STRATO Mercata Token Standards

Contents

1. Abstract
2. Motivation
3. Specification
    a. Asset Contract
    b. Sale Contract
    c. Order Contract
4. Technical Implementation
5. Security Considerations
6. Conclusion

Abstract:

UTXO Model Reimagined for Real World Assets
The STRATO token standard introduces an innovative approach to asset management on blockchain platforms, drawing inspiration from the UTXO (Unspent Transaction Output) model traditionally utilized by Bitcoin. This novel standard revolutionizes the way digital assets are handled by encoding the asset state within discrete transaction outputs. Essentially, every Real World Asset is a UTXO.

Enhanced Traceability with Unique Asset Identity
Each asset within the Mercata application maintains its unique identity and history, akin to the UTXO model, ensuring enhanced traceability and ownership clarity.

Revolutionary Transfer Mechanism
One of the groundbreaking features of this standard is the Asset contract's ownership transfer mechanism. In stark contrast to traditional ERC-20 tokens, assets within the STRATO ecosystem can only be transferred through a Sale contract.

Control and Security in RWA Transactions
This unique transfer approach provides a layer of control and security, which is particularly advantageous for high-value assets and complex transaction requirements.

Pioneering RWA Liquidity and Scalability
The STRATO token standard is poised to redefine asset liquidity on decentralized platforms. By coupling the proven benefits of UTXOs – such as increased privacy, improved scalability, and transaction efficiency – with our smart contract capabilities, STRATO sets a new bar for asset tokenization.

Overcoming Limitations of Account-Based Models
It also addresses some of the inherent limitations of account-based models, such as the challenges around parallelizability and state management.

A Paradigm Shift in Decentralized RWA Management
This new standard is not only a technical leap forward but also introduces a paradigm shift in asset exchange and control, opening up a realm of possibilities for developers, asset managers, and users within the blockchain space.

Versatility for the Future of Decentralized RWA Applications
It's designed to support a vast array of applications, from simple transfers to complex, multi-faceted transactions, making it a versatile foundation for the future of decentralized asset management.

We achieve this comprehensive framework for managing Real Word Assets and their sales, and orders through three interconnected contracts: **Asset, Sale, and Order.**

Motivation:

Enhanced Asset Traceability and Ownership Clarity
The STRATO Mercata token standard is motivated by the need for a blockchain asset management system that provides absolute clarity in asset history and ownership — a feature that is crucial for high-value and unique digital assets but is often insufficient in current token standards.

Introducing Controlled and Secure Asset Transfer
Traditional token transfer mechanisms offer flexibility but lack stringent controls for complex transactions. STRATO's innovative transfer protocol, where assets are moved exclusively through a Sale contract, introduces a new layer of security and governance, ensuring that transfers are executed in line with pre-agreed rules.

Spurring Advanced Functionalities and Liquidity
By enabling a more sophisticated transaction structure, STRATO Mercata sets the stage for a wide array of financial activities on the blockchain, from simple asset exchanges to complex, rule-based transfers, enhancing liquidity and enabling new economic models.

Specification:

The `Asset` Contract
The Asset contract is responsible for managing the properties and ownership of digital assets.

Methods:

1. **Constructor**
   Initializes the contract when deployed.

```
constructor(
        string _name,
        string _description,
        string[] _images,
        string[] _files,
        uint _createdDate,
        uint _quantity
    )
```

   Sets up the asset with a name, description, associated images, files, creation date, and quantity. It also establishes the initial ownership and item number.

2. **attachSale**
   Links a Sale contract to the asset, enabling it to be sold.

```
function attachSale() public requireOwnerOrigin("attach sale")
```

   This function is called by the Sale contract and can only be executed by the owner of the asset. It ensures that only one sale can be attached at a time.

3. **closeSale**
   Detaches the Sale contract from the asset, ending the sale.

```
function closeSale() public fromSale("close sale")
```

   This function can be called by the attached Sale contract to signify the end of a sale. It sets the sale attribute to the null address.

4. **transferOwnership**
   Transfers ownership of a specified quantity of the asset to a new owner.

```
function transferOwnership(address _newOwner, uint _quantity, bool
_isUserTransfer, uint _transferNumber) public fromSale("transfer
ownership")
```

   Facilitates the transfer of assets, whether as a user-initiated transfer or through a sale. Emits the OwnershipTransfer and potentially the ItemTransfers events.

5. **automaticTransfer**
   Automatically transfers ownership when certain conditions within the Sale contract are met.

```
function automaticTransfer(address _newOwner, uint _quantity, uint
_transferNumber) public requireOwner("automatic transfer") returns
(uint)
```

This function is intended for use by the Sale contract to automate the transfer process under specific scenarios, like automated trading or bidding systems.

6. **updateAsset**
   Updates the asset's images and files.

```
function updateAsset(
       string[] _images,
       string[] _files
   )
```

   Allows the owner to update the asset's associated images and files, which might be necessary to reflect changes or additional information about the asset.

Events:

1. **OwnershipTransfer**

```
event OwnershipTransfer(
       address originAddress,
       address sellerAddress,
       string sellerCommonName,
       address purchaserAddress,
       string purchaserCommonName,
       uint minItemNumber,
       uint maxItemNumber
   );
```

The `Sale` Contract
The Sale contract facilitates the selling process of assets.

Methods:

1. **Constructor**
   Sets the asset to be sold, its price, the quantity available for sale, and an array of addresses for payment providers. It also links the Sale to the Asset by calling attachSale() on the Asset contract.

```
constructor(
        address _assetToBeSold,
        uint _price,
        uint _quantity,
        address[] _paymentProviders
    )
```

2. **changePrice**
   Allows the seller to adjust the price of the asset.

```
function changePrice(uint _price) public requireSeller("change
price")
```

3. **addPaymentProviders**

Expands the list of payment providers that can be used to purchase the asset.

```
function addPaymentProviders(address[] _paymentProviders) public
requireSeller("add payment providers")
```

4. **completeSale**

Finalizes the sale of the asset once payment is confirmed.

```
function completeSale() public requireSeller("complete sale")
returns (uint)
```

5. **automaticTransfer**

Automatically processes the transfer of the asset in response to triggers

```
function automaticTransfer(address _newOwner, uint _quantity, uint
_transferNumber) public returns (uint)
```

6. **closeSale**

Closes the sale if all assets have been sold or if the sale needs to be ended prematurely.

```
function closeSale() public fromSale("close sale")
```

The `Order` Contract

Manages the order lifecycle including creation, status updates, and cancellation.

Methods:

1. **constructor**

```
constructor(
        uint _orderId,
        address[] _saleAddresses,
        uint[] _quantities,
        uint _createdDate,
        uint _shippingAddressId,
        string _paymentSessionId,
        OrderStatus _status
    )
```

Sets up the order with an ID, associated sales, quantities of assets being ordered, creation date, shipping address ID, payment session ID, and initial status. It also calculates the total price and locks the quantity of assets from each sale.

2. **completeOrder**

```
function completeOrder(uint _fulfillmentDate, string
_comments) external
```

Finalizes the order once all sales are completed and payment is confirmed.

3. **updateOrderStatus**

```
function updateOrderStatus(OrderStatus _status) external
```

Changes the status of the order to reflect its progression through various stages.

4. **cancelOrder**

   Cancels the order and unlocks any associated sales.

```
function cancelOrder(string _comments) external
```

Enums:

1. **OrderStatus**

```
enum OrderStatus {
        NULL,
        AWAITING_FULFILLMENT,
        AWAITING_SHIPMENT,
        CLOSED,
        CANCELED,
        PAYMENT_PENDING,
        MAX
    }
```

Technical Implementation:
The implementation details are encapsulated in the provided SolidVM/Solidity code, which defines the structure and functionality of the Asset, Sale, and Order contracts.

Conclusion:

The introduction of the STRATO Mercata token standards mark a pivotal evolution in blockchain technology. With its innovative approach to RWA management, STRATO Mercata not only challenges the status quo but also redefines what's possible in the realm of digital assets. By seamlessly integrating the strengths of the UTXO model into our rich smart contract environment, we offer unparalleled security, flexibility, and scalability.

This standard is more than just an incremental improvement; it is a leap forward that unlocks new possibilities for RWA tokenization and exchange. STRATO Mercata is poised to become the backbone of a new generation of RWA based blockchain applications, catalyzing development and adoption in sectors craving for a more sophisticated and refined RWA management solution.

As we stand on the cusp of this new era, STRATO Mercata is set to ignite the transformative potential of RWAs across industries and applications, making it not just groundbreaking, but a true cornerstone of blockchain innovation.