RoC V2 migration plan
Integrators technical documentation

Date: 13 April 2024

INTRODUCTION

Upon the acceptance of the community of the proposal made on Proposal to Launch an enhanced version of the RIF on Chain Protocol, the Rif on Chain (RoC) protocol will be migrating to a new version (V2) that, while keeping the basic mechanisms as the previous (V1) protocol, introduces many new features and allows for further expansion on an updated tech stack.

It's worth noticing that this is not a simple upgrade, but a full migration that breaks compatibility, requiring both the interfaces (ABIs) and many contracts addressed will be changed. From the user perspective though, nothing will change, both USDRIF and RifPro holders won't be required to perform any action.

This document will explore in detail the technical requirement for Integrators to transition from the current implementation into V2.

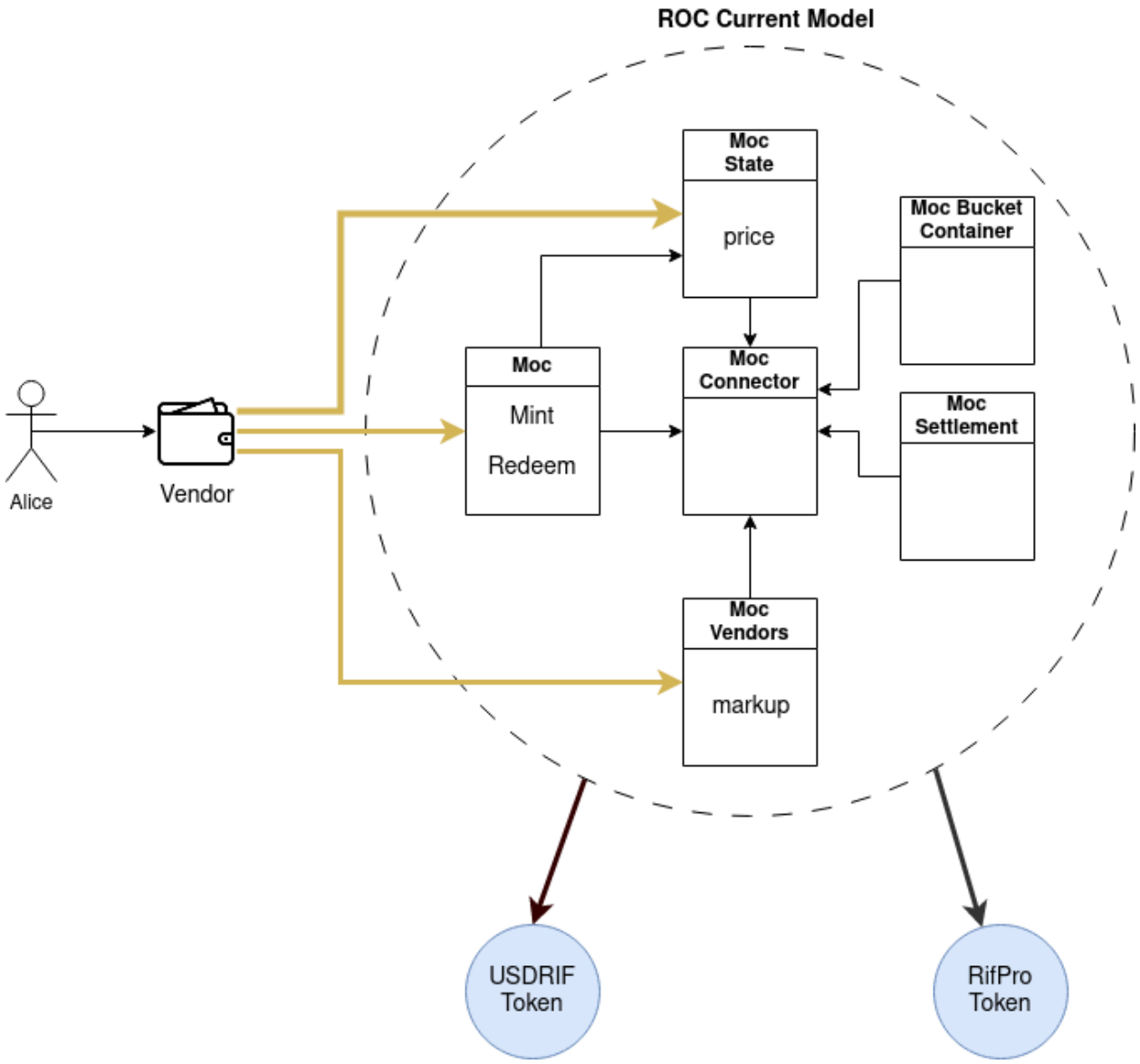
Rif on Chain V2 MIGRATION

Description

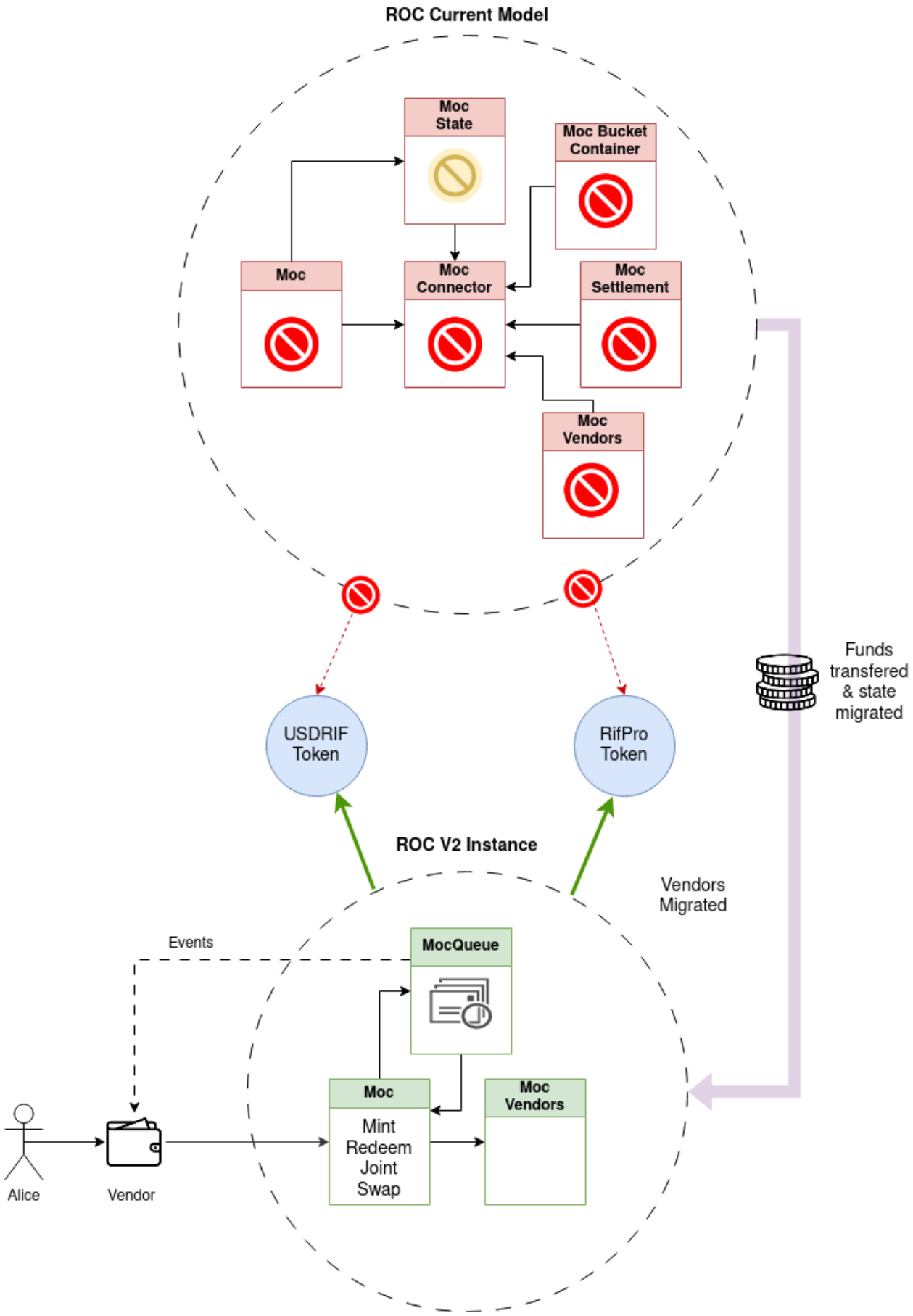
Vendors might possibly interact with the protocol in four ways, let's review how the migration impacts each of them:

1. Interact with USDRIF and RifPro tokens, for example, to get users balance, you now query to the tokens address directly as you would for any other ERC20 likeToken. This will not change, as the tokens remain the same on this migration.
2. Mint and redeem Tokens on behalf of a user. This will be done on a new contract and with a new ABI, although the end result will be equivalent, now it would involve a two steps process. When the user places the Operations, it first gets queued to later be executed in batches when conditions are met.
3. Get RifPro token price, you might have used MocState contract. In V2, this functionality will now be available on Moc itself along the operation methods.
4. Change your vendor markup. This was done by mocVendors contracts, and will also be migrated to a new contract, your current markup will need to be configured again. The Vendor Guardian will configure the markup on your behalf.

Let's explore current V1 Vendors interactions:



On current Model (V1), Alice will interact with her wallet through the Vendors dapp which will route her request through the Moc vendors specific methods, to apply the corresponding markup. This basic principle won't change, but it would happened on a new contracts after migration:


ROC Current Model
Moc State
Moc Bucket Container
Moc
Moc Connector
Moc Settlement
Moc Vendors
USDRIF Token
RifPro Token
Funds transfered & state migrated
ROC V2 Instance
Vendors Migrated
Events
MocQueue
Moc
Mint
Redeem
Joint
Swap
Moc Vendors
Alice
Vendor

As it is shown, after migration, all legacy contracts will be deprecated, and interacting with them will result in an error. So it's recommended that Vendors time this event with any adaptation on the Vendor's dapp to ease your users experience. The migration is atomic, and it will happen on a single transaction, so under normal circumstances no down time is expected.

Also, as mentioned before, USDRIF and RIFPRO tokens are not affected. A USDRIF will be equally redeemable for Rif collateral in V2 as it was in V1, holding its exact same pegged value. Same for RIFPRO. The rules that made up ROC protocol don't change, they just happened to be a subset of a larger scheme that V2 will allow to semesly expand.

Operational events will now be emitted from two different contracts, Moc (aka ROC) for Operation registering, and MocQueue for execution, more on this in the following sections.

Minting and redeeming changes

Although basic protocol operation at its core remains the same, new naming conventions have been introduced and new methods expressiveness criterias have been followed, let's compare each of the operation methods Vendors interact with and its equivalent in V2.

Note that this is merely a guide, for exact usage you should rely on the published ABI once the migration proposal is formalized.

New naming conventions

In the process of making a more generic protocol, you'll find that many names moved into that directions, here are some of the most relevant changes:

V1 Naming	**V2 Naming**
RiskPro Token	Token Collateral (TC)
Stable Token	Token Pegged (TP)
Reserve	Collateral Asset (CA)
Moc Token	Fee Token
mocCommission	qFeeToken (q from quantity)

Multi-Pegged protocol capability

You'll notice that on V2, there's an extra parameter on the operation methods, that indicates the TP address. This is because V2 is potentially multi-pegged, meaning that more than one pegged Token could be issued by the same collateral support, for example having USD and EUR issued on top of Rif.

USDRIF will be TP cero, and there is no short term plan for having more TPs, so you'll just need to pass USDRIF address as TP to keep it working as today.

Anti slippage methods expressiveness

In the way V1 interface was defined, there was the possibility that the context on which the users created his transactions and the one of the block on which it was actually included, could have changed, and the operational result might not exactly be the one the user was looking for. This is usually referred to as slippage in the financial context.

For example, let's suppose Alice wants to mint 100 USDRIF, she checks for the price (10), and following the protocol, she sends 1000 Rif to complete the operation:

```
mintStableToken(resTokensToMint = 1000)
```

But for any reason, her transaction takes several minutes to be mined, and in-between, the price has changed, and with current context, she receives 99 USDRIF. She might have needed to make a payment, and now that's not enough, needing her to perform a second transaction to get the extra USDRIF.

Methods in V2 prevent this mechanism by being more expressive on the user's intent, for example, to mint USDRIF, Alice will now indicate desired amount (qTP) as well as the maximum amount of Rif she is willing to pay for them.

```
mintTP(USDRIF, qTP = 100, qACmax = 1010, recipient, vendor)
```

Naturally, if the Collateral ends up falling short for the indicated pegged token quantity, the transactions will fail. And in an analogous way, if there is any reminder after meeting the desired amount, Alice will get it back.

Different recipient NOT supported

Even though V2 protocol has a different recipient address, RIF On Chain protocol will validate that the sender and recipient addresses are the same. A new parameter (*allowDifferentRecipient*) has been introduced to control this functionality and for security reasons, this parameter will remain **permanently deactivated**.

Deferred execution

V2 introduced a new step for Operations processing, along with a new contract, MocQueue to handle this. For details on the rationale behind this decision, see the Technical Proposal. Essentially, now when the user calls mint/redeem or any other Operation, he won't get the results on that same transaction, but just an OperationId, and he'll need to send an executionFee. His Operation will be queued along others, to be executed a couple of blocks down the line. This process will happen "automatically", not the user nor the Vendor needs to do anything (although the execute method is public). Only then, the operation will be evaluated, and the same rules as V1 apply. It can either success and yield the Operation specific event (see Main operations section for detail) or it can fail with:

```
event OperationError(uint256 operId_, bytes4 errorCode_, string msg_);
event UnhandledError(uint256 operId_, bytes reason_);
```

The OperationError event will capture all Protocol based errors, and will include both an *errorCode* and an english user friendly error *msg*. For example, if the Operation was rejected due to "*Low coverage*".

The *UnhandledError* on the other side, will capture all unsupported errors that shouldn't arise from an well intended use of the protocol or should be much less frequent, for example protocol paused.

For a complete list of errorCodes and reason, check contracts ABIs.

Execution Fee

The execution Fee, is a fee the user needs to pay upfront while queuing his Operation, to account for the gas consumption it's later execution will incur.

On V1, the user pays the Operation full cycle process on a single transaction, by paying the network fee (gas * gasCost) of the Transaction. On V2, this processing is split in two, registering the Operation and executing it; and the user will pay directly for one - as usual network fee - and indirect for the other, by sending a predefined amount of coinbase (rBTC) depending on the Operations. This fee will then be reworded to the executor to cover for this Operation execution gas consumption.
For example, let's suppose mintTC fixed execution fee is 15 Gwei and a mintTP is 18 Gwei, and Alice and Bob register a mintTC and mintTP Operations respectively, sending those **exact** amounts:

```
mintTC(...) { from: Alice, value: 15 Gwei }
mintTP(...) { from: Bob, value: 18 Gwei }
```

Then, when the queue containing thai two orders get executed, the sender will be rewarded with 33 Gwei.

It's worth noting that, even if V2 needs this extra payment, the total "gas cost" for the user, ends up being less than V1.

If *executionFee* is not the exact value the protocol expects, the transaction will revert with *WrongExecutionFee* error.

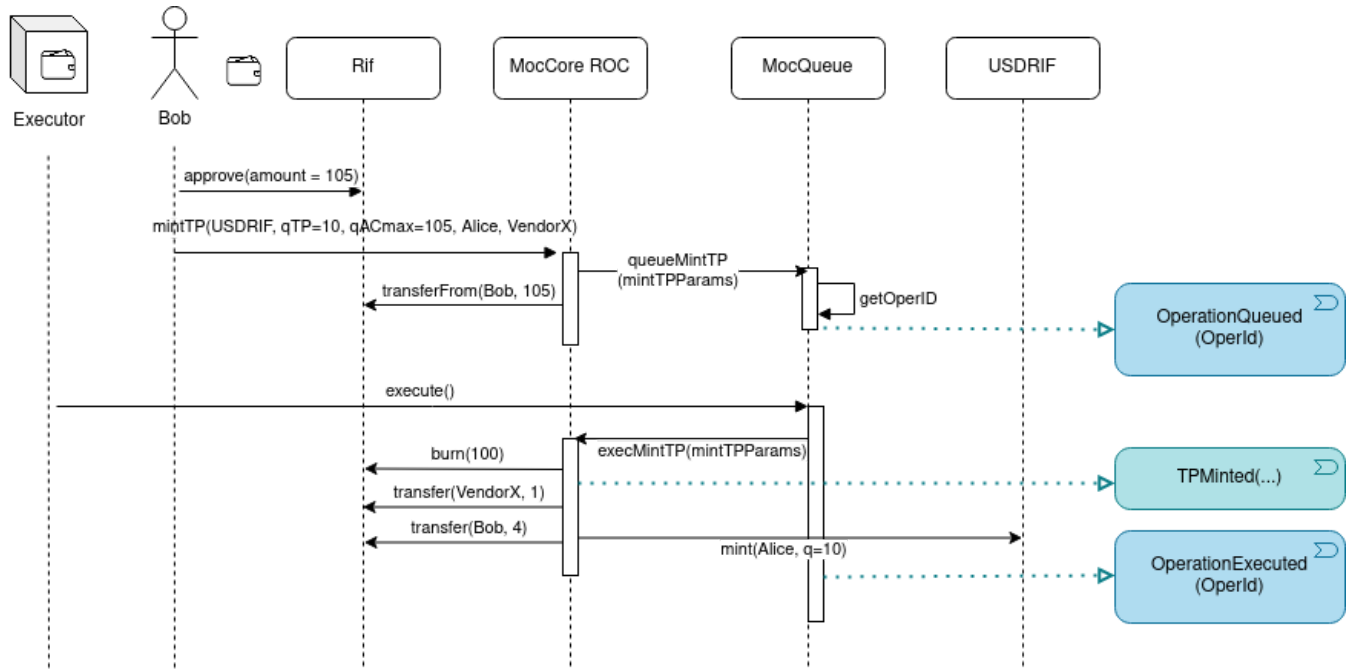
For detail on the executionFee values for each operation, please refer to the following table, repository configuration or query them on the smart contract itself. The latter would be the preferred option, as even if these values are fixed, they will be subject to regular analysis and eventual updates if the cost/reward falls out of balance.

Operation	Execution Fee
RedeemTC	6,909,000,000,000
MintTC	6,777,400,000,000
RedeemTP	7,994,700,000,000
MintTP	8,652,700,000,000
mintTCandTP	10,857,000,000,000
redeemTCandTP	11,482,100,000,000
swapTCforTP	9,705,500,000,000
swapTPforTC	9,212,000,000,000
swapTPforTP	10,133,200,000,000

Note: RSK gas price value is 65,800,000

Example:

Bob wants to mint 10 USDRIF to Alice using VendorX, which has a 1% markup. To simplify the example, we are not including *executionFee* not *platformFee*. Bob will invoke the *mintTP*, sending 105 Rif, even if the price is 10 and markup 1%, he sends 4 Rif extra to account for slippage. From this transaction, he'll get an *OperationQueued* event, with an *OperationId*.
Later, an executor will execute the queue, and Bob's Operations will be processed, this will yield two events related to his Operation, a generic *OperationExecuted(OperId)* and a *TPMinted* with the details of the results (see mintStableToken is now mintTP for detail). Notice that these later events came from the MocQueue contract, not from the one the user interacted with (ROC).
In the end, Alice will get her 10 USDRIF, VendorX his 1 markup RIF, and Bob will be refunded with the unused 4 extra Rif.



Main operations

mintRiskPro is now mintTC

V1:

```
/**
   @dev Mints RiskPro and pays the commissions of the operation.
   @param resTokensToMint Amount Reserve Tokens to spend in minting
   @param vendorAccount Vendor address
  */
 function mintRiskProVendors(uint256 resTokensToMint, address vendorAccount);

event RiskProMint(
   address indexed account,
   uint256 amount,
   uint256 reserveTotal,
   uint256 commission,
   uint256 reservePrice,
   uint256 mocCommissionValue,
   uint256 mocPrice,
   uint256 reserveTokenMarkup,
   uint256 mocMarkup,
  address vendorAccount
 );
```

V2:

```
/**
* @notice caller sends Collateral Asset and recipient receives Collateral
Token
*   `vendor_` receives a markup in Fee Token if possible or in qAC if not
*    Requires prior sender approval of Collateral Asset to this contract
* @param qTC_ amount of Collateral Token to mint
* @param qACmax_ maximum amount of Collateral Asset that can be spent
* @param recipient_ address who receives the Collateral Token
* @param vendor_ address who receives a markup
* @return qACtotalNeeded amount of AC used to mint qTC
* @return qFeeToken amount of Fee Token used by sender to pay fees. 0 if
qAC is used instead
*/
 function mintTC(
   uint256 qTC_,
   uint256 qACmax_,
   address recipient_,
   address vendor_
 )

  event TCMinted(
    address indexed sender_,
    address indexed recipient_,
    uint256 qTC_,
    uint256 qAC_,
    uint256 qACfee_,
    uint256 qFeeToken_,
    uint256 qACVendorMarkup_,
    uint256 qFeeTokenVendorMarkup_
  );
```

redeemRiskPro is now redeemTC

V1:

```
 /**
```

```
   @dev Redeems RiskPro Tokens and pays the commissions of the operation
in ReserveTokens
   @param riskProAmount Amount in RiskPro
   @param vendorAccount Vendor address
 */
 function redeemRiskProVendors(uint256 riskProAmount, address
vendorAccount);

 event RiskProRedeem(
   address indexed account,
   uint256 amount,
   uint256 reserveTotal,
   uint256 commission,
   uint256 reservePrice,
   uint256 mocCommissionValue,
   uint256 mocPrice,
   uint256 reserveTokenMarkup,
   uint256 mocMarkup,
   address vendorAccount
 );
```

V2:

```
   /**
    * @notice caller sends Collateral Token and recipient receives
Collateral Asset
    *  `vendor_` receives a markup in Fee Token if possible or in qAC if
not
    * @param qTC_ amount of Collateral Token to redeem
    * @param qACmin_ minimum amount of Collateral Asset that `recipient_`
expects to receive
    * @param recipient_ address who receives the Collateral Asset
    * @param vendor_ address who receives a markup, `0x0` if not using a
vendor
    * @return operId Identifier to track the Operation lifecycle
    */
   function redeemTC(
       uint256 qTC_,
       uint256 qACmin_,
       address recipient_,
```

```
        address vendor_
    ) external payable returns (uint256 operId);

    event TCRedeemed(
        address indexed sender_,
        address indexed recipient_,
        uint256 qTC_,
        uint256 qAC_,
        uint256 qACfee_,
        uint256 qFeeToken_,
        uint256 qACVendorMarkup_,
        uint256 qFeeTokenVendorMarkup_,
        address vendor_,
        uint256 operId_
    );
```

mintStableToken is now mintTP

V1:

```
/**
   @dev Mint StableToken tokens and pays the commissions of the operation
   @param resTokensToMint Amount in ReserveTokens to mint
   @param vendorAccount Vendor address
 */
 function mintStableTokenVendors(uint256 resTokensToMint, address
vendorAccount);

 event StableTokenMint(
   address indexed account,
   uint256 amount,
   uint256 reserveTotal,
   uint256 commission,
   uint256 reservePrice,
   uint256 mocCommissionValue,
   uint256 mocPrice,
   uint256 reserveTokenMarkup,
   uint256 mocMarkup,
   address vendorAccount
```

```
);
```

V2:

```
    /**
     * @notice caller sends Collateral Asset and recipient receives Pegged
Token
     *  `vendor_` receives a markup in Fee Token if possible or in qAC if
not
     *   Requires prior sender approval of Collateral Asset to this
contract
     * @param tp_ Pegged Token address to mint
     * @param qTP_ amount of Pegged Token to mint
     * @param qACmax_ maximum amount of Collateral Asset that can be spent
     * @param recipient_ address who receives the Pegged Token
     * @param vendor_ address who receives a markup, `0x0` if not using a
vendor
     * @return operId Identifier to track the Operation lifecycle
     */
    function mintTP(
        address tp_,
        uint256 qTP_,
        uint256 qACmax_,
        address recipient_,
        address vendor_
    ) external payable returns (uint256 operId);

    event TPMinted(
        address indexed tp_,
        address indexed sender_,
        address indexed recipient_,
        uint256 qTP_,
        uint256 qAC_,
        uint256 qACfee_,
        uint256 qFeeToken_,
        uint256 qACVendorMarkup_,
        uint256 qFeeTokenVendorMarkup_,
        address vendor_,
        uint256 operId_
    );
```

redeemStableToken is now redeemTP

V1:

```
 /**
   @dev Redeems the requested amount for the msg.sender, or the max amount
of free stableTokens possible.
   @param stableTokenAmount Amount of StableTokens to redeem.
   @param vendorAccount Vendor address
 */
 function redeemFreeStableTokenVendors(uint256 stableTokenAmount, address
vendorAccount);

 event FreeStableTokenRedeem(
   address indexed account,
   uint256 amount,
   uint256 reserveTotal,
   uint256 commission,
   uint256 interests,
   uint256 reservePrice,
   uint256 mocCommissionValue,
   uint256 mocPrice,
   uint256 reserveTokenMarkup,
   uint256 mocMarkup,
   address vendorAccount
 );
```

V2:

```
   /**
    * @notice caller sends Pegged Token and recipient receives Collateral
Asset
    *  `vendor_` receives a markup in Fee Token if possible or in qAC if
not
    * @param tp_ Pegged Token address to redeem
    * @param qTP_ amount of Pegged Token to redeem
```

```
    * @param qACmin_ minimum amount of Collateral Asset that `recipient_`
expects to receive
    * @param recipient_ address who receives the Collateral Asset
    * @param vendor_ address who receives a markup, `0x0` if not using a
vendor
    * @return operId Identifier to track the Operation lifecycle
    */
   function redeemTP(
       address tp_,
       uint256 qTP_,
       uint256 qACmin_,
       address recipient_,
       address vendor_
   ) external payable returns (uint256 operId);

   event TPRedeemed(
       address indexed tp_,
       address indexed sender_,
       address indexed recipient_,
       uint256 qTP_,
       uint256 qAC_,
       uint256 qACfee_,
       uint256 qFeeToken_,
       uint256 qACVendorMarkup_,
       uint256 qFeeTokenVendorMarkup_,
       address vendor_,
       uint256 operId_
   );
```

Vendors Markup changes

Even if Vendors requirement got quite simplified on V2, from migration and operational perspective there won't be any change. Your current address and markups configuration will be ported to the new Vendor contract and you'll be able to adjust it as usual there.
The biggest simplification comes from the removal of the staking requirement, which will allow you to un-stake your funds if desired. Though as we said, this won't affect operations.

Price querying changes

You might have been using a different contract from the operations one, MocState, to query for RifPro price (using `riskProTecPrice`). This contract, along with all of its public methods was also migrated and you can now find all the same information if new V2 one, all together with the operational methods.

V1:

```
function riskProTecPrice() external view returns(uint256);

function getReserveTokenPrice() external view returns(uint256);
```

V2:

```
  /**
   * @notice get Collateral Token price
   * @return pTCac [PREC]
   */
  function getPTCac() external view returns (uint256 pTCac);

  /**
   * @notice get how many Pegged Token equal 1 Collateral Asset
   * @param tp_ Pegged Token address
   * @return price [PREC]
   */
  function getPACtp(address tp_) public view virtual returns (uint256);
```

For *pACtp*, that is how many Pegged Token equals one Collateral Asset, you need to specify what Pegged Token address you need the price for. USDRIF in our case.

Benefits and new features

There are many benefits and new features yet to come, as V2 protocol flexibility and technical designs, allows for easier expansion and integration. But out of the box, besides what's already presented, V2 will bring a considerable network fee reduction.

Along with a new set of on-chain joint operations, that simplify processes and saves users both network and protocol fees, let's briefly explore them:

New Mint & Redeem joint and swap Operations

In situations in which the Protocol coverage is not healthy enough, some operations might be restricted or limited, for example minting USDRIF or redeeming RifPro. But maybe users really need some token liquidity, for example for other Defi applications; this might push him to look for other markets. On V2, we included methods to overcome this situation, allowing a user to both mint and redeem Pegged and Collateral Tokens, in a way that the rest of the protocol is not penalized, so he can get the desired Token.

mintTCandTP

```
    /**
     * @notice caller sends Collateral Asset and recipient receives Collateral Token and Pegged Token
     *  `vendor_` receives a markup in Fee Token if possible or in qAC if not
     *  Requires prior sender approval of Collateral Asset to this contract
     *  This operation is done without checking coverage
     *  Collateral Token and Pegged Token are minted in equivalent proportions so that its price
     *  and global coverage are not modified.
     *  Reverts if qAC sent are insufficient.
     * @param tp_ Pegged Token address
     * @param qTP_ amount of Pegged Token to mint
     * @param qACmax_ maximum amount of Collateral Asset that can be spent
     * @param recipient_ address who receives the Collateral Token and Pegged Token
     * @param vendor_ address who receives a markup, `0x0` if not using a vendor
     * @return operId Identifier to track the Operation lifecycle
     */
    function mintTCandTP(
        address tp_,
        uint256 qTP_,
        uint256 qACmax_,
        address recipient_,
        address vendor_
    ) external payable returns (uint256 operId);
```

redeemTCandTP

```
    /**
    * @notice Caller sends Collateral Token and Pegged Token and recipient receives Collateral Asset.
    *  `vendor_` receives a markup in Fee Token if possible or in Collateral Asset if not
    *  This operation is done without checking coverage
    *  Collateral Token and Pegged Token are redeemed in equivalent proportions so that their price
    *  and global coverage are not modified.
    *  Reverts if qTP sent are insufficient.
    * @param tp_ Pegged Token address
    * @param qTC_ Maximum amount of Collateral Token to redeem
    * @param qTP_ Maximum amount of Pegged Token to redeem
    * @param qACmin_ Minimum amount of Collateral Asset that `recipient_` expects to receive
    * @param recipient_ Address who receives the Collateral Asset
    * @param vendor_ address who receives a markup, `0x0` if not using a vendor
    * @return operId Identifier to track the Operation lifecycle
    */
   function redeemTCandTP(
       address tp_,
       uint256 qTC_,
       uint256 qTP_,
       uint256 qACmin_,
       address recipient_,
       address vendor_
   ) external payable returns (uint256 operId);
```

swapTPforTC

```
   /**
    * @notice caller sends a Pegged Token and recipient receives
Collateral Token
    *  `vendor_` receives a markup in Fee Token if possible or in qAC if
not
    * @param tp_ Pegged Token address
    * @param qTP_ amount of owned Pegged Token to swap
    * @param qTCmin_ minimum amount of Collateral Token that `recipient_`
expects to receive
    * @param qACmax_ maximum amount of Collateral Asset that can be spent
in fees
    * @param recipient_ address who receives the Collateral Token
    * @param vendor_ address who receives a markup, `0x0` if not using a
vendor
    * @return operId Identifier to track the Operation lifecycle
    */
   function swapTPforTC(
       address tp_,
       uint256 qTP_,
       uint256 qTCmin_,
       uint256 qACmax_,
       address recipient_,
       address vendor_
   ) external payable returns (uint256 operId);
```

swapTCforTP

```
/**
    * @notice Caller sends a Collateral Token and recipient receives
Pegged Token.
    *  `vendor_` receives a markup in Fee Token if possible or in qAC if
not.
    * @param tp_ Pegged Token address
    * @param qTC_ Amount of owned Collateral Token to swap
    * @param qTPmin_ Minimum amount of Pegged Token that `recipient_`
expects to receive
    * @param qACmax_ Maximum amount of Collateral Asset that can be spent
in fees
    * @param recipient_ Address who receives the Pegged Token
    * @param vendor_ address who receives a markup, `0x0` if not using a
vendor
    * @return operId Identifier to track the Operation lifecycle
    */
   function swapTCforTP(
       address tp_,
       uint256 qTC_,
       uint256 qTPmin_,
       uint256 qACmax_,
       address recipient_,
       address vendor_
   ) external payable returns (uint256 operId);
```