MaseerOne Protocol

Technical Whitepaper

June 5, 2025

1 Executive Summary

MaseerOne is an ERC-20 token contract with a built-in primary market. It enables on-chain issuance and redemption of tokenized real-world assets (RWAs), such as carbon credits, under conditions defined by a modular and upgradeable control system.

The protocol is centered around the MaseerOne contract, which inherits ERC-20 behavior from MaseerToken and embeds native mint and redemption mechanics. It integrates with external contracts — each deployed as its own upgradeable proxy — to enforce market constraints and governance logic. These include:

- **MaseerGate:** Market state and pricing.
- **MaseerPrice:** Asset NAV pricing.
- **MaseerTreasury:** Issuer authorization.
- **MaseerGuard:** Execution permissions.
- **MaseerConduit:** Token movement and fund custody.

Each module maintains its own internal permissioning (wards, rely, deny) and can be upgraded independently via proxy. MaseerOne does not control these modules — it simply reads from them or routes calls and funds through them to enforce protocol behavior.

The result is a composable, modular token architecture that cleanly separates token logic from control logic, making the system highly transparent, upgradeable, and suitable for managing regulated, collateralized asset issuance at scale.

2 Introduction

2.1 Problem Statement

Access to real-world assets (RWAs) such as regulated carbon credits, commodity instruments, or other environmental assets is typically limited to institutional investors operating through traditional financial infrastructure. For the general public, exposure is often only available indirectly through ETF-like products, which abstract away the underlying asset and introduce additional layers of custodial and regulatory overhead.

Even for sophisticated market participants, direct access to these asset classes is hampered by:

- Opaque pricing structures and over-the-counter settlement
- Restricted issuer participation and centralized gatekeeping
- Lack of public, composable pricing data for DeFi integration
- Inability to transact or settle on-chain with predictable terms

This fragmentation has left on-chain economies disconnected from global environmental and resource markets, making it difficult for decentralized applications, DAOs, and treasuries to hold or interact with RWAs in a transparent, trust-minimized way.

2.2 Vision and Objectives

MaseerOne is designed to bridge the gap between real-world asset issuance and decentralized financial infrastructure by creating an ERC-20 token with embedded, on-chain primary market mechanics. The protocol aims to:

- **Tokenize RWAs with native minting and redemption** logic, directly embedded in the token contract and governed by externalized market parameters.
- **Expose real-time NAV-based pricing** through an integrated oracle system (MaseerPrice), enabling deterministic mint and burn operations tied to underlying asset value.
- **Enforce issuance constraints** (e.g., capacity, cooldowns, pricing spreads) via modular control contracts like MaseerGate, which are independently upgradeable.
- **Cleanly separate asset logic from control logic**, allowing token behavior to remain stable while market rules, governance mechanisms, and pricing models evolve.
- **Offramp on-chain funds into regulated RWA markets** using MaseerConduit, which securely transfers capital received on-chain to real-world custodians, issuers, or settlement agents.
- **Support issuer-side control and authorization** through modules like MaseerTreasury and MaseerGuard, enabling permissioned participation in mint/redemption while preserving user-facing transparency.

By consolidating issuance and redemption into a single, standards-compliant token while delegating market conditions to modular components, MaseerOne brings RWAs on-chain in a secure, auditable, and operationally composable form.

3 Architecture Overview

3.1 Core Modules

MaseerOne operates as a self-contained ERC-20 token with embedded minting and redemption functionality. It enforces issuance rules and price integrity by reading from a set of independently governed and upgradeable modules, each responsible for a narrow operational concern. This separation of control enables governance, pricing, permissions, and flow constraints to evolve independently of the core token logic.

All modules are deployed as upgradeable proxies. Each contract maintains its own authentication model and is manipulated directly by authorized roles—MaseerOne has no control over any external module and only reads from them or sends value where required.

MaseerOne: The ERC-20 token itself. Inherits from MaseerToken. Includes built-in minting, redemption, and settlement functions. Serves as the system's on-chain asset address and market entry point.

MaseerGate: Defines market-wide constraints:

1. Whether minting or redemption is currently allowed
2. Capacity limits
3. Cooldowns
4. Pricing spreads (basis points over NAV)

Read-only by MaseerOne.

MaseerPrice: Oracle integration module that provides net asset value (NAV) pricing for the underlying RWA. NAV is used by MaseerOne to calculate minting costs and redemption value.

Read-only by MaseerOne.

MaseerTreasury: Stores the mapping of approved issuers and enforces participation restrictions across the protocol. MaseerOne reads from MaseerTreasury to determine whether an address is authorized to perform privileged actions such as:

1. Minting new tokens
2. Settling redemptions
3. Transferring tokens
4. Receiving approvals

This allows the protocol to prevent known bad actors or blacklisted addresses from interacting with the token in any meaningful way—even at the ERC-20 level. By enforcing checks in transfer() and approve(), the system preserves regulatory integrity and ensures that only compliant participants can hold or move the asset.

Read-only by MaseerOne.

MaseerGuard: Authorization gate for runtime checks. Controls whether an address is permitted to call critical functions (e.g., mint, redeem). MaseerOne queries MaseerGuard.pass().

Read-only by MaseerOne.

MaseerConduit: Handles value flow out of the protocol. Funds received during issuance are sent to MaseerConduit, which is responsible for routing assets to real-world custodians, brokers, or issuers in the RWA market.

Receives value from MaseerOne.

MaseerProxy: Used to deploy upgradeable proxy wrappers for all functional modules. Each module has its own instance of MaseerProxy, which delegates to its current implementation contract. All upgrade rights are gated via per-module relyProxy() access control.

3.2 System Flow

MaseerOne is an ERC-20 token with embedded issuance and redemption logic. It enforces dynamic constraints and permissioning by reading from external modules at runtime. These modules operate independently and are never written to or configured by MaseerOne.

Minting Process

1. User must first call approve(MaseerOne, amt) on the gem token to authorize payment.
2. User calls MaseerOne.mint(uint256 amt)
3. MaseerOne checks:
   (a) MaseerGate.openMint(), capacity(), cooldown(), and bpsin()
   (b) NAV pricing from MaseerPrice.read()
   (c) Permissioning via MaseerGuard.pass(msg.sender)
4. If all conditions are met:
   (a) gem is pulled via transferFrom()
   (b) Tokens are minted to the user or recipient

Redemption Process Redemption is a two-phase process:

1. **Phase 1** - Initiate Redemption
   - User calls MaseerOne.redeem(uint256 amt)
   - MaseerOne:
     (a) Checks MaseerGate.openBurn(), haltBurn(), cooldown(), bpsout()
     (b) Reads NAV from MaseerPrice.read()
     (c) Validates user via MaseerGuard.pass(msg.sender)
   - If valid:
     (a) Tokens are burned immediately
     (b) A redemption claim is recorded with a cooldown timestamp
2. **Phase 2** - Finalize Redemption
   - After the cooldown period defined by MaseerGate.cooldown(), the user calls MaseerOne.exit(uint256 id)
   - MaseerOne verifies the cooldown has passed and finalizes the redemption
   - This function may be externally triggered by the user, not an issuer

Transfers and Approvals

- All calls to transfer() and approve() are gated by MaseerGuard.pass()
- If the relevant address fails the pass() check, the transaction reverts
- No issuer or treasury lookup is used; Guard exclusively governs movement-level access

Upgrades and Governance

- Each module is a standalone upgradeable proxy via MaseerProxy
- Modules are upgraded via file() by addresses with relyProxy() access
- MaseerOne has no authority over module state or upgrade paths

4 Core Contract: MaseerOne

4.1 ERC-20 Behavior

MaseerOne is an immutable ERC-20 token contract with integrated runtime access control. It inherits from MaseerToken, which implements full ERC-20 behavior along with EIP-2612 permit support. While the system's pricing, gating, and redemption logic is modular and upgradeable via proxy patterns, MaseerOne itself is not upgradeable.

Token Characteristics

- **Name / Symbol / Decimals:** Set once at deployment
- **Total Supply:** Controlled by minting and burning under enforced constraints
- **Token Logic:** Immutable; deployed directly with no upgrade proxy
- **Permit Support:** EIP-2612 compliant

Permissioned Transfers and Approvals MaseerOne extends ERC-20 behavior by enforcing runtime permission checks via MaseerGuard:

- Every call to transfer(), transferFrom(), or approve() triggers MaseerGuard.pass(address)
- on the relevant participant(s): msg.sender, recipient, or spender.
- If any party fails the pass() check:
  - The transaction reverts
  - No token state is modified

This mechanism ensures that unauthorized actors cannot send, receive, or approve Maseer tokens—even indirectly—while maintaining full ERC-20 compatibility for approved users.

EIP-2612 Permit Support MaseerOne implements permit() per EIP-2612, enabling gasless approvals using signed messages.

- Nonces and domain separator follow EIP-712
- Enforced with MaseerGuard.pass(owner) and pass(spender)
- Allows composable DeFi usage within a permissioned compliance framework

This immutable ERC-20 design anchors the system, ensuring that token semantics cannot be changed post-deployment, while leaving pricing, redemption rules, and participant access fully dynamic and externally governed.

4.2 Minting Logic

MaseerOne enables controlled token issuance through its `mint(uint256 amt)` function. This function lets authorized users exchange approved collateral (`gem`) for newly minted Maseer tokens, under dynamic policy constraints enforced through runtime checks.

Collateral Setup

MaseerOne accepts a specific ERC-20 token (`gem`) as payment. Before calling `mint()`, the user must approve the MaseerOne contract:

```
gem.approve(MaseerOne, amt)
```

Function: `mint(uint256 amt)`

Execution Flow:

- **Access Control**
  - `MaseerGuard.pass(msg.sender)` must return `true`
  - Otherwise, the transaction reverts
- **Mint Conditions (Core Checks)**
  - `mintable()` must return `true` — reflects current system-wide policy and may depend on market or governance state
  - `capacity()` must not be exceeded — enforces protocol-wide issuance cap
  - `cooldown()` must permit the sender to mint again — avoids minting spam by enforcing per-address delay
  - Mint fee (`mintcost()`) is computed dynamically from NAV

- **Price and Fee Application**
  - NAV price is fetched from `MaseerPrice.read()`
  - Fee (in basis points) is applied to determine net issuance amount
- **Execution**
  - The contract pulls the input `amt` of `gem` from the user via `transferFrom()`
  - The net equivalent amount of Maseer tokens is minted and assigned to `msg.sender`

Gating Logic Summary

Constraint	Source	Purpose
`pass(msg.sender)`	`MaseerGuard`	Address-level authorization
`mintable()`	`MaseerOne`	System-wide mint enablement
`capacity()`	`MaseerOne`	Prevents over-issuance
`cooldown()`	`MaseerOne`	Throttles frequency per address
`read()`	`MaseerPrice`	Determines NAV-based mint ratio
`mintcost()`	`MaseerOne`	Calculates net fee-adjusted output

Table 1: Minting Constraints and Their Roles

This mechanism ensures that token issuance is:

- Externally priced
- Individually permissioned
- Globally rate-limited
- Modularly upgradable (via upstream modules like `MaseerPrice`, `MaseerGate`, and `MaseerGuard`)

4.3 Redemption Logic

MaseerOne supports redemptions via a two-step process: `redeem()` to initiate token burn and queue a redemption, and `exit()` to finalize repayment after a cooldown period. This design gives the protocol control over exit pacing and compliance, while allowing repayment to occur asynchronously by external actors.

Redemption Flow Summary

Step	Function	Purpose
1	`redeem(amt)`	Burns tokens and creates redemption claim
2	`exit(id)`	Finalizes redemption and transfers repayment (if available)

Table 2: Two-Step Redemption Flow

Step 1: `redeem(uint256 amt)`

When a user initiates a redemption:

- **Access Control**
  - `MaseerGuard.pass(msg.sender)` must return `true`
- **System State Checks**

  - `burnable()` must return `true`
  - `cooldown()` must permit the call

- **Fee and Accounting**
  - NAV is fetched from `MaseerPrice.read()` for UI/UX or off-chain pricing (not recorded on-chain)
  - Fee is calculated using `burncost()`

- **Execution**
  - User's tokens are burned immediately
  - A redemption record is stored, including:
    * Redeemer address
    * Total amount owed (after fee)
    * Timestamp
    * Fulfillment status (initially unfulfilled)
  - `redemptionCount()` is incremented

Note: There is no redemption capacity ceiling—redemptions are not volume-rate-limited.

Step 2: `exit(uint256 id)`

After the cooldown period, any authorized address may call `exit(id)` on behalf of the redeemer:

- **Access Control**
  - `MaseerGuard.pass(msg.sender)` must return `true`
  - Caller does not need to be the original redeemer

- **Validation**
  - The redemption ID must:
    * Exist and remain unfulfilled
    * Be past the cooldown timestamp
    * Have an unclaimed amount

- **Execution**
  - If sufficient `gem` is available:
    * Transfer the owed amount to the original redeemer
    * Mark the redemption (fully or partially) fulfilled
  - If insufficient:
    * Transfer only the available balance
    * Leave the redemption record open with remaining balance

This allows redemptions to be fulfilled incrementally as liquidity becomes available.

Settlement Mechanics

No automatic settlement occurs during `redeem()` or `exit()`. Instead, external actors (issuers, treasurers, or off-chain systems) are responsible for:

- Funding the contract with `gem`
- Calling `exit()` to settle pending redemptions

This ensures a clean separation between token lifecycle, accounting, and fiat/off-chain logistics.

Security Summary

Function	Gated By	Notes
`redeem()`	`MaseerGuard.pass(msg.sender)`	Prevents initiation by blocked users
`exit()`	`MaseerGuard.pass(msg.sender)`	Callable by any whitelisted address

Table 3: Redemption Function Security Constraints

4.4 Settlement Mechanics

The `settle()` function in MaseerOne transfers surplus `gem`—tokens received during minting—to the `MaseerConduit`. This function is used to offload funds that have been received by the protocol but are not obligated to pending redeemers.

Function: `settle()`

- **Signature:**

```
function settle() external returns (uint256)
```

- **Access Control:**
  - Requires `MaseerGuard.pass(msg.sender)` to succeed
- **Return Value:**
  - Returns the amount of `gem` transferred to the `MaseerConduit`

Purpose and Behavior

When users mint MaseerOne tokens, they deposit `gem` into the contract. Since tokens are issued immediately upon mint, these incoming `gem` funds are not owed to anyone—they are considered excess collateral. The `settle()` function moves this surplus to the `Conduit`, where it becomes available for offramping and off-chain real-world asset (RWA) acquisition.

Calling `settle()`:

- Reads the contract's current `unsettled()` amount (surplus `gem`)
- Transfers this entire amount to the `MaseerConduit`
- Returns the amount transferred

This function does not depend on or modify `obligated()`, which tracks redemption liabilities.

Clarification of Terms

Term	Description
`unsettled()`	The amount of `gem` received that is not owed to any redeemer
`obligated()`	The amount of `gem` the protocol owes to users who have redeemed
`settle()`	Transfers `unsettled()` to `MaseerConduit` and returns the transferred amount

Table 4: Key Terms Related to Settlement Logic

Operational Significance

- Provides liquidity to the Conduit for off-chain asset acquisition
- Separates token issuance from real-world capital deployment
- Allows minting to remain synchronous and non-blocking, while settlement can occur asynchronously

This separation of concerns supports a modular architecture, enabling the RWA offramping logic to evolve independently of core token issuance.

5 Supporting Modules

5.1 MaseerGate

`MaseerGate` governs the protocol's operational state, providing global toggles and parameters that control whether minting and redemption operations are currently permitted. It defines key economic conditions under which the system transitions between open and closed states and enforces timing constraints on redemptions.

Core Responsibilities

- **Market Open/Close Controls**
  - Exposes `mintable()` and `burnable()` flags that `MaseerOne` checks to determine whether minting or redemption requests may proceed.
- **Redemption Cooldown Management**
  - Specifies the required cooldown period between a user submitting a redemption request and being allowed to finalize it via `exit()`.
- **Capacity Enforcement**
  - Caps the maximum circulating supply of the token. Minting is disallowed if total supply exceeds the value returned by `capacity()`.

Key Methods

Method	Purpose
`mintable()`	Returns `true` if minting is currently allowed
`burnable()`	Returns `true` if redemptions may currently be submitted
`cooldown()`	Returns the required cooldown time (in seconds) for `exit()`
`capacity()`	Returns the max token supply permitted before halting mints

Table 5: Public Interface Methods of `MaseerGate`

Integration Points

- Queried by `MaseerOne.mint()` and `MaseerOne.redeem()` to enforce mint/burn eligibility
- Queried by `MaseerOne.exit()` to enforce redemption cooldowns
- Allows protocol maintainers to pause or resume economic activity without contract upgrades

5.2 MaseerGuard

`MaseerGuard` is the access control layer for the MaseerOne system. It defines a centralized `pass()` function that enforces transfer, approval, and interaction restrictions across the protocol. This enables the system to prevent known bad actors or blocked addresses from participating in core flows, while remaining modular and upgradeable.

Core Responsibilities

- **Address Screening**
  - Maintains dynamic rules to determine whether an address is allowed to interact with the protocol. These rules are enforced via `pass()` checks by `MaseerOne` and other modules.
- **ERC20-Level Gating**
  - Used in `transfer()`, `transferFrom()`, and `approve()` within `MaseerOne` to prevent blocked addresses from sending, receiving, or authorizing token transfers.
- **Protocol Access Control**
  - Called in minting, redemption, settlement, and exit flows to verify that the caller is not barred from participation.

Key Method

Method	Purpose
`pass(address)`	Returns `true` if the address is allowed to interact with the system

Table 6: Public Interface of `MaseerGuard`

Integration Points

- Queried by `MaseerOne` in:
  - `transfer()`, `transferFrom()`, `approve()`
  - `mint()`, `redeem()`, `exit()`, `settle()`
- Queried by supporting modules such as `MaseerConduit` and `MaseerTreasury` for privileged flows
- Acts as a pluggable compliance layer, enabling flexible policy enforcement without embedding logic in the token itself

5.3 MaseerPrice

`MaseerPrice` is the oracle module responsible for providing the current off-chain net asset value (NAV) of one MaseerOne token, denominated in `gem`. It acts as a single-point price feed and has no internal calculation or aggregation logic.

Core Responsibility

- **NAV Reporting**
  - Returns the price of 1 MaseerOne token in `gem`, as sourced from off-chain or external systems.

Key Method

Method	Purpose
`read()`	Returns the current off-chain NAV price in `gem` units

Table 7: Public Interface of `MaseerPrice`

Integration Points

- Called directly by `MaseerOne`
- Output is forwarded to `MaseerGate` for determining mint and redemption pricing

5.4 MaseerConduit

`MaseerConduit` is the offramp module responsible for receiving surplus `gem` from `MaseerOne` and directing it toward real-world asset (RWA) acquisition, exchanges, or other custodial flows. It acts as a controlled sink for protocol-owned liquidity that is not tied to outstanding obligations.

Core Responsibilities

- **Offramp Fund Routing**
  - Accepts unencumbered `gem` from `MaseerOne.settle()` and provides authorized operators the ability to transfer these funds to approved output destinations.
- **Dual Whitelist Model**
  - Enforces two independent allowlists:
    - **Operator Allowlist:** Determines which addresses may initiate transfers out of the Conduit.
    - **Output Allowlist:** Restricts which destination addresses may receive funds (e.g., exchanges, custodians, or `MaseerOne` itself for edge-case settlement top-ups).
- **Flexibility for Custodial Flows**
  - Can be upgraded to integrate with different banking, exchange, or on-chain liquidity venues for RWA purchases.

Integration Points

- Receives funds from `MaseerOne.settle()` only — plays no role in minting or redemption flows
- Internal logic includes no pricing, cooldown, or blacklisting enforcement
- Ensures compliance and safety for real-world execution by strictly gating operators and outputs

5.5 MaseerTreasury

`MaseerTreasury` manages the registry of issuer addresses permitted to call privileged functions such as `issue()`. It also provides emergency tooling for protocol maintainers to forcibly remove stuck or embargoed balances through superuser burn operations.

Core Responsibilities

- **Issuer Registry**
  - Stores and maintains a mapping of addresses authorized to call `MaseerOne.issue()`.
- **Superuser Burn Authority**
  - Allows approved Treasury operators to forcibly burn tokens from problematic accounts (e.g., frozen, blocked, or abandoned).

Integration Points

- Queried by `MaseerOne.issue()` to enforce issuer-only access
- Not involved in transfers, redemptions, minting, or user-level logic
- Provides administrative tooling for protocol governance and compliance

6 Governance and Upgradeability

The MaseerOne protocol is architected around upgradeable modules governed by multisig-controlled contracts. This enables flexible policy evolution—across pricing, redemption gating, access control, and fund routing—without disrupting the token contract or requiring holder migration.

6.1 Modular Upgrade Pattern

Each module—`MaseerGate`, `MaseerPrice`, `MaseerTreasury`, `MaseerGuard`, and `MaseerConduit`—is deployed via an upgradeable proxy. All modules inherit from a common implementation base, `MaseerImplementation`, to ensure consistent storage layout. This enables safe replacement of logic and supports modular, composable protocol upgrades.

- `MaseerOne` reads from these modules but does not mutate them
- Modules can be upgraded independently
- The token remains decoupled from changing business or policy logic

6.2 Core Contract Constraints

`MaseerOne` is not upgradeable. It inherits standard ERC-20 logic via `MaseerToken` and serves as the canonical entry point for minting, redemption, and token balance tracking.

- ERC-20 behavior, supply accounting, and allowances are fixed at deployment
- Privileged burn logic exists within `MaseerOne` and is gated by the issuer registry in `MaseerTreasury`
- This feature exists solely to allow removal of stuck or embargoed funds and may eventually be extended to support Layer 2 bridges or external markets

6.3 Governance Model

At launch, all upgradeable modules are managed by multisig contracts. These multisigs may:

- Upgrade or replace implementation contracts
- Configure policy parameters (e.g., basis point rates, minting capacity)
- Manage whitelists and issuer mappings

The system is designed for extensibility, with permissions that allow integration of external protocols—such as third-party oracles or cross-chain bridges—while preserving on-chain control and auditability.

6.4 Upgrade Safety and Isolation

Each upgradeable module is isolated from the token contract and from one another.

- `MaseerOne` cannot be mutated by modules
- Modules are sandboxed to avoid cross-contamination of state
- All upgrade actions are permissioned and externally auditable

Appendices

A. Contract Overview

Contract	Purpose
`MaseerOne`	Core ERC20 token contract with built-in mint and redemption logic
`MaseerToken`	Provides ERC20 implementation and `permit()` support
`MaseerGate`	Controls mint/redemption windows, capacity, cooldowns, and pricing rules
`MaseerPrice`	Oracle contract that returns NAV pricing data (1 token = X gem)
`MaseerConduit`	Offramp sink for surplus `gem` to be routed into RWA markets
`MaseerGuard`	Enforces access via `pass()` modifier; applies to user-facing flows
`MaseerTreasury`	Registry of issuer addresses and superuser burn rights
`MaseerPrecommit`	*[Omitted from core logic, used only for queuing intent to mint]*
`MaseerProxy`	Generic proxy contract used for all upgradeable modules

Table 8: Maseer Contract Overview

B. Role Summary

Role	Permissions
`User`	Can mint (when allowed), redeem, exit, transfer, and approve tokens
`Issuer`	Can call `issue()` via `MaseerOne` if registered in `MaseerTreasury`
`Operator`	May operate `MaseerConduit` if authorized via operator whitelist
`Owner/Multisig`	Can upgrade modules, configure gates, set pricing policy, etc.

Table 9: Role Permissions Summary

C. Whitelist and Access Systems

System	Purpose
`MaseerGuard.pass()`	Enforces blacklist and permission checks on sensitive user actions
`MaseerTreasury`	Manages issuer permissions and burn authorization
`MaseerConduit`	Separates operator and destination whitelists for offramp routing

Table 10: Whitelist and Access Control Mechanisms

D. Key Economic Constants

Parameter	Description	Source
`bpsin`	Minting markup applied to NAV	`MaseerGate`
`bpsout`	Redemption markdown applied to NAV	`MaseerGate`
`capacity()`	Maximum total supply before minting is disallowed	`MaseerGate`
`cooldown()`	Delay after `redeem` before user can call `exit()`	`MaseerGate`

Table 11: Key Economic Parameters and Sources

E. Upgrade Architecture

- All modules are deployed behind `MaseerProxy` contracts
- Implementation contracts inherit from `MaseerImplementation`
- `MaseerOne` is not upgradeable
- Upgrade permissions are governed by multisig contracts
- Authorization for future integrations (bridges, external oracles) is built in