ERC-1155 Metatokens

Ben Goetz bgoetz@bittrees.io bittrees.org

Abstract

Metatokens are first-class extensions to ERC-1155 contracts that allow for enhancements or restrictions to minting, burning, or transferring the underlying NFTs without requiring external oracles, intermediary transactions, or unsafe external calls[1]. They can be constructed to fit a particular purpose or simply to provide constraints on the underlying token, whether or not there are any metatokens actually minted. They can also describe the contract as a whole, rather than an individual token, and can be NFTs themselves. This eliminates intermediaries while ensuring the holder of an NFT is also its true owner.

Non-fungible Tokens

Fungible tokens are interchangeable with each other, without a unique identifying characteristic. They can be used for alternate currencies, reward tickets, vote counting, or other systems where quantity is necessary and identity isn't. Non-fungible tokens (NFTs) provide on-chain proof of ownership and are uniquely identifiable. They can be self-contained assets or representations of an off-chain asset (either digital or physical).

There are two main standards for fungible tokens: ERC-20 and ERC-777. ERC-20 contracts track the total supply of all minted tokens as well as the balances of individual holders. They can optionally provide a name or symbol[2], but do not provide any mechanism for off-chain metadata. Any references to ERC-20 contracts in this document can also be applied to ERC-777 contracts as we consider them to be functionally the same for the purposes of fungibility.

The first, and most common, NFT standard is ERC-721. It describes unique (one-of-one) NFTs, as well as a reference to off-chain metadata for each token. This metadata can additionally be immutable, as is common for profile picture NFTs stored on IPFS. Mutable off-chain metadata may be appropriate when the value or features of an NFT evolves over time. Not all ERC-721 contracts support metadata, but those that do must also provide a name and symbol as well as the off-chain URL. ERC-721 contacts track the individual owners of a token, but are not required to track the total supply of minted tokens.

ERC-1155 tokens can either be fungible or non-fungible, and this can vary depending on which token is being generated. Non-unique tokens can still be considered NFTs, however while ERC-721 contracts generate one-of-one tokens, ERC-1155 contracts

---

[1] `delegatecall` calls are used between the ERC-1155M contract and any registered metatokens, however this functionality is safe so long as the metatoken does not also make use of `delegatecall`.

[2] Analogous to ticker symbols used in financial markets; e.g., "WBTC" for the token representing Bitcoin on the Ethereum blockchain.

generate one-of-many. This allows you to represent semi-fungible tokens, like issuances of a comic or prints of a record. The total supply in each case is still limited. In this way, ERC-1155 tokens can be viewed as a combination of ERC-20 and ERC-721 tokens.

ERC-1155 contracts track token balances of individual owners; however, there is no mechanism for listing all owners of a token, the total supply of any individual token, nor the total supply of all tokens or issuances. Metadata is optionally provided with a per-token URL, as with ERC-721 contracts, but there is no concept of name or symbol for tokens.

For all token standards, regardless of how their supplies and balances may be queried on-chain, off-chain tracking is possible by indexing events emitted from transactions or by customizing the smart contract.

The ERC-1155 emulation of ERC-20 tokens is somewhat limited, as ERC-20 tokens can optionally provide a `decimals()` view that enables fixed-point expressions of their amounts; common values are 8 decimal places, as with Bitcoin and its ERC-20 token representations, and 18 decimal places, as with Ethereum. This limits how ERC-1155 tokens can interact with token trackers and liquidity pools. The ERC-20 tokens would also have unique addresses, while multiple ERC-1155 tokens on the same contract would share the same address, which inhibits how third parties can identify and separate out the tokens.

There is no mechanism for auctions, fractionalization, swaps, or other manipulations of the NFTs within any standard. Many solutions have arisen to address some of these problems, but there is no unifying answer that is applicable for all NFT contracts; part of that is due to the varying needs of a given project – NFTs that represent event passes benefit more from auctions than fractionalization. The varied approaches to these solutions prevent standardization around integrating third-party platforms or contracts.

For example, extracting liquidity from an NFT can be done with wrapping, vaulting, bundling, or fractionalizing the NFT (or a collection of NFTs); for each of these the end result is the same: making NFTs fungible. The core requirement of these is to lock the mutability of an NFT (minting, burning, or transferring) until some set of requirements is met. For fractionalization, this may be burning 80% of the fractionalized tokens in exchange for ownership of the NFT. In every case, the NFT must be held in escrow under the liquidity contract until the release requirement is met; this requires at least two transactions, possibly more[3].

The principle concept of NFTs is that the holder of the token is the owner of the token, with all rights reserved. Transferring a token to a third party, even temporarily, breaks this assertion and increases the risks when interacting with any third party protocol. Any benefit gained comes with an explicit reduction in ownership of the token.

Metatoken Extensions

Since ERC-1155 tokens can have variable supplies within the same contract, it's possible to group them according to their individual purpose. Metatokens expand this conceptually by creating wholly distinct groups of tokens, each backed by their own smart contract (see figure 1). Metatoken extensions allow users to retain verifiable ownership of an NFT while also enabling additional features or functionality that historically is reserved to centralized third parties.

All existing ERC-1155 contracts can be upgraded in place to support metatokens. ERC-721 contracts can also be upgraded in place, so long as their state is converted to match the ERC-1155 standard[4]. There is no limit to the number of extensions that can be registered; extensions may also be registered to any number of ERC-1155 contracts. We describe Metatoken supporting ERC-1155 contracts as "ERC-1155M".

---

[3] Transactions for creating the liquidity source (either by deploying a new contract or via an isolated source within a general liquidity contract), approving transfers to the liquidity source, and pulling from the NFT contract to the source, with possible additional transactions for configuration. This could be reduced to a single transaction by using `safeTransferFrom(... bytes)`, however this approach is high-risk and is not taken with current solutions which reduces its effectiveness.

[4] This may be impossible or impractical, depending on how much custom functionality is included in the contract. A multi-step upgrade plan may be necessary.

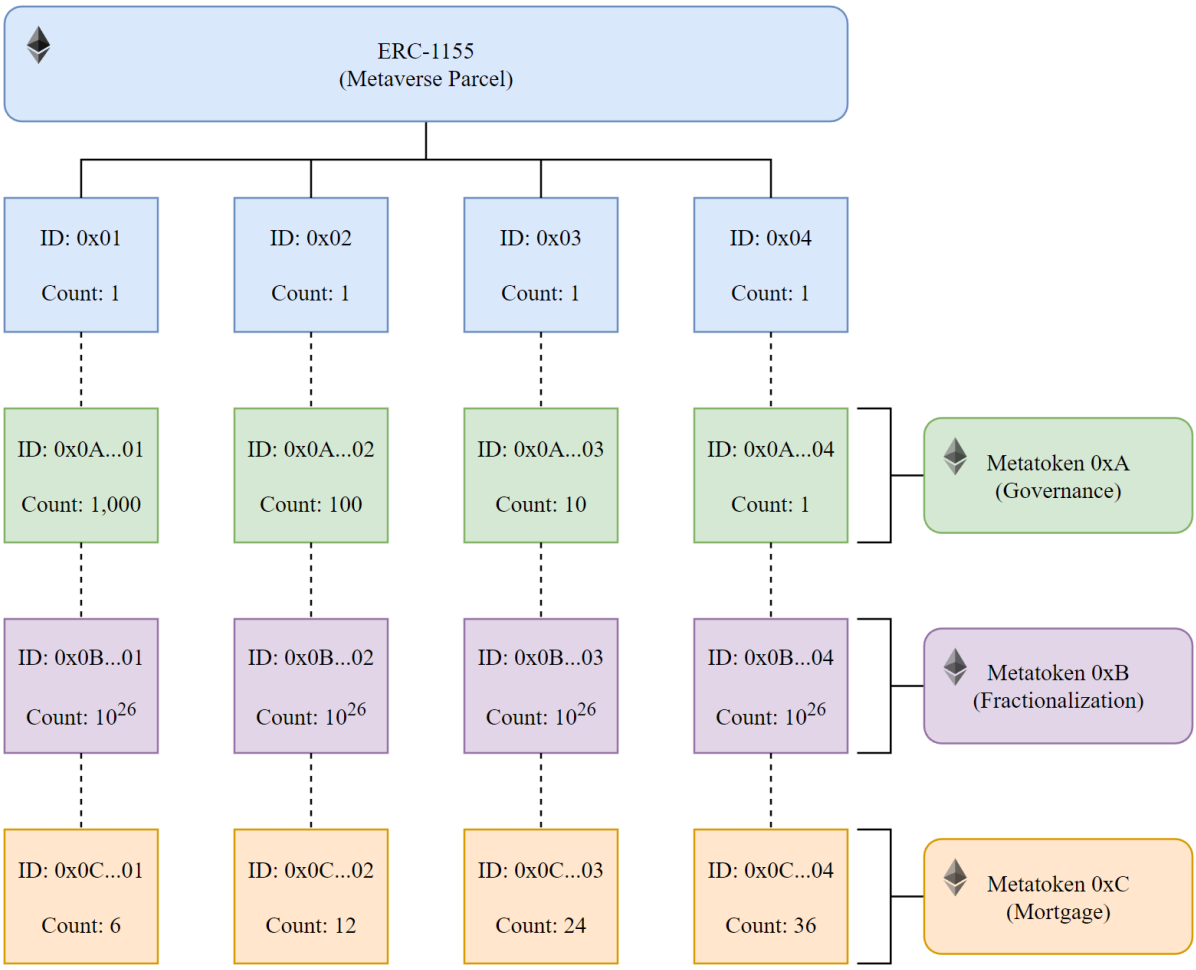

Figure 1

Relationship between ERC-1155 contract and its metatoken extension contracts.

Metatoken extensions act as nested ERC-1155 contracts that exist to support and enhance the base tokens. Extensions can be written to support current or future protocol standards, without requiring significant changes to the base contract. Extensions can issue a single metatoken that acts as an aggregate of all NFTs, or multiple metatokens that are matched one-to-one with base tokens. Metatokens are issued under the same base contract, for proof of authority, while offloading complex logic to external and standardizable contracts. They can be one-of-one NFTs, one-of-many NFTs, emulators of other token standards, and more.

Extension metatokens are fully valid ERC-1155 tokens with their own configurable supplies, restrictions, and metadata. They can be integrated into third party platforms and contracts without modification. Each token in the base ERC-1155M contract has a corresponding metatoken for each registered extension contract. These metatokens can add restrictions to token mutability – minting, burning, and transferring – as well as access and modify the state of the underlying contract as necessary.

Using external metatoken contracts does increase the complexity of deployments and the potential for vulnerabilities due to the use of `delegatecall`, however with careful planning and auditing this can be managed without much difficulty – automated deployment of all contracts is recommended to reduce the introduction of human error.

Metatoken extensions must be registered directly, and not through a proxy, as the chained `delegatecall` calls will prevent accurate resolution of the final extension contract. To that end, the ERC-1155M standard introduces a mechanism to upgrade previously registered extensions. However, upgrading is optional, and should

be done with care to ensure that the replacement extension is both valid within the context of its metatoken, as well as the base contract, with special care given to the storage layout.

The registration process is one-way and irreversible; once an extension has been registered against an ERC-1155M contract, it can only be disabled, otherwise any stateful changes it has made to storage could carry over to its non-conforming replacements. Re-enabling extensions should be considered carefully, as any transactions after its disablement would not be checked against its constraints. Tracking the total supply[5] of minted base tokens and metatokens is recommended to enable finer control over how the extensions apply constraints to the base tokens, however it is not strictly necessary.

ERC-1155M contracts provide hooks into the mutability actions of ERC-1155 tokens. Each extension can be registered against any number of actions, and must provide appropriate external functions for these hooks. The ERC-1155M contract provides two hooks for each action: pre-action and post-action. This results in a total of 6 groups and 12 hooks for extensions to implement (see figure 2); however, they must implement both the pre-action and post-action hooks for the specific action they are registered for, with a minimum of one action per extension[6]. Hooks may revert if applicable, but the pre-action hooks' mutability must be either `view` or `pure`.

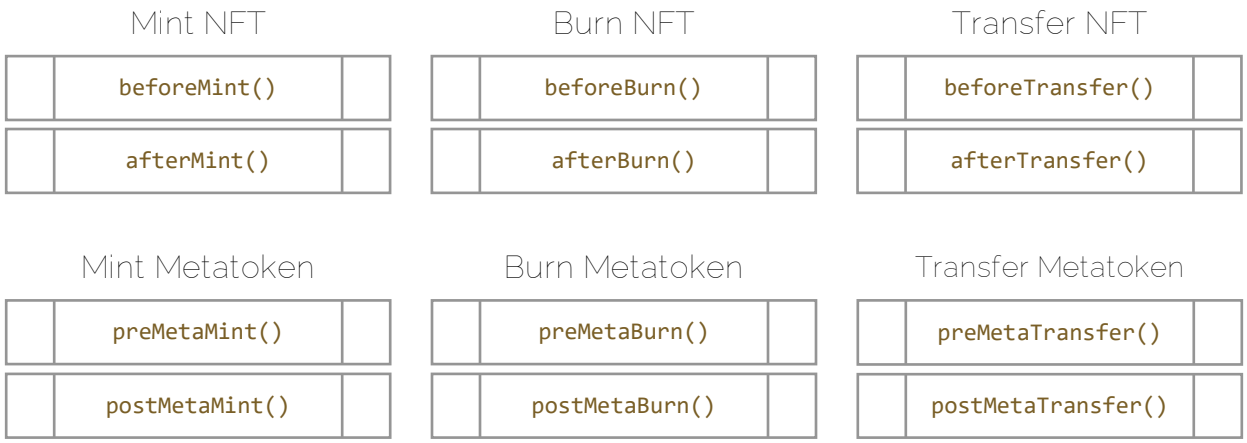

Figure 2
The metatoken extension hooks.

As both the base ERC-1155M token and the extension metatokens share the same token space, we differentiate between the two by prefixing the metatokens with their extension contract's address. It is thus trivial to identify which functionality is needed for a given action: if the extension address is zero, then treat it as an NFT and apply any necessary hooks; otherwise, treat it as a metatoken and delegate execution to the relevant extension.

Figure 3 shows the process for minting, but the process is the same for burning and transferring. First the extension address is extracted from the token ID; if this is zero, then it is an ERC-1155M NFT, if this is non-zero, then it is a metatoken. When minting an NFT, all registered extensions that hook into NFT minting are passed the full calldata for the batch-mint as a single-mint. There is no guarantee of execution order for the set of

[5] For an example, see OpenZeppelin's implementation at https://github.com/OpenZeppelin/openzeppelin-contracts/blob/docs-v4.x/contracts/token/ERC1155/extensions/ERC1155Supply.sol.

[6] See Appendix A for the `IMetatoken1155` interface.

registered extensions, however all pre-mint hooks will be executed prior to the mint, and all post-mint hooks will be executed before finishing the mint. When minting a metatoken, only the hooks for that metatoken's extension are executed.

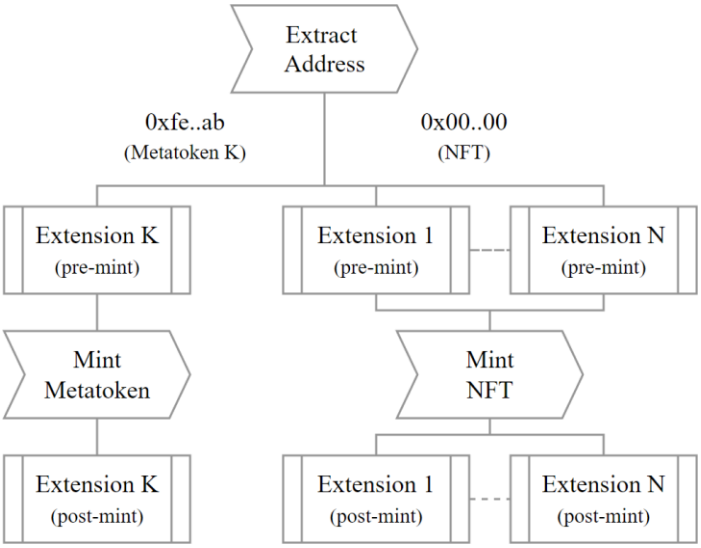

Figure 3
The process for minting tokens.

A `delegatecall` is made to the extension contract so it has access to the ERC-1155M's storage; additionally, each metatoken has semi-exclusive access to 32 KiB of storage[7]. There is a potential for security vulnerabilities if an assumption is made that only the metatoken has read / write access to their individual exclusive zones as both the contract storage itself is insecure via off-chain inspection and individual slots can be accessed trivially. It is recommended, to allocate such storage to metatokens on a case-by-case basis. This does not cause a practical increase to gas costs, and so long as the offsets are not changed or removed from upgraded contracts, or a duplicate offset is used in a separate contract, the storage will remain exclusive. Proxying Extension Calls

Simple metatoken extensions only need to register hooks with the core contract, however more complex ones may need to expose their own functions or views. For example, a metatoken extension that keeps an on-chain record of all historical owners of a token needs to be able to expose a view that will allow for querying the historical record (see figure 4).

```
afterBurn(...)
afterMint(...)
afterTransfer(...)
clearPreviousHolders()
getPreviousHolders(uint id)
```

Figure 4
ABI excerpt for on-chain historical token owner extension.

This example contract could also expose a function that will clear the historical record for a specific token on-demand, so long as the current owner of the token executes the transaction. This extension would use the metatoken hooks to keep the historical record up to date for each token automatically.

It is trivial to proxy non-view calls in the same way that metatoken hooks are proxied to extensions. ERC-1155M contracts contain a `proxyMetatokenCall()` function that will generate an arbitrary `delegatecall` to any previously-registered metatoken extension. This call requires allowlisting the selector of the function that will be proxied[8]. Great care must be taken when identifying which functions are allowed to be proxied to reduce the possibility of significant security vulnerabilities.

However, in the case of views, it is not possible to directly execute a `delegatecall` within a `staticcall`, as the solidity compiler cannot guarantee the call will remain static after control has passed to the delegate contract.

[7] This is done by creating a `uint256[0xFFFF * K]` state variable, where `K` is a unique offset. Exclusivity requires other extensions (or the base contract) to neither use direct access for a specific slot nor use overlapping offsets to access restricted space.

[8] The management of allowed proxy selectors is not enabled by default.

The solution we propose is to `delegatecall` a `staticcall` (see figure 5). This enables read-only validation of the `staticcall` while also enabling arbitrary extension logic via the `delegatecall`. While this is beneficial for off-chain lookups (in such cases where an indexer is nonapplicable), it does increase the cost and complexity for on-chain view calls. Currently, Solidity does not provide useful cross-contract error handling, so debugging and transaction rejection analysis are significantly worse. There are also valid security concerns with exposing arbitrary endpoints, even if the base contract and all metatokens are fully owned and developed by a single entity.

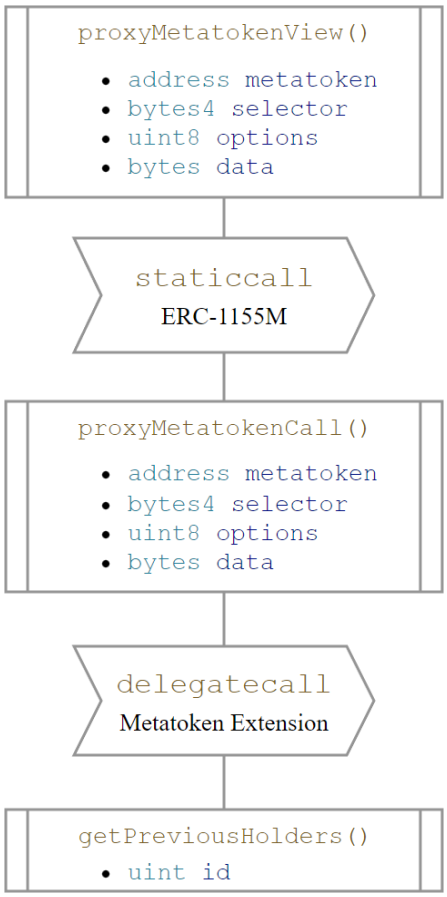

Figure 5
Method for proxied, delegated view calls.

The ERC-1155M contract therefore provides opt-in only access to proxied calls (view or otherwise). Each specific combination of (extension, selector) must be registered to enable the execution of proxied calls.

Examples

Fractionalization

First-class fractionalization of an NFT is now possible with a single metatoken. The extension would provide limitations on when the NFT can be transferred based on the outstanding balance of minted metatokens. You can even have the metatoken mimic an ERC-20 token for external liquidity pools and AMMs. As long as it implements a transformer[9] from ERC-20 to ERC-1155, external contracts and dapps will function as normal.

Protocol Emulation

Metatokens can emulate other protocols or standards. ERC-721 and ERC-20 tokens are especially trivial as they share much of the same ABI and storage architectures as ERC-1155 tokens. These emulation layers must take advantage of proxied extension calls, but allow for explicit expressivity and backwards compatibility. A single ERC-1155M token can generate multiple distinct ERC-20 tokens for any purpose.

Traditional DAOs use a dedicated governance token for consensus. With Metatokens, any ERC-1155M contract can have a dedicated ERC-20 or ERC-721 compatible token for simple governance, or multiple tokens for tiered governance solutions.

Music Label

Take a contract that represents an independent music label, with each token associated with an individual track. It issues one-of-one NFTs that provide full copyright ownership over the master track and one-of-many NFTs for individual purchasers of the track. Future NFTs could also be used for live event access passes or governance tokens for an artist DAO. However, external

[9] The only new functionality the metatoken would need to provide beyond mapping ERC-20 functions to ERC-1155 ones, is to allow for variable approvals of allowances for third-party transfers. It would also need to know the address of the ERC-1155M contract (or its proxy). `Transfer()` events would also need to be emitted in `postMetaBurn()`, `postMetaMint()`, and `postMetaTransfer()`.

contracts must be intimately familiar with the design of the contract to be able to identify the purpose of each NFT.

The music label could issue a metatoken that represents access to a live event. These could be available at a reduced price for previous token holders, free for master token holders, and within a specific minting window for everyone else. Handling this complex behavior on-chain eliminates risky signature validation or burdensome integrations.

WiV Technology[10]

WiV is a web3 investment company turning wine into financial assets. They facilitate buying and selling wines on the blockchain using NFT technology. Their NFTs represent underlying wine assets that are stored in a centralized warehouse under optimal conditions. Their NFT Wine assets can be bought and sold on decentralized marketplaces. $WIVA is the governance and utility token for the entire WiV protocol and ecosystem.

WiV Wine NFTs are the base asset in the $WIVA ecosystem. The $WIVX protocol, using NFTwrapper, is the liquidity vehicle for the Wine assets. Wine NFT owners will be able to deposit wine NFTs for a yield, take out a loan on their wine as collateral, and sell their wine to the protocol for the market value. Depositors will receive $WIVX tokens in return for their collateral and will be able to apply these tokens to many different defi applications.

The Wine NFTs are managed by an ERC-1155M contract; each NFT represents a specific bottle or case of a specific vintage of wine. Each physical asset is matched one-to-one with a single issuance of a token. NFTs are only minted when the physical wine has been properly stored, and are only burned when the WiV protocol has sold the physical wine.

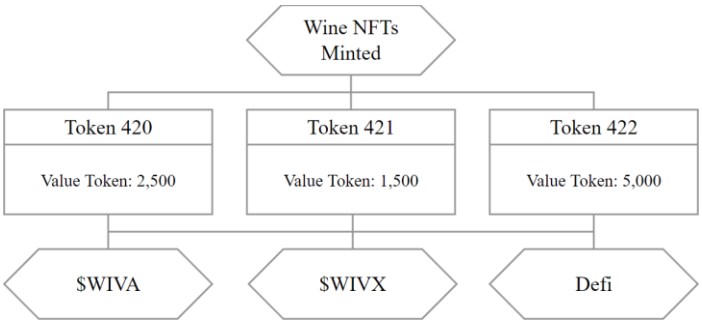

Figure 6

The lifecycle of Wine NFTs within the $WIVA ecosystem.

WiV uses Value metatoken that is minted when the NFTs are minted, representing the initial purchase price in US$. As the value of the wine matures over time, the fair market value of these Value tokens also appreciates. These Value metatokens are stored in a custom NFTwrapper vault which manages the $WIVX token. Thus, as assets are acquired and sold, the value of the $WIVX token is stably backed by the total combined value of all wine assets in the ecosystem.

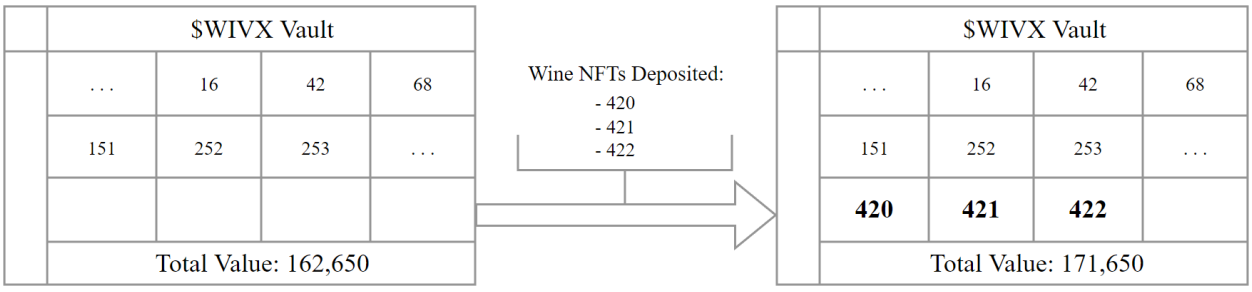

Figure 7

The relationship between the value metatoken and $WIVX.

[10] https://wiv.io/

Appendix A – IMetatoken1155

```
// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

/**
 * @dev A metatoken is an extension of metadata and logic on top of an ERC1155 NFT.
 *
 * There are 256 partitions in the token ID space. The first is used for the NFTs,
 * while the rest are used for the metatokens. The first byte (highest-order, big-
 * endian) of the ID is used to distinguish between NFTs and metatokens; with each
 * category of metatokens being an additive offset of the NFT ID. This limits the
 * maximum NFT ID to be 2^248-1.
 *
 * Libraries that implement metatokens will be trustfully registered to ERC1155 NFT
 * contracts.
 *
 * To reduce unintentional confusion between interacting with the root NFT and its
 * metatokens, the naming of the hooks differs slightly: before/after is used when
 * writing NFT logic, pre/post is used when writing metatoken logic.
 */
interface IMetatoken1155 is IERC165 {
    // Metatoken Registration Details //

    /**
     * @dev Which category (and slot) this metatoken belongs to. This value is used
     * to determine shifts and masks when converting an NFT to its metatoken and
     * back.
     *
     * Requirements:
     * - Must return a value between 1 and 255 (inclusive).
     */
    function metatokenCategory() external pure returns (uint8);

    // NFT - Precheck Hooks //

    /**
     * @dev Called prior to the burn of the root NFT.
     *
     * This should not modify state as it is used solely as a test for invariance
     * prior to the burning of an NFT.
     *
     * Requirements:
     * - `from` cannot be the zero address.
     * - `from` must have at least `amount` tokens of token type `id`.
     *
     * Example: Checking to make sure the metatoken exists before burning it.
     */
    function beforeBurn(
        address from,
        uint256 id,
        uint256 amount
    ) external view;

    /**
     * @dev Called prior to the mint of the root NFT.
     *
     * This should not modify state as it is used solely as a test for invariance
     * prior to the minting of a NFT.
     *
     * Requirements:
     * - `to` cannot be the zero address.
     * - If `to` refers to a smart contract, it must implement {IERC1155Receiver-
     * onERC1155Received} and return the
     * acceptance magic value.
     *
     * Example: Checking to make sure the metatoken does not exist before minting it.
     */
    function beforeMint(
        address to,
        uint256 id,
        uint256 amount,
        bytes memory data
    ) external view;

    /**
     * @dev Called prior to the transfer of the root NFT.
     *
     * This should not modify state as it is used solely as a test for invariance
     * prior to the transferring of a NFT.
     *
     * Example: Checking to make sure the metatoken has the correct amount before
     * transferring it.
     */
    function beforeTransfer(
        address from,
```

```
        address to,
        uint256 id,
        uint256 amount,
        bytes memory data
    ) external view;

    // NFT - Postaction Hooks //

    /**
     * @dev Called after the burn of the root NFT.
     *
     * This may modify state if necessary, however it must also test for invariances
     * after the burning of an NFT.
     *
     * Requirements:
     * - `from` cannot be the zero address.
     * - `from` must have at least `amount` tokens of token type `id`.
     *
     * Example: Checking to make sure secondary addresses associated with the
     * metatoken are cleared.
     */
    function afterBurn(
        address from,
        uint256 id,
        uint256 amount
    ) external;

    /**
     * @dev Called after the mint of the root NFT.
     *
     * This may modify state if necessary, however it must also test for invariances
     * after the minting of an NFT.
     *
     * Requirements:
     * - `to` cannot be the zero address.
     * - If `to` refers to a smart contract, it must implement {IERC1155Receiver-
     * onERC1155Received} and return the
     * acceptance magic value.
     *
     * Example: Checking to make sure secondary addresses associated with the
     * metatoken are set.
     */
    function afterMint(
        address to,
        uint256 id,
        uint256 amount,
        bytes memory data
    ) external;

    /**
     * @dev Called prior to the transfer of the root NFT.
     *
     * This may modify state if necessary, however it must also test for invariances
     * after the transferring of an NFT.
     *
     * Example: Checking to make sure secondary addresses associated with the
     * metatoken are updated.
     */
    function afterTransfer(
        address from,
        address to,
        uint256 id,
        uint256 amount,
        bytes memory data
    ) external;

    // Metatoken - Precheck Hooks //

    /**
     * @dev Called prior to the burn of the metatoken.
     *
     * This should not modify state as it is used solely as a test for invariance
     * prior to the burning of a metatoken.
     *
     * Requirements:
     * - `from` cannot be the zero address.
     * - `from` must have at least `amount` tokens of token type `id`.
     *
     * Example: Checking to make sure the metatoken exists before burning it.
     */
    function preMetaBurn(
        address from,
        uint256 id,
        uint256 amount
    ) external view;

    /**
     * @dev Called prior to the mint of the metatoken.
     *
     * This should not modify state as it is used solely as a test for invariance
     * prior to the minting of a metatoken.
     *
```

```
     * Requirements:
     * - `to` cannot be the zero address.
     * - If `to` refers to a smart contract, it must implement {IERC1155Receiver-
     * onERC1155Received} and return the
     * acceptance magic value.
     *
     * Example: Checking to make sure the metatoken does not exist before minting it.
     */
    function preMetaMint(
        address to,
        uint256 id,
        uint256 amount,
        bytes memory data
    ) external view;

    /**
     * @dev Called prior to the transfer of the metatoken.
     *
     * This should not modify state as it is used solely as a test for invariance
     * prior to the transferring  of a metatoken.
     *
     * Example: Checking to make sure the metatoken has the correct amount before
     * transferring it.
     */
    function preMetaTransfer(
        address from,
        address to,
        uint256 id,
        uint256 amount,
        bytes memory data
    ) external view;

    // Metatoken - Postaction Hooks //

    /**
     * @dev Called after the burn of the metatoken.
     *
     * This may modify state if necessary, however it must also test for invariances
     * after the burning of a metatoken.
     *
     * Requirements:
     * - `from` cannot be the zero address.
     * - `from` must have at least `amount` tokens of token type `id`.
     *
     * Example: Checking to make sure secondary addresses associated with the
     * metatoken are cleared.
     */
    function postMetaBurn(
        address from,
        uint256 id,
        uint256 amount
    ) external;

    /**
     * @dev Called after the mint of the metatoken.
     *
     * This may modify state if necessary, however it must also test for invariances
     * after the minting of a metatoken.
     *
     * Requirements:
     * - `to` cannot be the zero address.
     * - If `to` refers to a smart contract, it must implement {IERC1155Receiver-
     * onERC1155Received} and return the
     * acceptance magic value.
     *
     * Example: Checking to make sure secondary addresses associated with the
     * metatoken are set.
     */
    function postMetaMint(
        address to,
        uint256 id,
        uint256 amount,
        bytes memory data
    ) external;

    /**
     * @dev Called prior to the transfer of the metatoken.
     *
     * This may modify state if necessary, however it must also test for invariances
     * after the transferring of a metatoken.
     *
     * Example: Checking to make sure secondary addresses associated with the
     * metatoken are updated.
     */
    function postMetaTransfer(
        address from,
        address to,
        uint256 id,
        uint256 amount,
        bytes memory data
    ) external;
}
```