

Qumosi: A Plug-n-play Open-Market Protocol

Amir Khashayar Mohammadi
akm@qumosi.com
qumosi.com

Eduardo Fonseca
svn@qumosi.com
qumosi.com

Abstract. A set of smart contracts that enables an open market protocol for transferring digital assets¹ without needing pre-approvals from third parties, designed to deter spam and low quality content. The proposed ecosystem allows for the utilization of ERC-721 forks while abiding by the rules of a single ERC-20 smart contract, both interoperable. The two distinguished oracles are referred to as merchants² (QMSI-721) and the bank (QMSI-20). The bank and merchants both have rules in place that govern each other in a one-to-many relationship. The bank's rules supersede the merchant rules, however they are immutable unlike traditional banking and platform moderation rules. The bank maintains a max supply of ERC-20 tokens while dispensing them freely from an equal opportunity faucet system; designed to meet its eventual demise through halving events and time based constraints directly tied to its usage frequency. The bank maintains protocols to keep the system self-sustainable via burning and staking existing tokens in later stages. Bank token burning occurs dynamically each time a new entry is minted on some but not all associated merchant contracts; this is to deter spam and improve content quality. Staking involves freezing existing bank tokens against a fixed rate influenced by the time tokens are locked for; rewards only come from existing burned tokens. Bank tokens can be used within the built-in escrow system for transferring ownership of ledger entries, giving creators incentives to mint quality content in order to persuade. Merchants control the price of minting a new ledger entry (amount to be burned), which is influenced by a burn rate enforced by the bank (percent based on circulating supply of bank tokens). Merchants also control commission rate, set by the creator, to be paid to the creator of a ledger entry each time it is traded, enforced by the bank. Merchant related controls can be freely manipulated or omitted without requiring the bank to act or approve changes.

¹ *Digital assets* in this context is any media type that can have ownership changes of the same content.

² The term *market* and *merchant* are used interchangeably.

1. Introduction

Media platforms on the Internet have become reliant on moderation to reduce spam and low quality content. As the size of information transmitted and saved grows, and spam becomes indistinguishable from normal content, traditional means of moderation become obsolete. Impersonation problems further accelerate this issue, especially when the same cosmetic profile identifiers such as verification symbols are offered to bad actors. An asymmetric keypair system that allows digitally signing content on a public ledger system may help with impersonation via unique identifiers and cryptographic association, but not spam³. An induced fee could help segment the visibility of spam per new posts on this ledger; assuming the bots⁴ are not at a financial advantage when fees are too high due to network congestion. We propose a dynamic fee based on activity. Enforcing such a system while retaining the availability and the ability to remain interoperable in an open fashion requires the use of a distribution network that is decentralized to remain scalable and immutable to dissuade rule bending.

Naturally, blockchain based ledger systems are established as immutable and decentralized. Its usage can help prevent modification and censorship from bad moderation practices. In the modern day, a blog exchanging knowledge for monetary gain through an advertising network could find its creation censored through content guidelines designed to save face for the advertising vendor. Preserving the original content in some contexts can be seen as beneficial but the gain of any commodity should not influence it, in a perfect world these two concepts are independent and isolated from each other. The digital asset in our context is served “as is” with a safe channel to harbor its visibility and integrity, as a result of the underlying system⁵ it resides on.

This hypothetical system explores to solve the age long problem of low quality content and spam on an immutable ledger system designed to store metadata and ownership while preserving item integrity. It uses a template spend-ERC20-create-ERC721⁶ model in a manner that is dynamic in influencing one action⁷ over another to preserve⁸ itself.

Spend⁹ ERC-20 to create¹⁰ ERC-721. If there is too much burning of ERC-20 tokens, the system increases the reward for staking. If there is too much staking, then there are less rewards to claim. Stake rewards are burned tokens, and the only way to replenish the pool of rewards is to spend tokens on other ERC-721 contracts.

To reduce network spam, a *burnRate* is applied, a dynamic tax on all mints across all ERC-721 constructs the ERC-20 contract interfaces with. This *burnRate* is influenced by the supply of tokens that are neither in burned or staked status, but in circulation. Detering spam via a payment model that is influenced by network activity is a prominent method given current day advancements in captcha solving capabilities and other validation bypassing mechanisms. Users with the ability to transact with such a system have gone through an on-ramp that most likely implements a know-your-customer component.

³ *Spam* in this context is repetitive material designed to take over visibility and time from other streams of content.

⁴ *Bots* in this context are a high volume of agents or nodes in the network that act upon a master's will with the purpose of influencing.

⁵ The *underlying system* is a non-upgradable smart contract using the Solidity language while residing on the Ethereum network.

⁶ William Entriken, <https://fulldecent.github.io/spend-ERC20-create-ERC721/>, MIT License

⁷ The *actions* that can occur is the act of spending and creating.

⁸ To *preserve* in this context is to ensure that the majority controls the sequence of events that occur and influence the entire governing body based on rules that don't fundamentally change.

⁹ The term *spend* and *burn* are used interchangeably.

¹⁰ The term *create* and *mint* are used interchangeably.

Minted new content can be prioritized over those that are not, thus allowing for easier ways to showcase real users and their minted content over spam or low quality content. It's important to note, not all content in a system has to be minted. A hypothetical social network could persuade users to mint posts that want high visibility, while associated comments beneath it, like typical thread-like structures, do not; since visibility isn't as important for minor things like reactions.

The following is a diagram that shows the relationship between these smart contracts. It is meant to showcase how different minting fees may coexist in this model alongside a universal burn fee. It also highlights the different controls that are maintaining the mutual one-to-many relationship.

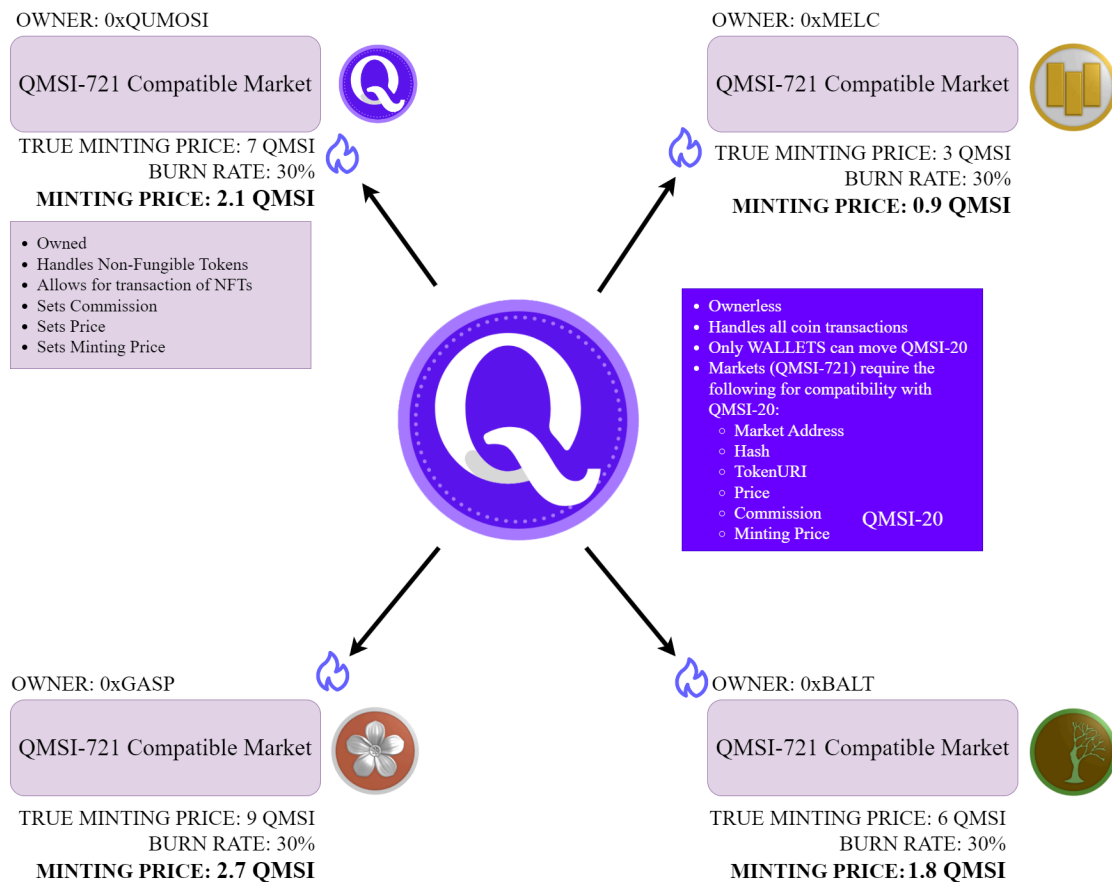


Figure 1. Qumosi Cross-Market Compatibility Requirements

There are two smart contract interfaces in this ecosystem. QMSI-ERC20 (or QMSI-20), the bank, the main contract that is responsible for ensuring the integrity and enforcing the base layer rules when transacting with QMSI-ERC721 (or QMSI-721), the merchant. The relationship between the two involves one QMSI-20 contract interacting with many QMSI-721 contracts in such a manner that does not require pre-approvals for the QMSI-721 counterpart to facilitate a contract interaction.

2. QMSI-20: The Bank

Maximum Supply and the Faucet

The maximum supply of QMSI-20 tokens that can ever be drained from the faucet is 37,000,000. This number has no mathematical importance, it was arbitrarily chosen, and other forks of QMSI-20 may choose different values. Current implementation keeps this value immutable and hardcoded in the smart contract. The maximum supply is not the same number of tokens that this smart contract starts with. There is no pre-mint, all coins created come from the faucet in a manner that is equal to all its consumers; this is verifiable given each time the faucet is used an event is emitted. Equality is defined as being able to execute the same function just as anyone else using the Ethereum mainnet, with the added assumption of the absence of adversarial network advantages such as MEV-style interactions.

The maximum supply of total tokens to exist does not guarantee existence. Depending on how the faucet is used throughout the lifetime of the QMSI-20 smart contract determines how many, up to the limit, will truly exist.

The best way to describe the faucet in this network is to use its analogy. Picture a real faucet with water. Each time someone drinks from the faucet, there is less water for the next person in line. Fear not, everyday there is rain that replenishes the faucet's reserves. However, even rainfall can be constrained if people keep drinking from the faucet because the act dislocates water. Every four years of *active* faucet usage, the rainfall is divided by two.

Conceptually the faucet will eventually run dry, while the overall system then transitions to using the burn and stake mechanisms exclusively to continue the flow of water. The faucet's purpose is to diversify the ownership of QMSI-20 tokens so that the majority of the total supply is not heavily concentrated in one controller, as a majority stake can allow for 51% style attacks on the staking mechanism, where one bad actor is able to constantly drain the burn pool in shorter lock times. Thus having a mechanism that dispenses QMSI-20 tokens in an equal but competitive manner within the initial phase is vital. These phases are inspired by the Ethereum transition from Proof-of-Work to Proof-of-Stake, the former was a necessity for the latter to succeed with less centralization, better distribution and security.

Halvings Events and Negative Decay

The faucet is influenced by time and chain of events. Time is equal to the number of days the faucet was in use. If seven days have passed and the faucet was only used for three of those days, then time is three days of total usage. The halving is hardcoded to *1,460 days*, this number is 4×365 , four years.

For a halving event to occur, four years of active faucet usage needs to be reached. The number of days the faucet is used is tracked in *daysConsumed*. When a halving event occurs, the number stored in *dailyClaimLimit* is divided by two and *daysConsumed* is reset to 0.

When the smart contract is first deployed, *dailyClaimLimit* starts at 3,700 coins. This value is the total amount of coins that can be drained from the faucet on a given day, before it is formally influenced by halving events. The number of total coins drained in one day is stored in *tokensClaimedToday*, a variable that is reset each day.

The first to drink from the faucet on a given day is rewarded 1% of *dailyClaimLimit*, stored in *rewardPerClaim*. Each time the faucet is used, *rewardPerClaim* value is adjusted following a negative decay formula. The following showcases a summation of the negative decay:

$$\sum_{n=1}^{37} e^{-\frac{1}{100} \cdot x} \quad \text{Eq (1)}$$

The following is equivalent Python code that simulates how rewards are drained from the faucet:

```
def calculate_reward(daily_limit):
    click = 0
    reward = 0.01 * daily_limit
    sum = reward
    while sum <= daily_limit:
        click += 1
        print(f"Clicks: {click}, Reward: {reward}, Sum: {sum}")
        reward = (reward * 99004983374916819303589981151435220778399087722496) / 1e50
        sum += reward
```

When executing this function against the same initial starting daily limit of 3,700:

```
>>> calculate_reward(3700)
Clicks: 1, Reward: 37.0, Sum: 37.0
Clicks: 2, Reward: 36.63184384871922, Sum: 73.63184384871923
Clicks: 3, Reward: 36.267350912349954, Sum: 109.89919476106918
Clicks: 4, Reward: 35.90648474129481, Sum: 145.805679502364
Clicks: 5, Reward: 35.54920924863597, Sum: 181.35488875099998
Clicks: 6, Reward: 35.195488706526426, Sum: 216.5503774575264
Clicks: 7, Reward: 34.84528774261721, Sum: 251.3956652001436
Clicks: 8, Reward: 34.498571336520094, Sum: 285.8942365366637
...
Clicks: 523, Reward: 0.2000711776783222, Sum: 3698.6235843772743
Clicks: 524, Reward: 0.19808043619842317, Sum: 3698.8216648134726
Clicks: 525, Reward: 0.19610950292721155, Sum: 3699.0177743164
Clicks: 526, Reward: 0.1941581807697178, Sum: 3699.2119324971695
Clicks: 527, Reward: 0.19222627459210004, Sum: 3699.4041587717616
Clicks: 528, Reward: 0.19031359120213057, Sum: 3699.5944723629636
Clicks: 529, Reward: 0.18841993932987652, Sum: 3699.7828923022935
Clicks: 530, Reward: 0.18654512960857259, Sum: 3699.969437431902
```

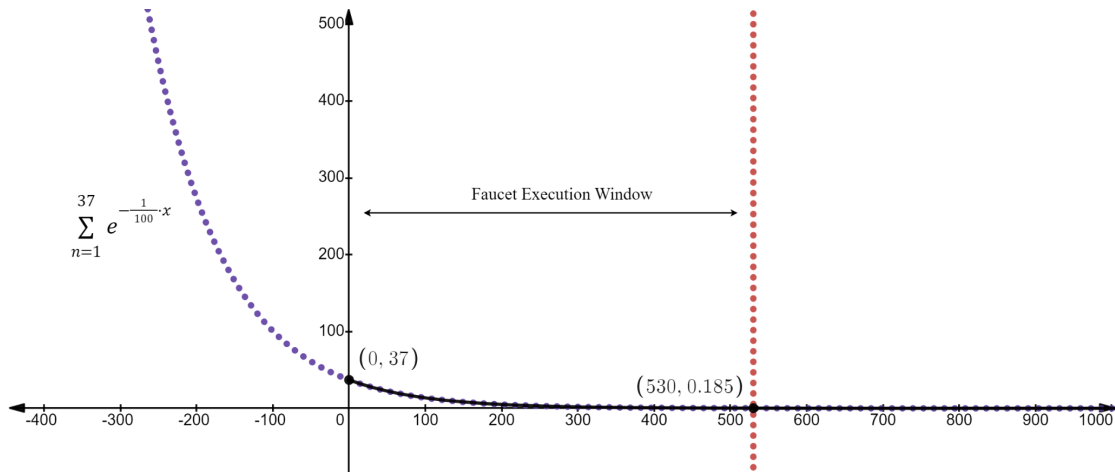


Figure 2. Faucet Execution Window Per Day

The output yields 530 clicks before reaching the limit. The number of clicks represents the total number of wallet users that may be rewarded by executing the *drinkFromFaucet* function. Each time the faucet function is “clicked”, the reward for the next user yields less coins until it is reset.

The value in *rewardPerClaim* is reset each day back to 1% of *dailyClaimLimit* which is divided by 2 per halving interval. Regardless of the daily limit value, given the negative decay curve, the number of daily unique faucet drinkers or clicks will always be 530 before the daily limit is reached. This can be tested by using different values as the parameter in the *calculate_reward()* Python function.

If we are to assume the faucet is fully utilized to its *dailyClaimLimit* each day we can assume the following effects of the halving events:

Year	Halving Interval	<i>dailyClaimLimit</i>	<i>daysConsumed</i>	<i>circulatingSupply</i>
2028	1	3,700	1,460	5,402,000
2032	2	1,850	2,920	8,103,000
2036	3	925	4,380	9,453,500
2040	4	462.5	5,840	10,128,750

Table 1. Example Halvings Calculations

*circulatingSupply*¹¹ is the total number of coins that have been minted using the faucet assuming maximum utilization. It’s important to understand these numbers do not reflect actual usage, if the faucet is used but not to its daily limit, then the remaining coins for that particular day are lost forever. An analogy to consider is having rainfall when all your water reserves are full. You still lose water in the process.

$$Circulating\ Supply = Total\ Supply + Staked\ Supply + Burned\ Supply \quad Eq\ (2)$$

¹¹ The function *circulatingSupply* is *totalSupply()* + *totalStaked()* + *burnPool()*, it accounts for all pools combined to get the real number of tokens regardless of what state they are in.

Both effects of the halvings and negative decay determines the fate of the faucet, it will eventually run dry leading to the drought event.

Drought Event

The drought event occurs when the faucet is no longer a viable source for new coins. Many events could have led to this event (including but not limited to) high Ethereum gas fees, poor rewards, max supply being reached. If we use the same analogy of water displacement as before, there is no natural rain in this phase but water continues to circulate.

There exist two token reserves¹² that allow for circulation even after the drought event, burn and stake pools. These pools complement each other by taking from the other in order to replenish itself. The only difference is how tokens are introduced to the pool and taken out.

Burn Pool

The burn pool's sole purpose is to collect coins that have been burned when minting new QMSI-721 certificates¹³. There exists a function called *burnRate()*, it returns an integer value between 0 and 100 representing the percentage fee to be applied over the minting price. Each QMSI-721 merchant smart contract can incorporate its own fee to mint a single or batch of certs. Some merchants may have high fees, some can be low, some can even be dynamic.

However, these fees are enforced at the QMSI-20 level, and the way they are applied is the following formula when calling the *CrossTokenCreate()* function:

```
uint256 burnValue = (mintingPrice*this.burnRate())/100;
```

$$\text{Burn Value} = \frac{\text{Minting Price} \times \text{Burn Rate}}{100} \quad \text{Eq (3)}$$

The *mintingPrice* is what the QMSI-721 returns via *QMSI721(market).trueMintingPrice()*; where *market* is the smart contract address of the merchant's store. The *burnRate*'s percentage is applied to the minting price. The *burnRate* is meant to be dynamic depending on the number of tokens that are liquid and isolated from the burn pool or stake pool. The *burnRate* is simply *totalSupply* divided by *maxSupply* (not to be confused with *circulatingSupply*). The *maxSupply* function is immutable and returns the number of possible tokens to exist regardless of all constraints that exist (such as faucet usage). In the Solidity programming language, we multiply it by 100 to get a integer based percentage of how much should be burned:

```
(totalSupply() * 100) / maxSupply();
```

The function that *CrossTokenCreate* uses to burn the tokens into the burn pool is *spend()*, a function that takes the number of *msg.sender*'s QMSI-20 tokens to burn.

¹² The term *reserve* and *pool* are used interchangeably.

¹³ The term *certificate* and *non-fungible tokens* also known as NFT are used interchangeably.

Stake Pool

The concept of staking QMSI-20 tokens works by locking them for a period of time. Locking means the set aside tokens cannot be moved or burned, in this state they are not part of the balance. The function *rewardsCalculator* can help estimate the potential reward that is to be obtained from locking tokens for a period of time. The formula for reward is the following:

```
uint256 reward = ((value_ * days_) / totalSupply());
```

$$\text{Stake Reward} = \frac{\text{No. of Tokens} \times \text{Days}}{\text{Total Supply}} \quad \text{Eq (4)}$$

Where *totalSupply* does not include burned or currently staked tokens, but the number of tokens that are liquid.

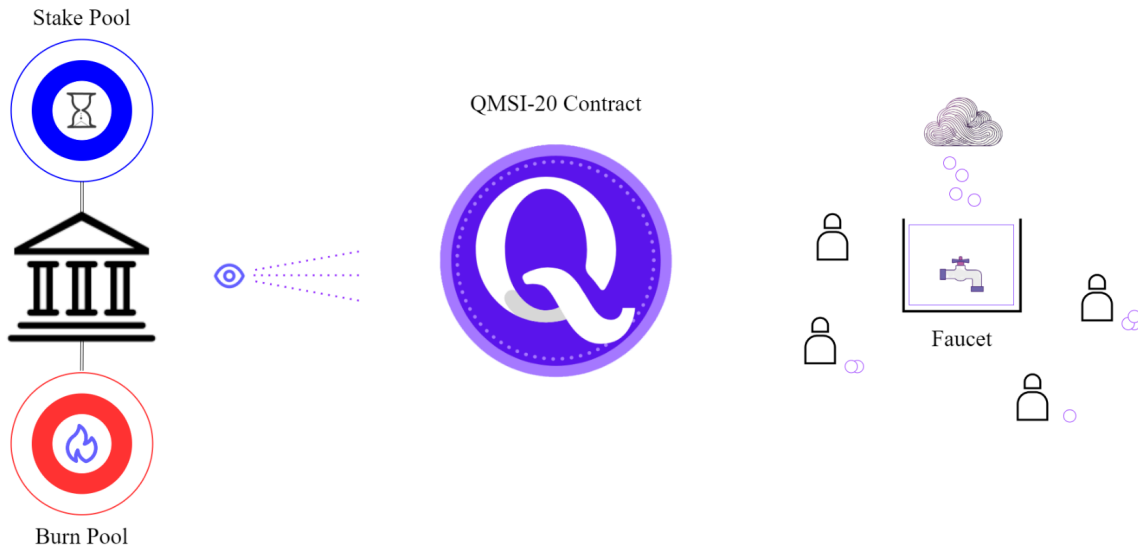


Figure 3. QMSI Ecosystem

The reward depletes the burn pool. In this sense, new tokens are not being created but reclaimed from a different state. This also means that if there are insufficient tokens in the burn pool, then there are no rewards to gain from staking. It is designed to be self-sufficient without the need to introduce new tokens while rewarding those who help slow the rate at which tokens are being burned.

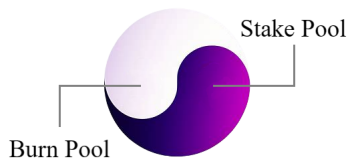


Figure 4. Ideal Token Distribution

3. QMSI-721: The Merchant

Enforcing Royalties

A useful analogy is to think of QMSI-20 as the bank, and QMSI-721 as the merchant. In order to utilize this system, you must abide by the rules of the bank while also abiding by the rules of the merchant. However, when it comes to the priority in which these rules should be followed, the bank has more power just like in current traditional systems.

This isn't to say that merchants can't influence the banks, they can if people decide to continue transacting with them. There are aspects to this that are mutual, for example the idea of royalty and commission collection. There exists a vendor that wants to sell their product using a merchant. This merchant will hold their product and allow for commissions to be collected each time this item is sold. If a person buys this product and decides to sell it again, 10% (a number defined by the vendor in this scenario) of the total cost of the item sold is given to the vendor as a commission. In practice, this could be healthy for a vendor that needs people to help sell their product through a merchant that holds a large quantity of said product. A flaw in traditional systems to think about with commission collection: Who would enforce this collection in practice?

The bank has an obligation to enforce this rule, but the merchant is the one who tracks this value. Some merchants may decide to ban commission collection altogether but its enforcement is done at the bank level, since the bank oversees all transactions that take place.

Since the merchant sets this commission rate value to be enforced, it can also decide how it can be changed in the future after a listing. Some merchants can be fine with setting it once and it being set in stone. Others may believe that it's okay to manipulate at any time after it's been sold. The entity allowed to change this value may also differ, should it be the product's creator or the last person who sold it? These are things to consider that not all participants of such a system may agree on. Thus, the importance for multiple merchants with different rules to coexist under a meta system that enforces the rules is most ideal.

Bank-to-Merchant ABI

There are many other aspects of this ecosystem that the merchant controls that the bank must enforce, the following are the functions of the current QMSI-721 interface that's used by the QMSI-20 contract¹⁴:

```
tokenCommission(uint256 tokenId) external view returns (uint256);
tokenURI(uint256 tokenId) external view returns (string memory);
tokenPrice(uint256 tokenId) external view returns (uint256);
ownerOf(uint256 tokenId) external view returns (address);
tokenMinter(uint256 tokenId) external view returns (address);
buyToken(address from, uint256 tokenId) external;
trueMintingPrice() external view returns (uint256);
create(bytes32 dataHash, string calldata tokenURI_, uint256 tokenPrice_,
uint256 commission_, address minter_) external returns (uint);
```

¹⁴ <https://github.com/Qumosi/QMSI/blob/08658dfef4cd04652715a2bc9c46ad29b476448d/QMSIToken.sol#L11>

tokenCommission is one we've already discussed, it returns a percent integer value between 0 - 100 that is applied to the selling event (*CrossTokenBuy*) that takes place at the QMSI-20 level, a value of 0 means it is no commission is to be collected, therefore optional.

tokenURI tracks the location of an artifact (typically JSON) that holds additional metadata surrounding the "product".

tokenPrice is a value in the units of QMSI-20 that the QMSI-721 product is being sold for, a value of 0 means it is not being sold, therefore optional.

ownerOf defines who the current owner of the product is, typically the vendor or a seller.

tokenMinter defines who the creator of the product is, typically the vendor, this is the address in which commissions are paid out to.

buyToken is a function that is responsible for transferring the product to its new owner when bought, this is enforced by the merchant since they are holding the product on behalf of the vendor, but used by the bank only after successfully transacting.

trueMintingPrice is the cost in QMSI-20 units of what it costs to place a product on the shelves of the merchant. This cost is not collected by the merchant, it is a value that is burned while also influenced by a rate set by the bank.

create is a function that mints a product on the shelves of the merchant's store.

These functions within the QMSI-721 interface can be defined in any way imaginable by the merchant owners so long as they follow the same input/output structure of the immutable interface. An example of what an ideal QMSI-721 merchant smart contract looks like from our perspective can be found on our Github repository¹⁵.

4. The Open-Market Protocol

This section defines the two core functions that bridge a QMSI-20 contract with many QMSI-721 contracts. The initial *market* parameter of both functions is the address of the QMSI-721 compatible market it will interface with. *crossTokenBuy* allows the user to buy any digital certificate that has an active listing (meaning it is being sold), and *crossTokenCreate* allows minting a new digital certificate.

Some security considerations that were applied in both of these functions is the use of parameterized dynamic values such as *tokenPrice*. This is to ensure that the user executing on these functions is fully aware of certain aspects of the digital certificate listing to prevent out-of-sync style attacks, like changing the certificate price or royalty percentage in a sandwich attack.

A copy of these functions can be found on our Github repository¹⁶.

Cross Token Buy

The following function allows claiming ownership of a certificate by transferring a unit of QMSI-20 tokens to its current owner while respecting royalties, if the certificate has an active listing.

¹⁵ [QMSI/QMSICertificate.sol at master · Qumosi/QMSI · GitHub](https://github.com/Qumosi/QMSI/blob/master/QMSICertificate.sol)
<https://github.com/Qumosi/QMSI/blob/master/QMSICertificate.sol>

¹⁶ [QMSI/QMSIToken.sol at master · Qumosi/QMSI · GitHub](https://github.com/Qumosi/QMSI/blob/master/QMSIToken.sol)

```

function crossTokenBuy(address market, address to, uint256 tokenId,
uint256 tokenPrice_, uint256 tokenCommission_) public returns (bool) {
    require(balanceOf(msg.sender) >= tokenPrice_, "QMSI-ERC20: Insufficient
tokens");
    require(_isContract(market) == true, "QMSI-ERC20: Only contract
addresses are considered markets.");
    // To prevent price manipulation by making user aware of the price by
including it in the function call
    require(tokenPrice_ == QMSI721(market).tokenPrice(tokenId),
"QMSI-ERC721: Price is not equal");
    // To prevent commission manipulation by making user aware of the rate
prior to making the function call
    require(tokenCommission_ == QMSI721(market).tokenCommission(tokenId),
"QMSI-ERC721: Commission rate does not match");
    require(tokenCommission_ <= 100 && tokenCommission_ >= 0, "QMSI-ERC721:
Commission must be a percent");
    require(bytes(QMSI721(market).tokenURI(tokenId)).length > 0,
"QMSI-ERC721: Nonexistent token");

    require(QMSI721(market).tokenPrice(tokenId) > 0, "QMSI-ERC721: Token not
for sale");
    require(msg.sender != QMSI721(market).ownerOf(tokenId), "QMSI-ERC721:
Cannot buy your own token");
    require(to == QMSI721(market).ownerOf(tokenId), "QMSI-ERC721: Sending
tokens to the wrong owner");

    if(tokenCommission_ > 0 && msg.sender !=
QMSI721(market).tokenMinter(tokenId)){
        transfer(to, (tokenPrice_ * (100 - tokenCommission_) / 100);
        transfer(QMSI721(market).tokenMinter(tokenId), (tokenPrice_ *
tokenCommission_) / 100);
    }else{
        transfer(to, tokenPrice_);
    }
    QMSI721(market).buyToken(msg.sender, tokenId);
    emit CrossTokenBuy(msg.sender, market, to, tokenPrice_);
    return true;
}

```

Parameter Name	Description
market	The address of the certificate contract
to	The address of who we're sending tokens to
tokenId	The tokenId of the token we're buying from market
tokenPrice_	The price of the token in QMSI-20 (1e18)
tokenCommission_	The commission of the token, a integer percentage

Cross Token Create

The following function allows minting digital certificates on any compatible QMSI-721 contract while abiding by the burnRate imposed on the set minting price

```
function crossTokenCreate(address market, bytes32 dataHash, string
calldata tokenURI_, uint256 tokenPrice_, uint256 commission_, uint256
mintingPrice_) public returns (uint) {
    // verify that user is aware of the 721 market mint price, so no
manipulation can occur
    uint256 mintingPrice = QMSI721(market).trueMintingPrice();
    require(mintingPrice_ == mintingPrice, "QMSI-ERC20: Minting price does
not match");
    require(tokenPrice_ <= maxSupply() && tokenPrice_ >= 0 && mintingPrice_
<= maxSupply() && mintingPrice_ >= 0, "QMSI-ERC20: Invalid units for token
or minting prices");
    require(commission_ <= 100 && commission_ >= 0, "QMSI-ERC721: Commission
must be a percent");
    require(bytes(tokenURI_).length > 0, "QMSI-ERC721: Must define token URI
string");
    // determine value that needs to be burned, will always be equal to or
less than minting price
    uint256 burnValue = (mintingPrice*this.burnRate())/100;
    require(balanceOf(msg.sender) >= burnValue, "QMSI-ERC20: Insufficient
tokens");
    spend(burnValue);
    emit CrossTokenCreate(msg.sender, market, burnValue);

    return QMSI721(market).create(dataHash, tokenURI_, tokenPrice_,
commission_, msg.sender);
}
```

Parameter Name	Description
market	The address of the certificate contract
dataHash	A representation of the certificate metadata using keccak-256
tokenURI_	The remote location of the certificate's JSON artifact, represented by the dataHash
tokenPrice_	The (optional) price of the certificate in token currency in order for someone to buy it and transfer ownership of it (can be 0 for no listing)
commission_	The (optional) percentage that the original minter will take each time the certificate is bought (can be 0 for no commission)
mintingPrice_	The price of minting a certificate without the burnRate applied (trueMintingPrice)

5. Conclusion

The hypothetical system described in this paper can help reduce spam while ranking high quality content in a dynamic manner that is self governed with no pre-approvals required. The system uses a faucet, burning, and staking mechanisms to simulate token circulation by influencing the cost barriers associated with each action. Although many of these concepts are subjective and hypothetical, we conclude this paper by including a reference to its existence on the Ethereum mainnet¹⁷.

¹⁷ QMSI-ERC20 at [0x1B06DfdE22bE46C89ECF43EE72C6D710F4A46fC](#)
QMSI-ERC721 at [0xA0D1025bC39106b04B755b96645498f351781B33](#)

6. Appendix

QMSI-20 token smart contract ABI

This is a standard ERC-20 contract implemented using the [OpenZeppelin template](#) with additional functions:

- `drinkFromFaucet()` – Allows end user to earn QMSI ERC-20 tokens by following negative decay curve for token distribution, reset each day.
- `dailyClaimLimit() view` – Showcases the number of QMSI ERC-20 tokens that are available to claim per day. This number is influenced by the halving events.
- `daysConsumed() view` – Showcases number of days the faucet was used.
- `halvingInterval() view` – Tracks the interval for halving events that slash daily reward limit by 1/2 each time. Each unit for the interval is based off of `daysConsumed`.
- `rewardPerClaim() view` – Showcases how much potential rewards can be earned by drinking from the faucet. This value changes if another user drinks after its execution.
- `tokensClaimedToday() view` – Tracks the number of total tokens claimed in the current day. Cannot surpass the daily claim limit.
- `lastClaim(address) view` – Tracks the last time the user used the faucet via `block.timestamp`.
- `circulatingSupply() view` – Returns the number of tokens that are liquid in addition to tokens that are in either pool (burn or stake). This number can be used to see how many coins have been dispensed by the faucet in total.
- `spend(uint256 value)` – Allows end user to burn their own tokens. It can only be triggered by the user, and is used in minting new certificates.
- `burnPool() view` – Returns the amount of tokens that can be reclaimed through staking. Using the `spend` function increments this pool
- `burnRate() view` – Returns a number 0 to 100 that represents the percentage of tokens in circulation (current supply / max supply)
- `stake(uint256 days_, uint256 value_)` – Allows locking ERC20 tokens for number of days to reclaim some burned tokens as reward
- `unlockTokens()` – Allows unlocking locked ERC20 tokens and reward after the unlock date has reached
- `totalStaked() view` – Returns total number of staked ERC20 tokens
- `lockedBalanceOf(address account) view` – Returns the locked balance of an address that is staking
- `unlockDate(address account) view` – Returns the unlock date for an address that is staking
- `rewardsCalculator(uint256 days_, uint256 value_) view` – Helps estimate proposed reward given by the system
- `crossTokenBuy(address market, address to, uint256 tokenId, uint256 tokenPrice_, uint256 tokenCommission_)` – Allows end user to buy ERC721 certificates that have a listing from any QMSI-721 implementation

- `crossTokenCreate(address market, bytes32 dataHash, string memory tokenURI_, uint256 tokenPrice_, uint256 commission_, uint256 mintingPrice_)` – Allows end user to create ERC721 certificates on any QMSI-721 implementation while enforcing burn rate ontop of minting cost
- `setQNS(uint256 qid)` – Allows end user to set their Qumosi profile ID on the associated wallet
- `getQNS(address account) view` – Returns Qumosi profile ID associated with wallet address

QMSI-721 market smart contract ABI

This is a standard ERC-721 contract implemented using the [0xcert template](#) with additional functions:

- `create(bytes32 dataHash, string memory tokenURI_, uint256 tokenPrice_, uint256 commission_, address minter_)` returns `(uint256)` – Allows anybody to create a certificate (NFT) via the ERC20 contract `crossTokenCreate` function. Causes the side effect of deducting a certain amount of money from the user, payable in ERC-20 tokens. The return value is a serial number. It is called by the ERC20 contract only.
- `hashForToken(uint256 tokenId) view` – Allows anybody to find the data hash for a given serial number.
- `mintingPrice() view` – Returns the mint price influenced by the burn rate.
- `trueMintingPrice() view` – Returns the mint price without influence.
- `mintingCurrency() view` – Returns the currency (ERC-20)
- `setMintingPrice(uint256)` – Allows owner (see [0xcert ownable contract](#)) to set price that is later influenced by the burn rate
- `setMintingCurrency(ERC20 contract)` – Allows owner (see [0xcert ownable contract](#)) to set currency
- `setBaseURI(string memory baseURI_)` – Allows the contract owner to set a prefix URI on all resource links. Implemented using [OpenZeppelin](#)
- `setTokenURI(uint256 tokenId, string memory tokenURI)` – Allows minter to set token URI resource link to JSON artifact. Implemented using [OpenZeppelin](#)
- `setTokenCommissionProperty(uint256 tokenId, uint256 percentage_)` – Allows minter to take percentage of proceeds from SoldNFT events
- `setDeadmanSwitch(address kin_, uint256 days_)` – Allows the contract owner to set ownership to transfer to a "kin" address if the number of specified days surpass without this function being called again, in order to reset the timer.
- `claimSwitch()` – Allows the `kin` of the switch to claim it, transferring ownership. Only works if the switch is expired from the time it was set. If switch expires, owner is assumed dead.
- `getKin() view` – Returns the address of the next of kin
- `getExpiry() view` – Returns the expiration timestamp of the switch that allows ownership transfer

- `buyToken(address from, uint256 tokenId)` – Allows transferring ownership of certificate if tokens have been transferred to the owner that made the sell token listing
- `sellToken(uint256 tokenId, uint256 _tokenPrice)` – Allows the certificate owner to sell token by specifying the price in ERC20 units and token id
- `removeListing(uint256 tokenId)` – Allows the certificate owner to remove an existing listing to sell the token by specifying the token id
- `tokenPrice(uint256 tokenId) view` – Returns the price of a certificate, if it is listed for sale
- `tokenMinter(uint256 tokenId) view` – Returns the original minter of a certificate
- `tokenURI(uint256 tokenId) view` – Returns the resource link to the token's (JSON) artifact. Implemented using [OpenZeppelin](#)
- `tokenCommission(uint256 tokenId) view` – Returns the percent token commission rate that is taken by the original minter per SoldNFT event