



Midnight Token Standard

Clark Alesna clark@saib.dev
Rj Lacanlale rjlacanlale@saib.dev

May 5, 2025

Prepared by SAIB, Inc. on behalf of NMKR

Abstract

The **Midnight Token Standard (MTS)** defines a comprehensive framework for creating and managing both fungible and non-fungible tokens on the Midnight network. This standard supports two distinct token types: **non-programmable tokens** that exist directly in user wallets with minimal contract interaction, and **programmable tokens** that implement complex behaviors through contract-based custody and logic. MTS stores all token metadata attributes directly on-chain while media files are referenced via URIs, ensuring metadata permanence while maintaining storage efficiency. MTS leverages Midnight's native privacy features to enable selective disclosure of token data while maintaining compatibility with established token standards from other blockchain ecosystems. By providing a unified interface for various token implementations with fully on-chain metadata, MTS aims to simplify token creation and management while supporting advanced use cases that require privacy-preserving functionality.

Introduction

Tokens on the Midnight network are **first-class citizens**, meaning they exist natively in the ledger similar to native assets on Cardano. This represents a fundamental difference from token implementations on Ethereum, where tokens are implemented as entries in smart contract storage. While native tokens provide efficiency advantages, they traditionally lack the programmability and flexibility offered by contract-based token systems. However, Cardano has addressed this limitation through **CIP-143**, which introduces programmability for native tokens.

The **Midnight Token Standard (MTS)** addresses this limitation by providing a unified interface for both **non-programmable tokens** and **programmable tokens**. This approach gives developers the freedom to choose the token implementation that best suits their specific use case, balancing simplicity, efficiency, and functional requirements.

MTS draws inspiration from established token standards across blockchain ecosystems, including **ERC-20**, **ERC-721**, and **ERC-1155** on Ethereum, as well as the Cardano token standards (**CIP-25**, **CIP-68**, and **CIP-143**). By adopting familiar interfaces and patterns where appropriate, MTS aims to facilitate cross-ecosystem integration and developer onboarding.

Midnight Network Architecture

The Midnight network's architecture provides several technical capabilities that influence this token standard:

- **Native Token Support:** Tokens exist as first-class citizens in the protocol rather than as smart contract mappings
- **Privacy Mechanisms:** Zero-knowledge proofs enable selective disclosure of transaction data
- **Programmable Circuits:** Smart contracts can implement complex logic while preserving privacy properties
- **Compact Language:** A TypeScript-based domain-specific language designed for implementing privacy-preserving smart contracts

Token Types

MTS defines two distinct token types:

Non-Programmable Tokens

Non-programmable tokens are similar to native assets with minimal contract involvement:

- **Direct ownership:** Tokens are held directly in user wallets
- **Simple transfers:** Transfers occur directly between wallets without contract execution
- **Metadata tracking:** The contract maintains token metadata and total supply information
- **Efficiency:** Reduced computational overhead for token operations
- **Ideal use cases:** Fungible tokens, simple NFTs, financial assets, loyalty points

Programmable Tokens

⚠ Warning: As of the time of writing, it is not yet technically possible for Midnight contracts to hold tokens other than DUST. Therefore, programmable tokens that require contract custody may not be implementable until this capability is added in future network updates.

Programmable tokens leverage contract capabilities for advanced behaviors:

- **Contract custody:** The contract retains custody of all tokens
- **Managed transfers:** All transfers pass through contract logic
- **Custom rules:** Programmable restrictions and conditions on transfers
- **Advanced features:** Time locks, vesting schedules, access controls, royalty enforcement
- **Ideal use cases:** Game assets, regulated tokens, identity tokens, royalty-bearing NFTs

This dual-type approach allows developers to select the appropriate token implementation based on their specific requirements for programmability, efficiency, and user experience.

Specification

This section defines the technical requirements for implementing the Midnight Token Standard. An MTS-compliant implementation must adhere to the interface and behavior specifications outlined below.

🔑 Key Insight: The MTS specification establishes a contract interface that supports both non-programmable and programmable tokens within the same framework, with metadata for both token types stored in Compact smart contracts on the Midnight network.

The standard specifies the required **contract state**, **interface functions**, and **metadata format** that implementations must follow to ensure interoperability within the Midnight ecosystem. The specification is designed to be minimally prescriptive while providing clear guidelines for compatibility across wallets, marketplaces, and applications.

Data Structures

Metadata Structure

The core data structure in MTS is the **Metadata** structure, which embeds all token properties directly on-chain:


```
struct MediaFile {
    name: Bytes<64>;
    mediaType: Bytes<32>;           // MIME type like "image/png"
    src: Bytes<256>;               // URI (https://, ipfs://, ar://, etc.)
}

struct Metadata {
    name: Bytes<32>;
    symbol: Maybe<Bytes<10>>;
    decimals: Maybe<Uint<8>>;
    description: Maybe<Bytes<256>>;
    image: Maybe<Bytes<256>>;       // Primary image URI
    mediaType: Maybe<Bytes<32>>;   // MIME type of primary image
    files: Maybe<Vector<MediaFile, 5>>; // Additional media files
    attributes: Maybe<Map<Bytes<32>, Bytes<128>, 20>>; // Key-value traits
```


```
version: Uint<16>;
}
```


Each field serves a specific purpose:


- **name**: Human-readable name of the token (REQUIRED)
- **symbol**: Short ticker symbol for the token (OPTIONAL)
- **decimals**: Number of decimal places for fungible tokens (OPTIONAL)

 **Tip**: Fungible tokens typically use 0-18 decimals and NFTs should omit this field.

- **description**: Brief description of the token (OPTIONAL)
- **image**: Primary image URI supporting various protocols (https://, ipfs:
- **mediaType**: MIME type of the primary image, e.g., "image/png" (OPTIONAL)
- **files**: Array of additional media files with their metadata (OPTIONAL)
- **attributes**: Key-value map for traits, properties, or custom metadata (OPTIONAL)
- **version**: Version number for the token standard implementation (REQUIRED)

 **Tip**: All metadata is stored directly on-chain, eliminating dependencies on external storage. For larger media files, IPFS CIDs can be used instead of base64 encoded content.


 **Key Insight**: The version field must be set to 1 for MTS v1 implementations. Future versions of the standard will increment this value. Implementations should validate version compatibility when interacting with tokens from other contracts.

 **Warning**: While metadata is stored on-chain, media files are referenced via URIs. Ensure URIs point to reliable, persistent storage solutions like IPFS or Arweave.

Media Storage Approach

MTS stores all metadata attributes on-chain while referencing media files via URIs. Supported URI schemes include:

- **HTTPS**: Traditional web URLs (e.g., https://example.com/image.png)
- **IPFS**: Decentralized storage (e.g., ipfs://QmXxx...)
- **Arweave**: Permanent storage (e.g., ar://xxx...)
- **Data URIs**: Small embedded content (e.g., data:image/svg+xml;base64,...)

 **Key Insight**: This hybrid approach balances on-chain permanence of metadata with practical storage of media files. All descriptive metadata lives on-chain while media files use appropriate external storage solutions.

Contract State

Common State Properties

All MTS-compliant contracts must maintain the following ledger state, regardless of token type:

```
// Common properties for all MTS contracts
export ledger project_name: Opaque<"string">;
export ledger metadata: Map<Bytes<32>, Metadata>;
export ledger total_supply: Map<Bytes<32>, Uint<128>>;
export ledger is_programmable: bool;
```

- **project_name**: The name of the project or token collection
- **metadata**: A mapping of token IDs to their metadata
- **total_supply**: A mapping of token IDs to their total supply
- **is_programmable**: A boolean indicating whether the token is programmable or not

The **is_programmable** flag is particularly important as it signals to wallets and applications how to interact with the token. When set to false, the token is treated as non-programmable with direct wallet transfers. When set to true, all transfers must go through the contract's logic.

Programmable Token State

Contracts implementing programmable tokens must maintain additional state to track ownership and permissions:

```
// Additional state for programmable tokens
export ledger balances: Map<Bytes<32>, Map<Bytes<32>, Uint<128>>>;
export ledger approvals: Map<Bytes<32>, Map<Bytes<32>, Map<Bytes<32>, bool>>>;
```

- **balances:** Maps token ID to a map of owner addresses and their balance amounts
- **approvals:** Maps owner-operator address pairs to approval status

⚠ Warning: Programmable token implementations must consistently maintain balance and approval state to prevent unauthorized transfers or token loss. All balance updates should properly validate inputs and maintain supply invariants.

Event Logging

While Midnight's event system is still evolving, implementations should prepare for future event emission capabilities by documenting the following standard events in comments:

```
// Standard events to be emitted when supported:
// Transfer(from: Bytes<32>, to: Bytes<32>, token_id: Bytes<32>, amount: Uint<128>)
// Approval(owner: Bytes<32>, operator: Bytes<32>, token_id: Bytes<32>, approved: bool)
// Mint(to: Bytes<32>, token_id: Bytes<32>, amount: Uint<128>)
```

💡 Tip: Once Midnight supports event emission, these events will enable efficient indexing and monitoring of token activities by wallets, marketplaces, and other applications.

Interface Functions

MTS defines a set of functions that implementations must provide to be considered compliant. Note that constructor and minting functions are implementation-specific and not part of the standard interface.

Public Ledger State

Both non-programmable and programmable tokens expose metadata and total supply as public ledger state:

💡 Tip: Since metadata and total_supply are public ledger state, they can be queried directly from the blockchain without requiring dedicated getter functions.

Programmable Token Functions

Contracts implementing programmable tokens must additionally implement:

```
// Required only for programmable tokens
export circuit transfer(to: Bytes<32>, token_id: Bytes<32>, amount: Uint<128>): bool;
export circuit transferFrom(from: Bytes<32>, to: Bytes<32>, token_id: Bytes<32>, amount: Uint<128>): bool;
export circuit approve(operator: Bytes<32>, token_id: Bytes<32>, approved: bool): bool;
```

- **transfer(to, token_id, amount):** Transfers tokens from the sender to the recipient

- **to**: The recipient's address
- **token_id**: The ID of the token being transferred
- **amount**: The amount of tokens to transfer
- **transferFrom(from, to, token_id, amount)**: Transfers tokens from one address to another on behalf of the owner (requires approval)
 - **from**: The token owner's address
 - **to**: The recipient's address
 - **token_id**: The ID of the token being transferred
 - **amount**: The amount of tokens to transfer
- **approve(operator, token_id, approved)**: Grants or revokes an operator's permission to manage the caller's tokens
 - **operator**: The address of the operator
 - **token_id**: The ID of the token for which approval is being set
 - **approved**: A boolean indicating whether the operator is approved or not

Implementation Guide

This section provides reference implementations of MTS-compliant contracts for both non-programmable and programmable tokens. These implementations demonstrate the basic structure and required interfaces while allowing flexibility for developers to extend with additional functionality.

Key Insight: The example implementations serve as templates and should be adapted to specific use cases. Security considerations, such as access control and input validation, should be incorporated into production implementations.

Programmable Token Implementation

The following is a reference implementation for a programmable token contract:

```
pragma language_version >= 0.14.0;

import CompactStandardLibrary;

struct MediaFile {
    name: Bytes<64>;
    mediaType: Bytes<32>;
    src: Bytes<256>;
}

struct Metadata {
    name: Bytes<32>;
    symbol: Maybe<Bytes<10>>;
    decimals: Maybe<Uint<8>>;
    description: Maybe<Bytes<256>>;
    image: Maybe<Bytes<256>>;
    mediaType: Maybe<Bytes<32>>;
    files: Maybe<Vector<MediaFile, 5>>;
    attributes: Maybe<Map<Bytes<32>, Bytes<128>, 20>>;
    version: Uint<16>;
}

// common properties
export ledger project_name: Opaque<"string">;
export ledger metadata: Map<Bytes<32>, Metadata>;
export ledger total_supply: Map<Bytes<32>, Uint<128>>;
export ledger is_programmable: bool;

// programmable properties
export ledger balances: Map<Bytes<32>, Map<Bytes<32>, Uint<128>>>;
export ledger approvals: Map<Bytes<32>, Map<Bytes<32>, Map<Bytes<32>, bool>>>;

constructor(name: Opaque<"string">) {
    // Initialize the contract with the project name
    // Implementation decides how to set is_programmable and other initial state
}

export circuit transfer(to: Bytes<32>, token_id: Bytes<32>, amount: Uint<128>): bool
{
    // This function transfers a specified amount of a token ID to a given address
}

export circuit transferFrom(from: Bytes<32>, to: Bytes<32>, token_id: Bytes<32>, amount: Uint<128>):
bool {
```



```
// This function transfers tokens from one address to another on behalf of the owner
// Requires prior approval from the token owner
}

export circuit approve(operator: Bytes<32>, token_id: Bytes<32>, approved: bool): bool {
  // This function approves or disapproves an operator for a specific token ID
}

export circuit mint(metadata: Metadata): [] {
  // Implementation-specific: decides token_id generation, version setting, etc.
}
```

Non-programmable Token Implementation

The following is a reference implementation for a non-programmable token contract:

```
pragma language_version >= 0.14.0;

import CompactStandardLibrary;

struct MediaFile {
    name: Bytes<64>;
    mediaType: Bytes<32>;
    src: Bytes<256>;
}

struct Metadata {
    name: Bytes<32>;
    symbol: Maybe<Bytes<10>>;
    decimals: Maybe<Uint<8>>;
    description: Maybe<Bytes<256>>;
    image: Maybe<Bytes<256>>;
    mediaType: Maybe<Bytes<32>>;
    files: Maybe<Vector<MediaFile, 5>>;
    attributes: Maybe<Map<Bytes<32>, Bytes<128>, 20>>;
    version: Uint<16>;
}

// common properties
export ledger project_name: Opaque<"string">;
export ledger metadata: Map<Bytes<32>, Metadata>;
export ledger total_supply: Map<Bytes<32>, Uint<128>>;
export ledger is_programmable: bool;

constructor(name: Opaque<"string">) {
    // Initialize the contract with the project name
    // Implementation decides how to set is_programmable and other initial state
}

export circuit mint(metadata: Metadata): [] {
    // This function mints a new token and updates the metadata, total_supply and properties. This
    is application-specific.
}
```

⚠ Warning: Non-programmable token implementations should carefully validate minting operations, as tokens cannot be recovered through contract logic once they are in user wallets.

💡 Tip: Token ID generation is implementation-specific. Common approaches include:

- Sequential counters for NFT collections
- Hash-based IDs derived from metadata
- User-specified IDs with uniqueness validation
- Fixed IDs for fungible tokens (e.g., "0x00" for a single fungible token type)

Future Considerations

This standard represents the initial version of the MTS specification and will evolve to incorporate additional capabilities and address emergent requirements.

Privacy Features

The current specification does not fully incorporate Midnight's privacy features, which will be formally addressed in subsequent versions. Future iterations will define:

- **Selective Disclosure:** Mechanisms for privacy-preserving metadata and ownership information
- **Private Transactions:** Protocols for shielded transfers while maintaining token functionality

Standardization Process

The MTS will undergo continuous refinement through community feedback and implementation experience. Contributors are encouraged to:

- Provide feedback on specification clarity and technical accuracy
- Submit implementation examples and extensions
- Propose improvements to enhance interoperability and functionality

The goal remains to establish a robust, flexible, and privacy-preserving token standard that meets the needs of the Midnight ecosystem while maintaining compatibility with broader blockchain standards.

References

- ERC-20: Fungible Token Standard
- ERC-721: Non-Fungible Token Standard
- ERC-1155: Multi Token Standard
- CIP-25: Media NFT Metadata Standard - Inspiration for on-chain metadata approach
- CIP-68: Datum Metadata Standard
- CIP-143: Interoperable Programmable Tokens
- Midnight Network Documentation