key_,
        address val_
    ) external;
    function setValidatorInfoValidatorShareContractByIndex(
        uint256 index_,
        address val_
    ) external;

    function setValidatorInfoStatus(address key_, Status val_) external;

    function setValidatorInfoStatusByIndex(
        uint256 index_,
        Status val_
    ) external;

    function setValidatorInfoInfraCommissionRate(
        address key_,
        uint256 val_
    ) external;

    function setValidatorInfoInfraCommissionRateByIndex(
        uint256 index_,
        uint256 val_
    ) external;

    function incDescValidatorInfoInfraCommissionRate(
        address key_,
        uint256 val_,
        ChangeMode mode_
    ) external;

    function incDescValidatorInfoInfraCommissionRateByIndex(
        uint256 index_,
        uint256 val_,
        ChangeMode mode_
    ) external;

    function setValidatorInfoInfraCommissionAmount(
        address key_,
        uint256 val_
    ) external;

    function setValidatorInfoInfraCommissionAmountByIndex(
        uint256 index_,
        uint256 val_
    ) external;

    function incDescValidatorInfoInfraCommissionAmount(
        address key_,
        uint256 val_,
        ChangeMode mode_
    ) external;

    function incDescValidatorInfoInfraCommissionAmountByIndex(
        uint256 index_,
        uint256 val_,
        ChangeMode mode_
    ) external;

    function setValidatorInfoCommissionRate(
        address key_,
        uint256 val_
    ) external;

    function setValidatorInfoCommissionRateByIndex(
        uint256 index_,
        uint256 val_
    ) external;

    function incDescValidatorInfoCommissionRate(
        address key_,
        uint256 val_,
        ChangeMode mode_
    ) external;

    function incDescValidatorInfoCommissionRateByIndex(
        uint256 index_,
        uint256 val_,
        ChangeMode mode_
    ) external;

    function setValidatorInfoValidatorCommissionAmount(
        address key_,
        uint256 val_
    ) external;

    function setValidatorInfoValidatorCommissionAmountByIndex(
        uint256 index_,
        uint256 val_
    ) external;

    function incDescValidatorInfoValidatorCommissionAmount(
        address key_,
        uint256 val_,
        ChangeMode mode_
    ) external;

    function incDescValidatorInfoValidatorCommissionAmountByIndex(
        uint256 index_,
        uint256 val_,
        ChangeMode mode_
    ) external;

    function setValidatorInfoDelegatorCommissionAmount(
        address key_,
        uint256 val_
    ) external;

    function setValidatorInfoDelegatorCommissionAmountByIndex(
        uint256 index_,
        uint256 val_
    ) external;

    function incDescValidatorInfoDelegatorCommissionAmount(
        address key_,
        uint256 val_,
        ChangeMode mode_
    ) external;

    function incDescValidatorInfoDelegatorCommissionAmountByIndex(
        uint256 index_,
        uint256 val_,
        ChangeMode mode_
    ) external;

    function setValidatorInfoDelegatorsReward(
        address key_,
        uint256 val_
    ) external;

    function setValidatorInfoDelegatorsRewardByIndex(
        uint256 index_,
        uint256 val_
    ) external;

    function incDescValidatorInfoDelegatorsReward(
        address key_,
        uint256 val_,
        ChangeMode mode_
    ) external;

    function incDescValidatorInfoDelegatorsRewardByIndex(
        uint256 index_,
        uint256 val_,
        ChangeMode mode_
    ) external;

    function setValidatorInfoDelegatedAmount(
        address key_,
        uint256 val_
    ) external;

    function setValidatorInfoDelegatedAmountByIndex(
        uint256 index_,
        uint256 val_
    ) external;

    function incDescValidatorInfoDelegatedAmount(
        address key_,
        uint256 val_,
        ChangeMode mode_
    ) external;

    function incDescValidatorInfoDelegatedAmountByIndex(
        uint256 index_,
        uint256 val_,
        ChangeMode mode_
    ) external;

    function getValidatorInfo(
        address key_
    ) external view returns (Validator memory);

    function getValidatorInfoByIndex(
        uint256 index_
    ) external view returns (Validator memory);

    function getValidatorInfoMinDeposit(
        address key_
    ) external view returns (uint256);

    function getValidatorInfoMinDepositByIndex(
        uint256 index_
    ) external view returns (uint256);

    function getValidatorInfoAmount(
        address key_
    ) external view returns (uint256);

    function getValidatorInfoAmountByIndex(
        uint256 index_
    ) external view returns (uint256);

    function getValidatorInfoReward(
        address key_
    ) external view returns (uint256);

    function getValidatorInfoRewardByIndex(
        uint256 index_
    ) external view returns (uint256);

    function getValidatorInfoSigner(
        address key_
    ) external view returns (address);

    function getValidatorInfoSignerByIndex(
        uint256 index_
    ) external view returns (address);

    function getValidatorInfoValidatorShareContract(
        address key_
    ) external view returns (address);

    function getValidatorInfoValidatorShareContractByIndex(
        uint256 index_
    ) external view returns (address);

    function getValidatorInfoStatus(
        address key_
    ) external view returns (Status);

    function getValidatorInfoStatusByIndex(
        uint256 index_
    ) external view returns (Status);

    function getValidatorInfoInfraCommissionRate(
        address key_
    ) external view returns (uint256);
    function getValidatorInfoInfraCommissionRateByIndex(
        uint256 index_
    ) external view returns (uint256);

    function getValidatorInfoInfraCommissionAmount(
        address key_
    ) external view returns (uint256);

    function getValidatorInfoInfraCommissionAmountByIndex(
        uint256 index_
    ) external view returns (uint256);

    function getValidatorInfoCommissionRate(
        address key_
    ) external view returns (uint256);

    function getValidatorInfoCommissionRateByIndex(
        uint256 index_
    ) external view returns (uint256);

    function getValidatorInfoValidatorCommissionAmount(
        address key_
    ) external view returns (uint256);

    function getValidatorInfoValidatorCommissionAmountByIndex(
        uint256 index_
    ) external view returns (uint256);

    function getValidatorInfoDelegatorCommissionAmount(
        address key_
    ) external view returns (uint256);

    function getValidatorInfoDelegatorCommissionAmountByIndex(
        uint256 index_
    ) external view returns (uint256);

    function getValidatorInfoDelegatorsReward(
        address key_
    ) external view returns (uint256);

    function getValidatorInfoDelegatorsRewardByIndex(
        uint256 index_
    ) external view returns (uint256);
    function getValidatorInfoDelegatedAmount(
        address key_
    ) external view returns (uint256);

    function getValidatorInfoDelegatedAmountByIndex(
        uint256 index_
    ) external view returns (uint256);

    function getValidatorListLength() external view returns (uint256);

    function isInValidatorList(address val_) external view returns (bool);

    function getValidatorCurrentIndex(
        address val_,
        bool revertIfNotFound_
    ) external view returns (uint256, bool);

    function getValidatorByIndex(
        uint256 index_
    ) external view returns (address);

    function getAllValidator() external view returns (address[] memory);

    function getValidatorByPage(
        uint256 page,
        uint256 limit
    ) external view returns (address[] memory);

    function getValidatorIndexLength(
        address val_
    ) external view returns (uint256);

    function getValidatorIndexByIndex(
        address val_,
        uint256 index_
    ) external view returns (uint256);

    function getMinimalValidatorListLength() external view returns (uint256);

    function isInMinimalValidatorList(
        address val_
    ) external view returns (bool);

    function isInMinimalValidatorListByValidatorId(
        uint256 validatorId_
    ) external view returns (bool);

    function getMinimalValidatorIndex(
        address val_,
        bool revertIfNotFound_
    ) external view returns (uint256, bool);

    function getMinimalValidatorByIndex(
        uint256 index_
    ) external view returns (address);

    function getAllMinimalValidators() external view returns (address[] memory);

    function getMinimalValidatorsByPage(
        uint256 page,
        uint256 limit
    ) external view returns (address[] memory);

    function addMinimalValidator(address val_) external;

    function removeMinimalValidator(address val_) external;

    function removeMinimalValidatorByIndex(uint256 index_) external;

    function getMinimalValidatorsWithValidatorPowerByPage(
        uint256 page_,
        uint256 limit_
    ) external view returns (address[] memory, uint256[] memory);
}

// * Copyright 2023 Bitkub Blockchain Technology Co., Ltd. - All Rights Reserved.
// * This code is proprietary.
// * Unauthorized copying, modification or distribution of this code, via any
medium, is strictly prohibited without express permission.

abstract contract Initializable {
    bool private _inited = false;

    modifier initializer() {
        require(!_inited, "already inited");
        _inited = true;

        _;
    }
}
```

```
abstract contract ReentrancyGuard {
    uint256 private constant _NOT_ENTERED = 1;
    uint256 private constant _ENTERED = 2;

    uint256 private _status;

    function _nonReentrantBefore() private {
        // On the first call to nonReentrant, _notEntered will be true
        require(_status != _ENTERED, "ReentrancyGuard: reentrant call");

        // Any calls to nonReentrant after this point will fail
        _status = _ENTERED;
    }

    function _nonReentrantAfter() private {
        // By storing the original value once again, a refund is triggered (see
        // https://eips.ethereum.org/EIPS/eip-2200)
        _status = _NOT_ENTERED;
    }

    modifier nonReentrant() {
        _nonReentrantBefore();
        _;
        _nonReentrantAfter();
    }

    constructor() {
        _status = _NOT_ENTERED;
    }
}

interface IKAP20 {
    function totalSupply() external view returns (uint256);

    function decimals() external view returns (uint8);

    function symbol() external view returns (string memory);

    function name() external view returns (string memory);

    function balanceOf(address account) external view returns (uint256);

    function transfer(
        address recipient,
        uint256 amount
    ) external returns (bool);

    function allowance(
        address _owner,
        address spender
    ) external view returns (uint256);

    function approve(address spender, uint256 amount) external returns (bool);

    function transferFrom(
        address sender,
        address recipient,
        uint256 amount
    ) external returns (bool);

    function adminTransfer(
        address sender,
        address recipient,
        uint256 amount
    ) external returns (bool success);

    event Transfer(address indexed from, address indexed to, uint256 value);

    event AdminTransfer(
        address indexed from,
        address indexed to,
        uint256 value
    );

    event Approval(
        address indexed owner,
        address indexed spender,
        uint256 value
    );
}

interface IKKUB is IKAP20 {
    function deposit() external payable;

    function withdraw(uint256 _value) external;

    function approve(address spender, uint256 amount) external returns (bool);
}
```

```
// note this contract interface is only for stakeManager use
interface IValidatorShare {
    function delegate() external payable returns (uint256 amountToDeposit);

    function undelegate(uint256 claimAmount) external returns (uint256);

    function claimRewards() external returns (uint256);

    function updateDelegation(bool _delegation) external;

    function updateMinDelegated(uint256 _minDelegated) external;

    function getLiquidRewards(address user) external view returns (uint256);
}

interface IStakeManager {
    function setOfficialPoolStaker(address officialPoolStaker_) external;

    function transferUnallocatedReward(
        address payable to,
        uint256 amount
    ) external returns (bool);

    function transferInfraCommission(
        uint256[] memory validatorIds_,
        uint256[] memory amounts_,
        address payable to_
    ) external;

    function transferFunds(
        uint256 validatorId,
        uint256 amount,
        address delegator,
        TransferTokenMode mode
    ) external returns (bool);

    function delegationDeposit(
        uint256 validatorId
    ) external payable returns (bool);

    function stake(address signer, bool delegation) external payable;

    function stake(
        address signer,
        bool delegation,
        uint256 amount,
        address bitkubNext
    ) external;

    function unstake(uint256 validatorId) external;

    function unstake(uint256 validatorId, address bitkubNext) external;

    function restake(uint256 validatorId) external payable;

    function restake(
        uint256 validatorId,
        uint256 amount,
        address bitkubNext
    ) external;

    function distributeReward() external payable;

    function claimRewards(uint256 validatorId) external;

    function claimRewards(uint256 validatorId, address bitkubNext) external;

    function claimCommissionRewards(uint256 validatorId) external;

    function claimCommissionRewards(
        uint256 validatorId,
        address bitkubNext
    ) external;

    function updateInfraCommissionRate(
        uint256 validatorId_,
        uint256 newInfraCommissionRate_
    ) external;

    function updateCommissionRate(
        uint256 validatorId,
        uint256 newCommissionRate
    ) external;

    function updateCommissionRate(
        uint256 validatorId,
        uint256 newCommissionRate,
        address bitkubNext
    ) external;
    function updateMinDelegated(
        uint256 validatorId,
        uint256 newMinDelegated
    ) external;

    function updateMinDelegated(
        uint256 validatorId,
        uint256 newMinDelegated,
        address bitkubNext
    ) external;

    function withdrawDelegatorsReward(
        uint256 validatorId
    ) external returns (uint256);

    function updateValidatorDelegation(
        uint256 validatorId,
        bool delegation
    ) external;

    function updateValidatorDelegation(
        uint256 validatorId,
        bool delegation,
        address bitkubNext
    ) external;

    function slash(address _signer) external returns (bool);

    function updateValidatorState(uint256 validatorId, int256 amount) external;

    function getMinimalValidators()
        external
        view
        returns (MinimalValidator[] memory);

    function getMinimalValidatorsByPage(
        uint256 page_,
        uint256 limit_
    ) external view returns (MinimalValidator[] memory);

    function getMinimalValidatorsLength() external view returns (uint256);

    function getValidatorId(address _signer) external view returns (uint256);

    function getMinimalValidatorIndex(
        address _signer
    ) external view returns (uint256);

    function getValidators() external view returns (address[] memory);
}

interface IKYCBitkubChain {
    function setKycCompleted(address _addr, uint256 _level) external;

    function kycsLevel(address _addr) external view returns (uint256);
}

abstract contract KYCHandler {
    event OnlyKYCAddressActivated(address indexed caller);
    event KYCSet(
        address indexed oldKYC,
        address indexed newKYC,
        address indexed caller
    );
    event AcceptedKYCLevelSet(
        uint256 indexed oldAcceptedKYCLevelSet,
        uint256 indexed newAcceptedKYCLevelSet,
        address indexed caller
    );

    IKYCBitkubChain public kyc;

    uint256 public acceptedKycLevel;
    bool public isActivatedOnlyKycAddress;

    modifier onlyBitkubNext(address bitkubNext_) {
        _onlyBitkubNext(bitkubNext_);
        _;
    }

    constructor(address kyc_, uint256 acceptedKycLevel_) {
        _setKyc(IKYCBitkubChain(kyc_));
        _setAcceptedKycLevel(acceptedKycLevel_);
    }

    function _activateOnlyKycAddress() internal virtual {
        isActivatedOnlyKycAddress = true;
        emit OnlyKYCAddressActivated(msg.sender);
    }
    function _setKyc(IKYCBitkubChain _kyc) internal virtual {
        emit KYCSet(address(kyc), address(_kyc), msg.sender);
        kyc = _kyc;
    }

    function _setAcceptedKycLevel(uint256 _kycLevel) internal virtual {
        emit AcceptedKYCLevelSet(acceptedKycLevel, _kycLevel, msg.sender);
        acceptedKycLevel = _kycLevel;
    }

    function _onlyBitkubNext(address bitkubNext_) internal view {
        require(
            _isBitkubNext(bitkubNext_),
            "KYCHandler: only Bitkub NEXT user"
        );
    }

    function _isBitkubNext(address bitkubNext_) internal view returns (bool) {
        return kyc.kycsLevel(bitkubNext_) >= acceptedKycLevel;
    }
}

abstract contract Committee {
    address public committee;

    event CommitteeSet(
        address indexed oldCommittee,
        address indexed newCommittee,
        address indexed caller
    );

    function _onlyCommittee() internal view {
        require(
            msg.sender == committee,
            "Committee: restricted only committee"
        );
    }

    modifier onlyCommittee() {
        _onlyCommittee();
        _;
    }

    constructor(address committee_) {
        committee = committee_;
    }

    function setCommittee(address _committee) public virtual onlyCommittee {
        emit CommitteeSet(committee, _committee, msg.sender);
        committee = _committee;
    }
}

abstract contract AccessController is Committee, KYCHandler {
    address public transferRouter;

    event TransferRouterSet(
        address indexed oldTransferRouter,
        address indexed newTransferRouter,
        address indexed caller
    );

    modifier onlyTransferRouter() {
        require(
            msg.sender == transferRouter,
            "AccessController: restricted only transfer router"
        );
        _;
    }

    constructor(
        address committee_,
        address kyc_,
        uint256 acceptedKycLevel_
    ) Committee(committee_) KYCHandler(kyc_, acceptedKycLevel_) {}

    function activateOnlyKycAddress() external onlyCommittee {
        _activateOnlyKycAddress();
    }

    function setKyc(IKYCBitkubChain _kyc) external onlyCommittee {
        _setKyc(_kyc);
    }

    function setAcceptedKycLevel(uint256 _kycLevel) external onlyCommittee {
        _setAcceptedKycLevel(_kycLevel);
    }

    function setTransferRouter(address _transferRouter) external onlyCommittee {
        emit TransferRouterSet(transferRouter, _transferRouter, msg.sender);
        transferRouter = _transferRouter;
    }
}

abstract contract Context {
    function _msgSender() internal view virtual returns (address) {
        return msg.sender;
    }

    function _msgData() internal view virtual returns (bytes calldata) {
        return msg.data;
    }
}

abstract contract Pausable is Context {
    bool private _paused;

    event Paused(address account);

    event Unpaused(address account);

    modifier whenNotPaused() {
        require(!paused(), "Pausable: paused");
        _;
    }

    modifier whenPaused() {
        require(paused(), "Pausable: not paused");
        _;
    }

    constructor() {
        _paused = false;
    }

    function paused() public view virtual returns (bool) {
        return _paused;
    }

    function _pause() internal virtual whenNotPaused {
        _paused = true;
        emit Paused(_msgSender());
    }

    function _unpause() internal virtual whenPaused {
        _paused = false;
        emit Unpaused(_msgSender());
    }
}

interface IKToken {
    function internalTransfer(
        address sender,
        address recipient,
        uint256 amount
    ) external returns (bool);

    function externalTransfer(
        address sender,
        address recipient,
        uint256 amount
    ) external returns (bool);
}

abstract contract KAP20 is IKAP20, IKToken, Pausable, AccessController {
    mapping(address => uint256) _balances;

    mapping(address => mapping(address => uint256)) internal _allowances;

    uint256 private _totalSupply;

    string public override name;
    string public override symbol;
    uint8 public override decimals;

    constructor(
        string memory _name,
        string memory _symbol,
        uint8 _decimals,
        address _kyc,
        address _committee,
        address _transferRouter,
        uint256 _acceptedKycLevel
    ) AccessController(_committee, _kyc, _acceptedKycLevel) {
        name = _name;
        symbol = _symbol;
        decimals = _decimals;
        transferRouter = _transferRouter;
    }
    function totalSupply() public view virtual override returns (uint256) {
        return _totalSupply;
    }

    function balanceOf(
        address account
    ) public view virtual override returns (uint256) {
        return _balances[account];
    }

    function transfer(
        address recipient,
        uint256 amount
    ) public virtual override whenNotPaused returns (bool) {
        _transfer(msg.sender, recipient, amount);
        return true;
    }

    function allowance(
        address owner,
        address spender
    ) public view virtual override returns (uint256) {
        return _allowances[owner][spender];
    }

    function approve(
        address spender,
        uint256 amount
    ) public virtual override returns (bool) {
        _approve(msg.sender, spender, amount);
        return true;
    }

    function transferFrom(
        address sender,
        address recipient,
        uint256 amount
    ) public virtual override whenNotPaused returns (bool) {
        _transfer(sender, recipient, amount);

        uint256 currentAllowance = _allowances[sender][msg.sender];
        require(
            currentAllowance >= amount,
            "KAP20: Transfer amount exceeds allowance"
        );
        unchecked {
            _approve(sender, msg.sender, currentAllowance - amount);
        }

        return true;
    }

    function increaseAllowance(
        address spender,
        uint256 addedValue
    ) public virtual returns (bool) {
        _approve(
            msg.sender,
            spender,
            _allowances[msg.sender][spender] + addedValue
        );
        return true;
    }

    function decreaseAllowance(
        address spender,
        uint256 subtractedValue
    ) public virtual returns (bool) {
        uint256 currentAllowance = _allowances[msg.sender][spender];
        require(
            currentAllowance >= subtractedValue,
            "KAP20: Decreased allowance below zero"
        );
        unchecked {
            _approve(msg.sender, spender, currentAllowance - subtractedValue);
        }

        return true;
    }

    function _transfer(
        address sender,
        address recipient,
        uint256 amount
    ) internal virtual {
        require(sender != address(0), "KAP20: Transfer from the zero address");
        require(recipient != address(0), "KAP20: Transfer to the zero address");

        uint256 senderBalance = _balances[sender];
        require(
            senderBalance >= amount,
            "KAP20: Transfer amount exceeds balance"
        );
        unchecked {
            _balances[sender] = senderBalance - amount;
        }
        _balances[recipient] += amount;

        emit Transfer(sender, recipient, amount);
    }

    function _adminTransfer(
        address sender,
        address recipient,
        uint256 amount
    ) internal virtual {
        uint256 senderBalance = _balances[sender];
        require(
            senderBalance >= amount,
            "KAP20: Transfer amount exceeds balance"
        );
        unchecked {
            _balances[sender] = senderBalance - amount;
        }
        _balances[recipient] += amount;

        emit Transfer(sender, recipient, amount);
        emit AdminTransfer(sender, recipient, amount);
    }

    function _mint(address account, uint256 amount) internal virtual {
        require(account != address(0), "KAP20: Mint to the zero address");

        _totalSupply += amount;
        _balances[account] += amount;
        emit Transfer(address(0), account, amount);
    }

    function _burn(address account, uint256 amount) internal virtual {
        require(account != address(0), "KAP20: Burn from the zero address");

        uint256 accountBalance = _balances[account];
        require(accountBalance >= amount, "KAP20: Burn amount exceeds balance");
        unchecked {
            _balances[account] = accountBalance - amount;
        }
        _totalSupply -= amount;

        emit Transfer(account, address(0), amount);
    }

    function _approve(
        address owner,
        address spender,
        uint256 amount
    ) internal virtual {
        require(owner != address(0), "KAP20: Approve from the zero address");
        require(spender != address(0), "KAP20: Approve to the zero address");

        _allowances[owner][spender] = amount;
        emit Approval(owner, spender, amount);
    }

    function adminTransfer(
        address sender,
        address recipient,
        uint256 amount
    ) public virtual override onlyCommittee returns (bool) {
        _adminTransfer(sender, recipient, amount);
        return true;
    }

    function internalTransfer(
        address sender,
        address recipient,
        uint256 amount
    ) external override whenNotPaused onlyTransferRouter returns (bool) {
        require(
            kyc.kycsLevel(sender) >= acceptedKycLevel &&
                kyc.kycsLevel(recipient) >= acceptedKycLevel,
            "KAP20: Only internal purpose"
        );

        _transfer(sender, recipient, amount);
        return true;
    }

    function externalTransfer(
        address sender,
        address recipient,
        uint256 amount
    ) external override whenNotPaused onlyTransferRouter returns (bool) {
        require(
            kyc.kycsLevel(sender) >= acceptedKycLevel,
            "KAP20: Only internal purpose"
        );

        _transfer(sender, recipient, amount);
        return true;
    }
}

contract ValidatorShare is
    IValidatorShare,
    Initializable,
    ReentrancyGuard,
    KAP20
{
    uint256 public constant REWARD_PRECISION = 10 ** 25;

    IStakeManagerStorage public immutable stakeManagerStorage;
    uint256 public validatorId;

    uint256 public rewardPerShare;
    uint256 public activeAmount;

    bool public delegation;
    uint256 public minDelegated;

    mapping(address => uint256) public initialRewardPerShare;

    IKKUB public immutable kkub;

    address public callHelper;
    ITransferRouter public posTransferRouter;

    bool public isOfficialPool;

    event Delegated(
        uint256 indexed validatorId,
        address indexed user,
        uint256 amount
    );
    event Undelegated(
        uint256 indexed validatorId,
        address indexed user,
        uint256 amount
    );
    event ClaimedRewards(
        uint256 indexed validatorId,
        address indexed user,
        uint256 amount
    );

    event DelegationUpdated(bool indexed delegation, address indexed caller);
    event MinDelegatedUpdated(
        uint256 indexed oldMinDelegated,
        uint256 indexed newMinDelegated,
        address indexed caller
    );

    event CallHelperUpdated(
        address indexed oldCallHelper,
        address indexed newCallHelper,
        address indexed caller
    );

    event PosTransferRouterUpdated(
        address indexed oldPosTransferRouter,
        address indexed newPosTransferRouter,
        address indexed caller
    );

    modifier onlyWhenDelegated() {
        require(delegation, "ValidatorShare: delegation is disabled");
        _;
    }

    modifier onlyCallHelper() {
        require(
            msg.sender == callHelper,
            "ValidatorShare: restricted only call helper"
        );
        _;
    }

    function _onlyStakeManager() private view {
        require(
            msg.sender == stakeManagerStorage.stakeManager(),
            "ValidatorShare: caller is not stake manager"
        );
    }

    modifier onlyStakeManager() {
        _onlyStakeManager();
        _;
    }

    receive() external payable {
        require(
            msg.sender == address(kkub),
            "ValidatorShare: KUB received from unknown origin"
        );
    }

    constructor(
        address kyc_,
        address committee_,
        address kkub_,
        address posTransferRouter_,
        address adminKap20Router_,
        address callHelper_,
        address stakeManagerStorage_,
        bool isOfficialPool_
    )
        KAP20(
            "BKC POS Delegator",
            "BKC-D",
            18,
            kyc_,
            committee_,
            adminKap20Router_,
            4
        )
    {
        callHelper = callHelper_;
        posTransferRouter = ITransferRouter(posTransferRouter_);

        kkub = IKKUB(kkub_);
        stakeManagerStorage = IStakeManagerStorage(stakeManagerStorage_);

        isOfficialPool = isOfficialPool_;
    }

    function setCallHelper(address callHelper_) external onlyCommittee {
        emit CallHelperUpdated(address(callHelper), callHelper_, msg.sender);
        callHelper = callHelper_;
    }

    function setPosTransferRouter(
        address posTransferRouter_
    ) external onlyCommittee {
        emit PosTransferRouterUpdated(
            address(posTransferRouter),
            posTransferRouter_,
            msg.sender
        );
        posTransferRouter = ITransferRouter(posTransferRouter_);
    }

    /*********************** External Functions *************************/
    /// @notice initialize this contract
    /// @param validatorId_ validator id
    /// @param minDelegated_ minimum amount of native token to delegate
    /// @dev this function is called only once during the contract deployment
    /// onlyOwner will prevent this contract from initializing, since it's owner is
going to be 0x0 address
    function initialize(
        uint256 validatorId_,
        uint256 minDelegated_
    ) external initializer {
        validatorId = validatorId_;
        minDelegated = minDelegated_;
        delegation = true;
    }

    /// @notice delegate to a validator
    /// @return amountToDeposit amount of native token deposited
    function delegate() external payable nonReentrant returns (uint256) {
        return _delegateCore(msg.sender, msg.value);
    }

    function delegate(
        uint256 amount_,
        address bitkubNext_
    ) external nonReentrant onlyCallHelper returns (uint256) {
        posTransferRouter.transferFromKKUB(bitkubNext_, address(this), amount_);
        kkub.withdraw(amount_);
        return _delegateCore(bitkubNext_, amount_);
    }
    function _delegateCore(
        address sender_,
        uint256 amount_
    ) private returns (uint256) {
        _officialPoolValidate(sender_);

        _withdrawAndTransferReward(sender_);

        require(
            balanceOf(sender_) + amount_ >= minDelegated,
            "ValidatorShare: amount is less than minimum delegated amount"
        );

        _buyShares(amount_, sender_);

        emit Delegated(validatorId, sender_, amount_);
        return amount_;
    }

    /// @notice undelegate from a validator
    /// @param claimAmount_ amount of native token to claim
    /// @return shares amount of shares burned
    function undelegate(
        uint256 claimAmount_
    ) external nonReentrant returns (uint256) {
        return _undelegateCore(claimAmount_, msg.sender);
    }

    function undelegate(
        uint256 claimAmount_,
        address bitkubNext_
    ) external nonReentrant onlyCallHelper returns (uint256) {
        return _undelegateCore(claimAmount_, bitkubNext_);
    }

    function _undelegateCore(
        uint256 claimAmount_,
        address sender_
    ) private returns (uint256) {
        _officialPoolValidate(sender_);

        claimAmount_ = _sellShares(claimAmount_, sender_);

        emit Undelegated(validatorId, sender_, claimAmount_);
        return claimAmount_;
    }

    function claimRewards() external nonReentrant returns (uint256) {
        return _claimRewardsCore(msg.sender);
    }

    function claimRewards(
        address bitkubNext_
    ) external nonReentrant onlyCallHelper returns (uint256) {
        return _claimRewardsCore(bitkubNext_);
    }

    function _claimRewardsCore(address sender_) private returns (uint256) {
        _officialPoolValidate(sender_);

        return _withdrawAndTransferReward(sender_);
    }

    function updateDelegation(bool delegation_) external onlyStakeManager {
        emit DelegationUpdated(delegation_, msg.sender);
        delegation = delegation_;
    }

    function updateMinDelegated(
        uint256 minDelegated_
    ) external onlyStakeManager {
        emit MinDelegatedUpdated(minDelegated, minDelegated_, msg.sender);
        minDelegated = minDelegated_;
    }

    /*********************** Public Functions *************************/

    function getLiquidRewards(address user_) public view returns (uint256) {
        return _calculateReward(user_, getRewardPerShare());
    }

    function getRewardPerShare() public view returns (uint256) {
        return
            _calculateRewardPerShareWithRewards(
                stakeManagerStorage.getValidatorInfoDelegatorsRewardByIndex(
                    validatorId
                )
            );
    }
```

```
/*********************** Private Functions **************************/

function _stakeManager() internal view returns (IStakeManager) {
    return IStakeManager(stakeManagerStorage.stakeManager());
}

function _transfer(
    address from_,
    address to_,
    uint256 value_
) internal override {
    // get rewards for recipient
    _withdrawAndTransferReward(to_);
    // convert rewards to shares
    _withdrawAndTransferReward(from_);
    // move shares to recipient
    super._transfer(from_, to_, value_);
}

function _adminTransfer(
    address from_,
    address to_,
    uint256 value_
) internal override {
    _withdrawAndTransferReward(from_);
    _withdrawAndTransferReward(to_);
    super._adminTransfer(from_, to_, value_);
}

function _sellShares(
    uint256 claimAmount_,
    address sender_
) private returns (uint256) {
    // first get how much staked in total and compare to target unstake amount
    uint256 totalStaked = balanceOf(sender_);
    require(
        totalStaked != 0 && totalStaked >= claimAmount_,
        "ValidatorShare: too much requested"
    );

    if (totalStaked - claimAmount_ < minDelegated) {
        claimAmount_ = totalStaked;
    }
```

```
        _withdrawAndTransferReward(sender_);

        _burn(sender_, claimAmount_);
        _stakeManager().updateValidatorState(
            validatorId,
            -int256(claimAmount_)
        );
        activeAmount = activeAmount - claimAmount_;

        require(
            _stakeManager().transferFunds(
                validatorId,
                claimAmount_,
                sender_,
                TransferTokenMode.Staked
            ),
            "ValidatorShare: insufficent rewards"
        );

        return claimAmount_;
    }

    function _withdrawReward(address user_) private returns (uint256) {
        uint256 localRewardPerShare = _calculateRewardPerShareWithRewards(
            _stakeManager().withdrawDelegatorsReward(validatorId)
        );
        uint256 liquidRewards = _calculateReward(user_, localRewardPerShare);

        rewardPerShare = localRewardPerShare;
        initialRewardPerShare[user_] = localRewardPerShare;
        return liquidRewards;
    }

    function _withdrawAndTransferReward(
        address user_
    ) private returns (uint256) {
        uint256 liquidRewards = _withdrawReward(user_);

        if (liquidRewards != 0) {
            require(
                _stakeManager().transferFunds(
                    validatorId,
                    liquidRewards,
                    user_,
                    TransferTokenMode.Reward
                ),
                "ValidatorShare: insufficent rewards"
            );
            emit ClaimedRewards(validatorId, user_, liquidRewards);
        }

        return liquidRewards;
    }

    function _buyShares(
        uint256 amount_,
        address user_
    ) private onlyWhenDelegated {
        _mint(user_, amount_);

        _stakeManager().updateValidatorState(validatorId, int256(amount_));
        activeAmount = activeAmount + amount_;

        bool success = _stakeManager().delegationDeposit{ value: amount_ }(
            validatorId
        );
        require(success, "ValidatorShare: deposit failed");
    }

    function _calculateRewardPerShareWithRewards(
        uint256 accumulatedReward_
    ) private view returns (uint256) {
        uint256 localRewardPerShare = rewardPerShare;
        if (accumulatedReward_ != 0) {
            uint256 totalShares = totalSupply();

            if (totalShares != 0) {
                localRewardPerShare =
                    localRewardPerShare +
                    ((accumulatedReward_ * REWARD_PRECISION) / totalShares);
            }
        }

        return localRewardPerShare;
    }

    function _calculateReward(
        address user_,
        uint256 rewardPerShare_
    ) private view returns (uint256) {
        uint256 shares = balanceOf(user_);
        if (shares == 0) {
            return 0;
        }

        uint256 localInitialRewardPerShare = initialRewardPerShare[user_];

        if (localInitialRewardPerShare == rewardPerShare_) {
            return 0;
        }

        return
            ((rewardPerShare_ - localInitialRewardPerShare) * shares) /
            REWARD_PRECISION;
    }

    function _officialPoolValidate(address sender_) private view {
        if (isOfficialPool) {
            require(
                _isBitkubNext(sender_),
                "ValidatorShare: sender is not BitkubNext"
            );
        }
    }
}

interface IKYC {
    function setKycCompleted(address _addr, uint256 _level) external;
}

contract ValidatorShareFactory {
    address[] public validatorPool;

    address public immutable kkub;
    address public immutable stakeManagerStorage;

    address public kyc;
    address public committee;
    address public posTransferRouter;
    address public adminKap20Router;
    address public callHelper;

    event ValidatorShareCreated(
        address indexed addressShare,
        address indexed owner
    );
    event KycUpdated(
        address indexed oldKyc,
        address indexed newKyc,
        address indexed caller
    );
    event CommitteeUpdated(
        address indexed oldCommittee,
        address indexed newCommittee,
        address indexed caller
    );
    event PosTransferRouterUpdated(
        address indexed oldPosTransferRouter,
        address indexed newPosTransferRouter,
        address indexed caller
    );
    event AdminKap20RouterUpdated(
        address indexed oldAdminKap20Router,
        address indexed newAdminKap20Router,
        address indexed caller
    );
    event CallHelperUpdated(
        address indexed oldCallHelper,
        address indexed newCallHelper,
        address indexed caller
    );

    modifier onlyCommittee() {
        require(
            msg.sender == committee,
            "ValidatorShareFactory: restricted only committee"
        );
        _;
    }

    modifier onlyStakeManager() {
        require(
            msg.sender ==
                IStakeManagerStorage(stakeManagerStorage).stakeManager(),
            "ValidatorShareFactory: restricted only stake manager"
        );
        _;
    }

    constructor(
        address kyc_,
        address committee_,
        address posTransferRouter_,
        address adminKap20Router_,
        address kkub_,
        address callHelper_,
        address stakeManagerStorage_
    ) {
        kyc = kyc_;
        committee = committee_;
        posTransferRouter = posTransferRouter_;
        adminKap20Router = adminKap20Router_;
        kkub = kkub_;
        callHelper = callHelper_;
        stakeManagerStorage = stakeManagerStorage_;
    }

    function create(
        uint256 validatorId_,
        uint256 minDelegated_,
        bool isOfficialPool_
    ) public onlyStakeManager returns (address) {
        ValidatorShare vs = new ValidatorShare(
            kyc,
            committee,
            kkub,
            posTransferRouter,
            adminKap20Router,
            callHelper,
            stakeManagerStorage,
            isOfficialPool_
        );

        vs.initialize(validatorId_, minDelegated_);

        address addr = address(vs);
        validatorPool.push(addr);

        ITransferRouter(posTransferRouter).addAllowedAddress(addr);
        IKYC(kyc).setKycCompleted(addr, 16);

        emit ValidatorShareCreated(addr, msg.sender);

        return addr;
    }
    function validatorPoolLength() external view returns (uint256) {
        return validatorPool.length;
    }

    ////////////////////////////////////////// setter
//////////////////////////////////////////

    function setKyc(address kyc_) external onlyCommittee {
        emit KycUpdated(kyc, kyc_, msg.sender);
        kyc = kyc_;
    }

    function setCommittee(address committee_) external onlyCommittee {
        emit CommitteeUpdated(committee, committee_, msg.sender);
        committee = committee_;
    }

    function setPosTransferRouter(
        address posTransferRouter_
    ) external onlyCommittee {
        emit PosTransferRouterUpdated(
            posTransferRouter,
            posTransferRouter_,
            msg.sender
        );
        posTransferRouter = posTransferRouter_;
    }

    function setAdminKap20Router(
        address adminKap20Router_
    ) external onlyCommittee {
        emit AdminKap20RouterUpdated(
            adminKap20Router,
            adminKap20Router_,
            msg.sender
        );
        adminKap20Router = adminKap20Router_;
    }

    function setCallHelper(address callHelper_) external onlyCommittee {
        emit CallHelperUpdated(callHelper, callHelper_, msg.sender);
        callHelper = callHelper_;
    }
}
```

Appendix G: Central Limit Theorem

The Central Limit Theorem states that the sampling distribution of the sample means approaches a normal distribution (i.e., a Bell curve) as the sample size gets larger. In other words, CLT is a statistical premise that, given a sufficiently large sample size from a population with a finite level of variance, the mean of all sampled variables from the same population will be approximately equal to the mean of the whole population. Furthermore, these samples approximate a normal distribution, with their variances being approximately equal to the variance of the population as the sample size gets larger. CLT can be described more precisely using the definition of a limit. The CDF of the standardized sample mean $(\bar{X} - \mu)/\sigma$ converges pointwise to the Cumulative distribution function (CDF) (Φ) of the standard normal distribution. This is shown with the integral:

$$\lim_{n\to\infty} \mathbb{P}\frac{\left(\overline{X}_n - \mu\right)}{\sigma} \leq z = \Phi(z)$$

CLT equation

Where: X_n is a sequence of sample data
P is a probability function which states the following equation.

$$\frac{1}{\sqrt{2\pi}} \int_{-\infty}^{z} e^{\frac{x^2}{2}} dz \,.$$

Equation of P

Appendix H: Wrap KUB contract (KKUB)

```
pragma solidity 0.6.6;

interface IAdminAsset {
    function isSuperAdmin(address _addr, string calldata _token) external view returns (bool);
}

interface IKYC {
    function kycsLevel(address _addr) external view returns (uint256);
}

interface IKAP20 {
    event Transfer(address indexed from, address indexed to, uint256 tokens);
    event Approval(address indexed tokenOwner, address indexed spender, uint256 tokens);

    function totalSupply() external view returns (uint256);

    function balanceOf(address tokenOwner) external view returns (uint256 balance);

    function allowance(address tokenOwner, address spender) external view returns (uint256 remaining);

    function transfer(address to, uint256 tokens) external returns (bool success);

    function approve(address spender, uint256 tokens) external returns (bool success);

    function transferFrom(address from, address to, uint256 tokens) external returns (bool success);

    function getOwner() external view returns (address);

    function batchTransfer(address[] calldata _from, address[] calldata _to, uint256[] calldata _value) external
returns (bool success);

    function adminTransfer(address _from, address _to, uint256 _value) external returns (bool success);
}

contract KKUB is IKAP20 {
    string public name     = "Wrapped KUB";
    string public symbol   = "KKUB";
    uint8  public decimals = 18;

    event Transfer(address indexed from, address indexed to, uint256 value);
    event Approval(address indexed tokenOwner, address indexed spender, uint256 value);
    event Deposit(address indexed dst, uint256 value);
    event Withdrawal(address indexed src, uint256 value);
    event Paused(address account);
    event Unpaused(address account);
```

```
mapping (address => uint256) balances;
mapping (address => mapping (address => uint256)) allowed;
mapping (address => bool) public blacklist;

IAdminAsset public admin;
IKYC public kyc;
bool public paused;

uint256 public kycsLevel;

modifier onlySuperAdmin() {
    require(admin.isSuperAdmin(msg.sender, symbol), "Restricted only super admin");
    _;
}

modifier whenNotPaused() {
    require(!paused, "Pausable: paused");
    _;
}

modifier whenPaused() {
    require(paused, "Pausable: not paused");
    _;
}

constructor(address _admin, address _kyc) public {
    admin = IAdminAsset(_admin);
    kyc = IKYC(_kyc);
    kycsLevel = 1;
}

function setKYC(address _kyc) external onlySuperAdmin {
    kyc = IKYC(_kyc);
}

function setKYCsLevel(uint256 _kycsLevel) external onlySuperAdmin {
    require(_kycsLevel > 0);
    kycsLevel = _kycsLevel;
}

function getOwner() external view override returns (address) {
    return address(admin);
}

fallback() external payable {
    deposit();
}

receive() external payable {
    deposit();
}

function deposit() public whenNotPaused payable {
    balances[msg.sender] += msg.value;
      emit Deposit(msg.sender, msg.value);
      emit Transfer(address(0), msg.sender, msg.value);
}

function withdraw(uint256 _value) public whenNotPaused  {
    require(!blacklist[msg.sender], "Address is in the blacklist");
    _withdraw(_value, msg.sender);
}

function withdrawAdmin(uint256 _value, address _addr) public onlySuperAdmin {
    _withdraw(_value, _addr);
}

function _withdraw(uint256 _value, address _addr) internal {
    require(balances[_addr] >= _value);
    require(kyc.kycsLevel(_addr) > kycsLevel, "only kyc address registered with phone number can withdraw");

    balances[_addr] -= _value;
    payable(_addr).transfer(_value);
    emit Withdrawal(_addr, _value);
    emit Transfer(_addr, address(0), _value);
}

function totalSupply() public view override returns (uint256) {
    return address(this).balance;
}

function balanceOf(address _addr) public view override returns (uint256) {
    return balances[_addr];
}

function allowance(address _owner, address _spender) public view override returns (uint256) {
    return allowed[_owner][_spender];
}

function approve(address _spender, uint256 _value) public override whenNotPaused returns (bool) {
    require(!blacklist[msg.sender], "Address is in the blacklist");
    _approve(msg.sender, _spender, _value);
    return true;
}

function _approve(address owner, address spender, uint256 amount) internal {
    require(owner != address(0), "KAP20: approve from the zero address");
    require(spender != address(0), "KAP20: approve to the zero address");
    allowed[owner][spender] = amount;
    emit Approval(owner, spender, amount);
}

function transfer(address _to, uint256 _value) public override whenNotPaused returns (bool) {
    require(_value <= balances[msg.sender], "Insufficient Balance");
    require(blacklist[msg.sender] == false && blacklist[_to] == false, "Address is in the blacklist");

    balances[msg.sender] -= _value;
    balances[_to] += _value;
    emit Transfer(msg.sender, _to, _value);

    return true;
}

function transferFrom(
    address _from,
    address _to,
    uint256 _value
) public override whenNotPaused returns (bool) {
    require(_value <= balances[_from]);
    require(_value <= allowed[_from][msg.sender]);
    require(blacklist[_from] == false && blacklist[_to] == false, "Address is in the blacklist");

    balances[_from] -= _value;
    balances[_to] += _value;
    allowed[_from][msg.sender] -= _value;
    emit Transfer(_from, _to, _value);
    return true;
}

function batchTransfer(
    address[] calldata _from,
    address[] calldata _to,
    uint256[] calldata _value
) external override onlySuperAdmin returns (bool) {
    require(_from.length == _to.length && _to.length == _value.length, "Need all input in same length");

    for (uint256 i = 0; i < _from.length; i++) {
        if(blacklist[_from[i]] == true || blacklist[_to[i]] == true){
              continue;
        }

        if (balances[_from[i]] >= _value[i]) {
            balances[_from[i]] -= _value[i];
            balances[_to[i]] += _value[i];
            emit Transfer(_from[i], _to[i], _value[i]);
        }
    }
    return true;
}

function adminTransfer(
    address _from,
    address _to,
    uint256 _value
) external override onlySuperAdmin returns (bool) {
    require(balances[_from] >= _value);
    balances[_from] -= _value;
    balances[_to] += _value;
    emit Transfer(_from, _to, _value);

    return true;
}

function pause() external onlySuperAdmin whenNotPaused {
    paused = true;
    emit Paused(msg.sender);
}

function unpause() external onlySuperAdmin whenPaused {
    paused = false;
    emit Unpaused(msg.sender);
}

function addBlacklist(address _addr) external onlySuperAdmin {
    blacklist[_addr] = true;
}

function revokeBlacklist(address _addr) external onlySuperAdmin {
    blacklist[_addr] = false;
}
}
```

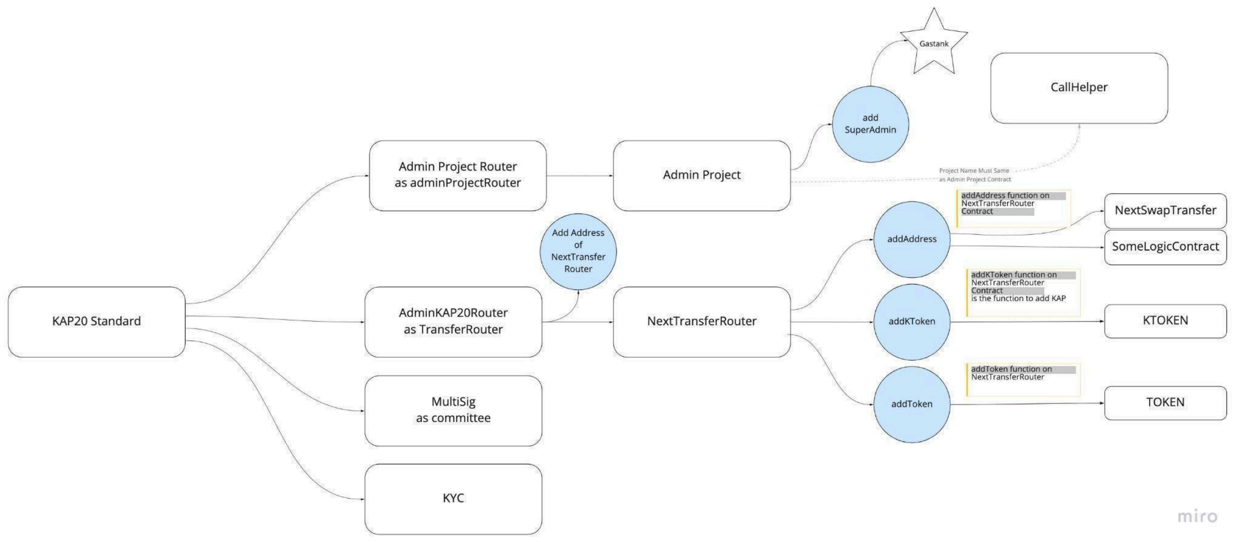
Appendix I: Register smart contract

```
// Sources flattened with hardhat v2.4.0 https://hardhat.org

// File contracts/interfaces/IAdmin.sol

pragma solidity 0.6.6;

interface IAdmin {
    function isSuperAdmin(address _addr) external view returns (bool);

    function isAdmin(address _addr) external view returns (bool);
}


// File contracts/KYCBitkubChainV2.sol

pragma solidity 0.6.6;

contract KYCBitkubChainV2 {
    IAdmin public admin;

    mapping(address => uint256) public kycsLevel;
    mapping(address => bool) public isAddressKyc;
    address[] public kycAddresses;

    // projectName => functionName => KYC Levels
    mapping(string => mapping(string => uint256)) public kycsProjectLevel;

    mapping(string => uint256) public kycTitleToLevel;
    mapping(uint256 => string) public kycLevelToTitle;

    uint256 public version = 2;

    event KycCompleted(address indexed addr, address indexed caller, uint256 previousLevel, uint256 level);
    event KycRevoked(address indexed addr, address indexed caller, uint256 previousLevel, uint256 level);

    event KycProject(address indexed _caller, string projectName, string functionName, uint256 level);
    event KycTitle(address indexed _caller, string title, uint256 level);

    modifier onlySuperAdmin() {
        require(admin.isSuperAdmin(msg.sender), "Restrict only super admin");
        _;
    }

    modifier onlyAdmin() {
        require(
            admin.isSuperAdmin(msg.sender) || admin.isAdmin(msg.sender),
            "Restrict only address is admin smart contract"
        );
        _;
    }

    constructor(address _admin) public {
        admin = IAdmin(_admin);
    }
```

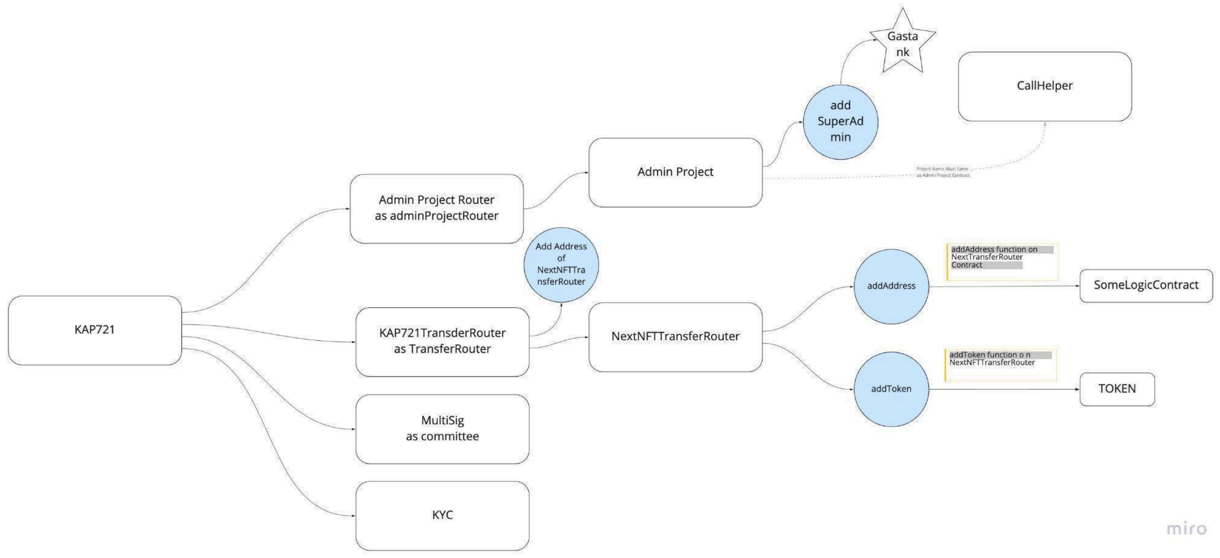
```
function kycAddressesLength() external view returns (uint256) {
    return kycAddresses.length;
}

function setAdmin(address _admin) external onlySuperAdmin {
    admin = IAdmin(_admin);
}

function _isPowerOfTwo(uint256 n) private pure returns (bool) {
    return n > 0 ? (n & (n - 1)) == 0 : false;
}

function setKycTitle(string calldata _title, uint256 _level) external onlySuperAdmin {
    require(_isPowerOfTwo(_level), "Level must be power of 2");

    kycTitleToLevel[_title] = _level;
    kycLevelToTitle[_level] = _title;

    emit KycTitle(msg.sender, _title, _level);
}

function setKycProjectLevel(
    string calldata _projectName,
    string calldata _functionName,
    uint256 kycsLevel_
) external onlySuperAdmin {
    kycsProjectLevel[_projectName][_functionName] = kycsLevel_;
    emit KycProject(msg.sender, _projectName, _functionName, kycsLevel_);
}

function setKycCompleted(address _addr, uint256 _level) public onlyAdmin {
    _setKycCompleted(_addr, _level);
}

function batchSetKycCompleted(address[] calldata _addrs, uint256 level) external onlyAdmin {
    for (uint256 i = 0; i < _addrs.length; i++) {
        _setKycCompleted(_addrs[i], level);
    }
}

function _setKycCompleted(address _addr, uint256 _level) internal {
    if (_level > 1) {
        uint256 previousLevel = kycsLevel[_addr];
        kycsLevel[_addr] = _level;

        if (!isAddressKyc[_addr]) {
            kycAddresses.push(_addr);
            isAddressKyc[_addr] = true;
        }

        emit KycCompleted(_addr, msg.sender, previousLevel, _level);
    }
}

// No kyc level set to no kyc
function setKycRevoked(address _addr) external onlyAdmin {
        _setKycRevoked(_addr);
    }

    function batchSetKycRevoked(address[] calldata _addrs) external onlyAdmin {
        for (uint256 i = 0; i < _addrs.length; i++) {
            _setKycRevoked(_addrs[i]);
        }
    }

    function _setKycRevoked(address _addr) internal {
        uint256 previousLevel = kycsLevel[_addr];
        kycsLevel[_addr] = 0;
        emit KycRevoked(_addr, msg.sender, previousLevel, 0);
    }
}
```

Appendix J: HyperBlock Smart contract & Gas tank flow

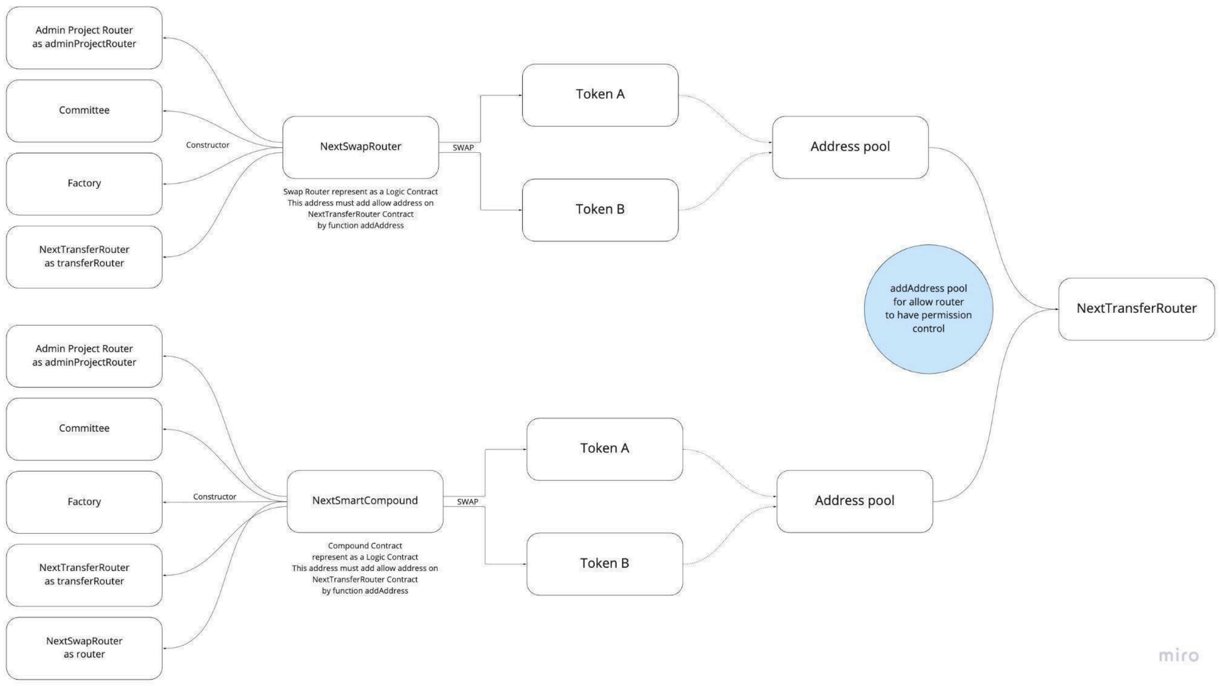
1. KAP-20 technical flow


Gastank
CallHelper
add SuperAdmin
Admin Project Router as adminProjectRouter
Admin Project
Project Name Must Same as Admin Project Contract
addAddress function on NextTransferRouter Contract
NextSwapTransfer
SomeLogicContract
Add Address of NextTransfer Router
addAddress
addKToken function on NextTransferRouter Contract is the function to add KAP
KAP20 Standard
AdminKAP20Router as TransferRouter
NextTransferRouter
addKToken
KTOKEN
addToken function on NextTransferRouter
addToken
TOKEN
MultiSig as committee
KYC
miro

2. KAP-721 technical flow


Gastank
CallHelper
add SuperAdmin
Admin Project
Project Name Must Same as Admin Project Contract
Admin Project Router as adminProjectRouter
Add Address of NextNFTTransferRouter
addAddress function on NextTransferRouter Contract
addAddress
SomeLogicContract
KAP721
KAP721TransderRouter as TransferRouter
NextNFTTransferRouter
addToken function o n NextNFTTransferRouter
addToken
TOKEN
MultiSig as committee
KYC
miro

3. Swap KAP-20 technical flow


Admin Project Router as adminProjectRouter
Committee
Factory
NextTransferRouter as transferRouter
Constructor
NextSwapRouter
SWAP
Swap Router represent as a Logic Contract This address must add allow address on NextTransferRouter Contract by function addAddress
Token A
Token B
Address pool
addAddress pool for allow router to have permission control
NextTransferRouter
Admin Project Router as adminProjectRouter
Committee
Factory
NextTransferRouter as transferRouter
NextSwapRouter as router
Constructor
NextSmartCompound
SWAP
Compound Contract represent as a Logic Contract This address must add allow address on NextTransferRouter Contract by function addAddress
Token A
Token B
Address pool
miro