A PROPOSAL FOR A REORGANIZED EIP 918

Author: Rick Park, rpsnoopy@me.com, ceo@xbdao.io

The current revision of the EIP918 standard suffers of some usability problems related to: some incongruences in naming convention in the various sections, apparent definition of externally visible variables and not of the methods used to access them (even if in the most case some their implicit getter is implied), huge technical material not perfectly congruent in the various sections, and so on. As a result of the actual situation, the use of the specification is somehow difficult for the mean skilled programmer.

The proposed revision aim is to have a part organized using the so called “lawyer paradigm”, where what is not specified is permitted and any word of the specification RESTRICTS the possible implementations to the proposed standard. Suitable examples of this approach in the Ethereum field can be found in the ERC20 specification. The proposal have a first part (here below described) mandatory, where all the new proposed minable tokens as per the proposed standard are asked to precisely follow in any aspect. There is a second part which recommends some not mandatory aspect, which are suitable to simplify and rationalize the internal structure of the contract.

A suitable backwards compatibility section is presented, where already existing token, if implementing at least that interface, should be declared compliant to EIP918-B because substantially EIP918 compliant, but for some naming convention.

For the sake of simplicity, a suitable EIP918 interface file (“IEIP918.sol”) is presented to be included in the source code for compatibility assurance by means of inheritance and local overloading, while an Abstract Contract file (“AEIP918.sol”) suited to guarantee both EIP918 and EIP918B compatibility is presented for possible inclusion.

All the huge technical material present in the previous draft of this EIP should be reorganized under the section “Implementation notes and examples”.

Is to be evaluated the needing and opportunity to include the merge mining and delegate mining definition at this level: probably it can be treated in the recommended practice section only, being a very particular case.

EIP 918 – Mineable Token Standard

Simple Summary

A specification for a standardized Mineable Token that uses a Proof of Work algorithm for distribution.

Abstract

The following standard allows for the implementation of a standard API for tokens within smart contracts when including a POW (Proof of Work) mining distribution facility.

In this kind of token contract, tokens are locked within a token smart contract and slowly dispensed by means of a mint() function which acts like a POW faucet when the user submit a valid solution of some Proof of Work algorithm. The tokens are dispensed in chunks formed by some tokens (reward).

Motivation

The rationale for this model is that this approach can both minimize the gas fees paid by miners in order to obtain tokens and precisely control the distribution rate.

The standardization of the API will allow the development of standardized CPU and GPU token mining software, token mining pools and other external tools in the token mining ecosystem.

This standard is intended to be integrable in any token smart contract which eventually implements any other EIP standard for the non-mining-related functions, like ERC20, ERC223, ERC721, etc.

It can be here mentioned that token distribution via POW is considered very interesting in order to minimize the investor's risks exposure related to eventual illicit behavior of the "human actors" who launch and manage the distribution, like those implementing ICO's (Initial Coin Offer) and derivatives. The POW model can be totally realized by means of a smart contract, excluding human interferences.

Specification

Mandatory methods

challengeNumber()

Returns the current challengeNumber, i.e. a byte32 number to be included (with other elements, see later) in the POW algorithm input in order to synthetize a valid solution. It is expected that a new challengeNumber is generated after that the valid solution has been found and the reward tokens have been assigned.

```
function challengeNumber() view public returns (bytes32)
```

NOTES: in a common implementation, challengeNumber is calculated starting from some immutable data, like elements derived from some past ethereum blocks.

difficulty()

Returns the current difficulty, i.e. a number useful to estimate (by means of some known algorithm) the mean time required to find a valid POW solution. It is expected that the difficulty varies if the smart contract controls the mean time between valid solutions by means of some control loop.

```
function difficulty() view public returns (uint256)
```

NOTES: in a common implementation, difficulty varies when computational power is added/subtracted to the network, in order to maintain stable the mean time between valid solutions found.

epochCount()

Returns the current epoch, i.e. the number of successful minting operation so far (starting from zero).

```
function epochCount() view public returns (uint256)
```

adjustmentInterval()

Returns the interval, in seconds, between two successive difficulty adjustment.

```
function adjustmentInterval () view public returns (uint256)
```

NOTES: in a common implementation, while difficulty varies when computational power is added/subtracted to the network, the adjustmentInterval is fixed at deploy time.

miningTarget()

Returns the miningTarget, i.e. a number which is a threshold useful to evaluate if a given submitted POW solution is valid.

```
function miningTarget () view public returns (uint256)
```

NOTES: in a common implementation, the solution is valid if lower than the miningTarget.

miningReward()

Returns the number of tokens that POW faucet shall dispense as next reward.

```
function miningReward() view public returns (uint256)
```

NOTES: in a common implementation, the reward progressively diminishes toward zero trough the epochs ("epoch" is mining cycle started by the generation of a new challengeNumber and ended by the reward assignment), in order to have a maximum number of tokens dispensed in the whole life of the token smart contract, i.e. after that the maximum number of tokens has been dispensed, no more tokens will be dispensed.

tokensMinted()

Returns the total number of tokens dispensed so far by POW faucet

```
function tokensMinted() view public returns (uint256)
```

mint()

Returns a flag indicating that the submitted solution has been considered the valid solution for the current epoch and rewarded, and that all the activities needed in order to launch the new epoch have been successfully completed.

function mint(uint256 nonce) public returns (bool success)

In particular, the method verifies a submitted solution, described by the nonce (see later):

- If the solution found is the first valid solution submitted for the current epoch:
  - rewards the solution found sending No. miningReward tokens to msg.sender;
  - creates a new challengeNumber valid for the next POW epoch;
  - eventually adjusts the POW difficulty;
  - return true
- ELSE if the solution is not the first valid solution submitted for the current epoch, it returns false (or revert)

NOTE: **The first phase (hash check) MUST BE implemented using the specified public function hash()**, while an internal function structure is recommended (see Reccomendation), but it is not mandatory.

hash()

Returns the digest calculated by the algorithm of hashing used in the particular implementation, whatever it will be.

function hash(uint256 nonce, address minter, bytes32 challengeNumber) public returns (bytes32 digest)

NOTES: hash() is to be declared public and to be written including explicitly uint256 nonce, address minter, bytes32 challengeNumber in order to be useful as test function for mining software development and debugging.

event Mint()

TO BE MANDATORY EMITTED immediately after that the submitted solution is rewarded. The Mint event indicates the rewarded address, the reward amount, the epoch count and the challenge number used.

event Mint(address indexed _to, uint _reward, uint _epochCount, bytes32 _challengeNumber)

Recommendation

MITM attacks

To prevent man-in-the-middle attacks, the msg.sender address, which is the address eventually rewarded, should be part of the hash so that any nonce solution found is valid only for that particular Ethereum account and it is not susceptible to be used by other. This also allows pools to operate without being easily cheated by the miners because pools can force miners to mine using the pool's address in the hash algorithm. In that a case, indeed, the pool is the only address able to collect rewards.

Anticipated mining

In order to avoid that miners are in condition to calculate anticipated solutions for later epoch, a "challengeNumber", i.e. a number somehow derived from existing but mutable conditions, should be part of the hash so that future blocks cannot be mined before. The "challengeNumber" acts like a random piece of data that is not revealed until a mining round starts.

Hash functions

The use of solidity keccak256 algorithm is strongly recommended, even if not mandatory, because it is a very cost effective one-way algorithm to compute in the EVM environment and it is available as built-in function in solidity.

Solution representation

The recommended representation of the solution found is by a 'nonce', i.e. a number, that miners try to find, that being part of the digest make the value of the hash of the digest itself under the required threshold for validity.

mint() internal structure

From the miner point of view, submitting a solution for possible reward means to call the mint() function with the suitable arguments and waiting for evaluation results.

It is recommended that internally the mint() function be realized invoking 4 separate successive phases: hash check, rewarding, epoch increment, difficulty adjustment.

The first phase (hash check) MUST BE implemented using the specified public function hash(), while an internal function structure is recommended, but it is not mandatory. In particular the following phases, being totally internal to the contract, cannot be specified as mandatory, but the following schema is recommended:

In the preferred realization, for each of those steps a suitable function is declared and called:

1) hash check: -> MANDATORY: by means of *hash()* (already specified);

2) rewarding: -> by means of some function *_reward()* internal returns (uint);

3) epoch increment -> by means of some function *_epoch()* internal returns (uint);

4) difficulty adj. -> by means of some function *_adjustDifficulty()* internal returns (uint);

It may be useful to recall that a Mint event MUST BE emitted before returning a boolean success flag.

In a sample compliant realization, the mint can be roughly described as follows:

```
function mint(uint256 nonce) public returns (bool success) {
        require (hash(nonce, minter, challengeNumber) >= byte32(miningTarget), "Invalid solution");
        emit Mint(minter, _reward(),_epochCount, _challengeNumber);
        _epoch();
        _adjustDifficulty();
        return(true);
}
```

Backwards Compatibility

In order to facilitate the use of both existing mining programs and existing pool software already used to mine previous minable tokens, the following function can be included in the contract. They are simply a wrapping of the above defined functions:

```
function getAdjustmentInterval() public view returns (uint) {
         return adjustmentInterval();
}

function getChallengeNumber() public view returns (bytes32) {
         return challengeNumber();
}

function getMiningDifficulty() public view returns (uint) {
         return difficulty();
}
function getMiningReward() public view returns (uint) {
         return miningReward();
}

function mint(uint256 _nonce, bytes32 _challenge_digest) public returns (bool success) {
         return mint (_nonce);
}
```

Any already existing token implementing this interface can be declared compliant to EIP918-B. A suitable inheritable abstract contract ("AEIP918B.sol") is provided for reference.

Implementation notes and examples

< here, properly reorganized, all the suitable elements from the current draft (interface, abstract contract, etc.) >

Test Cases

-

-

-

-