GCC3D TOKEN TECHNICAL SPECIFICATION

BASIC SPECIFICATION

erc20Name	**Global 3D Currency**
erc20Symbol	**GCC3D**
erc20Decimals	**6**
maxSupply	**2 400 000 000 (2 400 mln)**
stakeMinAge	**24h**
stakeMaxAge	**90 days**
stakeReward	= according to table, fixed daily reward pool calculated for each year separately
stakeMinAmount	**10 000**
stakeStart	`Moment.utc().add(1, 'days').unix()` // next midnight! **60 days** since SWAP starts
stakeInterval	86400 (**24h**)

stakeMinAge	min coinAge counted in days to claim reward
stakeMaxAge	max coinAge counted in days to claim reward
maxTotalSupply	limit total supply
minStakeAmount	this is a minimal token amount which enables staking
stakeStart	it's a timestamp, when staking and MN is launched in the network

COSTS

Costs of GCC token operations cannot be precisely described as fixed. Cost depends on number of operations which contract needs to perform. This depends on absolute (total) staking length, current staking length of particular user and other users' staking.

Min. cost would be applied after 1st day of users staking, highest when user will wait whole 90 days stake interval.

COSTS RANGE:

stake/stakeAll	– 0.01 - 0.02
unstake/unstakeAll	– 0.0075 - 0.015
withdrawReward/estimateReward	– 0.005 - 0.01

Above have to be multiplied by GAS price, which user is going to use executing command/transaction. Gas price = how fast transaction is going to be approved/ confirmed by the network. For latest prices see https://ethgasstation.info/

FEATURES AND FUNCTIONS

Possibility to pause contract	✓
Minting of new tokens by contract owner	✗
Burning tokens by contract owner	✓
Code upgrades possible	✗

TOKEN IS COMPATIBLE WITH INTERFACE ERC20 EIP–20 + ADDITIONAL FUNCTIONS:

- **stake**
- **stakeAll**
- **unstake**
- **unstakeAll**
- **estimateReward**
- **withdrawReward**

To stake GCC Token User must have wallet Ethereum supporting Smart Contracts (like MyEtherWallet, Jaxx, Mist etc). To start staking – user execute command "stake" or "stakeAll". To withdraw rewards, execute "withdrawReward". To cancel staking "unstakeAll".

TOKENOMY

There's a reward pool assigned for each year and hardcoded into the smart contract according to the chart. One year is divided for 365 days what gives fixed daily reward pool to share between staking addresses:

STATUS

'STAKE NODE'

addresses with less than 1 000 000 GCC put to staking, shares with other 'Stake node' status addresses 20% of daily rewards.

STATUS

'MASTER NODE'

addresses with over 1 000 000 GCC put to staking, shares with other 'Masternode' addresses 80% of daily rewards.

MN: DAILY REWARD POOL * 0.8 * myMnStake / allMnStake

STAKE: DAILY REWARD POOL * 0.2 * myStake / allStandardStake

User becomes a masternode automatically when staking more than
1M GCC tokens (1 000 000 GCC = 1 Masternode 1 500 000 = 1.5 Masternode)

TOKEN STATUS

There are three type of 'token status' we could consider:

1. Balance immediately available (Fully disposable) – always ready to transfer, stake… or burn :)
2. Staked balance (locked)
3. Calculated stake rewards – physically don't exists until converted to 'disposable balance', could be called 'rights to rewards' or 'rewards due'

User can stake any amount of his GCC token balance. For example, if he has total 1 000 GCC, he can launch staking by executing commands:

Stake (200)

and next

Stake (400)

After executing such commands user will have 400 GCC left at his immediate disposal and 600 GCC locked for staking.

To get 600 GCC back to his standard wallet balance, user need to execute 'unstake' command.

Rewards are calculated basing only on currently staked tokens, which comes from wallet balance.

So, continuing example above, if user after a month will receive rights to 6 GCC rewards – he will be still staking 600 tokens only. To add 6 GCC to stake, user should execute WITHDRAW REWARDS command, so 6 GCC adds to his general wallet balance. Then need to execute STAKE (6) command to add 6 GCC to already staking 600 GCC tokens. Instead of those 2 commands, user could execute STAKE ALL command only to get the same result. Coinage of 606 GCC stake resets to 0.

Another option would be executing UNSTAKE (600) or UNSTAKE ALL – both initial staking balance and received rewards will increase general wallet balance. Next, user also need to use STAKE (tokens_amount) or STAKE ALL command to start staking again.

Minimum staking time is 24h. If user will launch UNSTAKE or WITHDRAW REWARS commands before 24h since staking cycle started, he will not receive any fraction of his latest daily reward – all latest staking hours < 24h will not be considered to generate reward.

MaxStakeTime – 90 days. After this time staking is automatically turned off. User should execute UNSTAKE ALL or WITHDRAW REWARDS to put all tokens into general wallet balance and then launch STAKE ALL/STAKE (amount) commands

EstimateReward function - calculates estimated rewards available to withdraw.

WithdrawReward – do not reset coinage/staking interval (90 days)

GCCPEDIA

A contract that implements GCC logic inside ETH blockchain. Its main feature is the staking mechanism for tokens. Staking is implemented discreetly.

FEATURES

- ◯ **ERC20-compatbile**
- ◯ **Allows setting up name, symbol and number of decimals**
- ◯ **Ownable**
- ◯ **Burnable**
- ◯ **Stakable Discreetly**
- ◯ **Pausable**
- ◯ **Oraclized with GCCOracle**

ERC20-COMPATIBILITY

This contract has all the methods specified in ERC20 and behaves as ERC20 token.

OWNABLE

This contract exposes few methods that can be used only by the owner.

BURNABILITY

This contract allows holders to burn their tokens. Burning removes given amount of token from circulation, decreases its current supply, but does not change max possible supply.

To burn tokens user can use either burn or burnFrom methods which behaves similar to ERC20's transfer and transferFrom with the distinction that recipient of burn transaction is "zero-adress" which makes token sent there unrecoverable.

STAKABILITY

This token allows its holder to stake their holdings. Any holder can manually stake any amount of coins belonging to him as long as specified amount is larger that minimal stake amount. Staked coins are taken from user balance and he cannot use those coins until he manually unstakes them. Unstaking can be completed at any point of time, but it resets coinage (expressed in a set of discrete intervals) linked to those coins effectively resetting staking reward as well.

Stakbility of this token differs from relative model in two major ways:

- Coinage is no longer expressed as linear function, but set of discrete intervals
- Reward is not calculated as const % of staked coins, but sum of rewards generated for each discrete invterval during which user had coins staked. Reward for each interval is calculated separately for each staking group (wallet/masternode) and expressed as % from max stake reward. Ratio for determining the share of max reward is calculated as amount of all staked coins by user divided by sum of all staked coins by everyone.

ACTIVE STATE / TECHNICAL INTRODUCTION

Stakability module works on top of Active State design.

To properly implement discrete stakability logic on quite limiting Ethereum environment, the timeline for staking is divied into equal intervals. Each interval generated possible reward that users can claim by staking. Those intervals, inside code, are called "active states" and can be compared to virtual blockchain inside contract.

Contracts deployed on Ethereum do not posses active lifecycle. That means, they cannot execute any logic in the background by themselves. That means to make any change to internal state of the contracts an external transaction needs to be made. Unfortunatelly discrete staking needs to apply changes to internal state of the contract on each new interval.

To workaround this limitation an Active State design was implemented. It works similar to long-pulling in more standarized IT solutions and takes advantage of the observability rule that object does not exist unless its observable. For staking needs that translates to fact that change in contract state does not exist unless its being accessed. If consumer wants to access or change any information in the state - only then - the missing states between last access and now are calculated. That ensures the newest state in the contract and simulates it calculating new states by themself all the time.

STAKING TOKENS

Staking of tokens can be performed at any time. Staking work in additive way, so by staking user adds specified amount to stake instead of replacing it. Staked tokens are moved form user balance to user stake. Staked tokens can then be used to generate rewards. Staked tokens cannot be used in transfers, they need to be withdrawn first.

Staking triggers reward generation, if user already had stake eligible for a reward.

UNSTAKING TOKENS

Unstaking of tokens can be performed at any time. This process removes all of user staked tokens and moves it back to his/her balance. That way those tokens can be used for transfers once again, but they do not generate rewards any longer.

Unstaking triggers reward generation, if user was eligible for reward for the funds he unstaked.

STAKING INTERVALS

Discrete staking uses staking intervals rather than coinage for determing staking reward for each user. Each interval generates possible reward that users can claim by staking. Those intervals, inside code, are called "active states" and can be compared to virtual blockchain inside a contract.

REWARDS

Rewards are calculated as the sum of the rewards for each staking interval during which user had tokens staked. Reward for each interval is calculated by following formula: `maxRewardForInterval` * (`userStakedCoins / allStakedCoins`). If the reward is equal to 500 and number of all staked coins is equal to 10000 and user held 1000 coins, then he will get 50 coins (10% of 500). Since users are able to stake and unstake their funds freely, the ratio and reward size for each interval can differ.

User can withdraw reward whenever he wants if his estimated reward is more than zero. By withdrawing, the reward is added to user's balance while staked tokens remaining staked for future intervals. If user wants to stop staking he needs to manually unstake after withdrawing reward. It is possible to withdraw and unstake all funds at once with appropriate function.

In addition each user is able to determine his estimated reward without need to actually withdraw it.

REWARDS EXPIRY

Rewards do not expire at any time for the user, but there is a limit for how long user coins can generate rewards. This limit can be configured during contract deployment and is introduced to "cancel" staking from zombie-accounts at some point.

PAUSABILITY

This contract allows its owner to pause execution of methods that change state of the contracts. That includes all of the methods related to transfers, allowance, burning, minting and staking. When contract is paused all of the getters are working normally.

INTERFACE

Only contract owner is able to unpause the contract.

This contract has ERC20-compliant interface that can be found in EIP-20. In addition it introduces methods for Burning and Pausing it defined in components `ERC20Burnable, Pausable`. On top of that it adds the custom logic related to staking and handling GCCOracle.

STAKING INTERFACE

public :: stakeOf

Returns the number of coins staked by given account. This method returns all the funds, including expired stakes.

```
function stakeOf(address account) public view returns
(uint256)
```

public :: activeStakeOf

Returns the number of valid coins staked by given account. This method returns only number of funds which are still eligible for reward in the next interval.

```
function activeStakeOf(address account) public view returns
(uint256)
```

public :: stake

Allows transaction sender to stake his coins. This method works by adding specified amount to stake, not by replacing it.

```
function stake(uint256 amount) public activeStake returns
(bool)
```

public :: stakeAll

Similar method to stake, but it stakes all of the tokens held by user at that time.

```
function stakeAll() public activeStake returns (bool)
```

public :: unstake

Allows transaction sender to unstake some of his coins.

```
function unstake(uint256 amount) public activeStake returns
(bool)
```

public :: unstakeAll

Allows transaction sender to unstake all of his coins.

```
function unstakeAll() public activeStake returns (bool)
```

public :: estimateReward

Returns reward estimated for transaction sender. It does not withdraw the reward, only calculates it.

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

public :: withdrawReward

Withdraws reward for user's staked coins.

```
function withdrawReward() public activeStake returns (bool)
```

public :: nextCheckpoint

Returns timestamp when next staking interval starts. Can be used to determine at which point new reward can be received.

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

public :: lastCheckpoint

Returns timestamp when current staking interval started.

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

onlyOwner :: tick

Forces contract to try recalculate newest possible state within limit of steps performed.

This method is exposed only for owner for the purpose of performing partial state update. Partial state update is useful when contract has not been used in a very, very long time and activeState is not able to update itself because of limit of block size. This should never happen but just in case this method is able to solve the problem.

```
function tick(uint256 repeats) public onlyOwner returns (bool)
```

onlyOwner :: tickNext

Forced contract to try recalculate next state. This method works same as tick with step limit of one.

```
function tickNext() public onlyOwner returns (bool)
```

onlyOwner :: rewardUser

This method allows owner to withdraw staking reward on behalf of the user. Reward is send to the user (not owner!) and maybe used to automate sending up rewards if needed.

```
function rewardStaker(address account) public onlyOwner
activeStake returns (bool)
```

Oracle Consumer Interface

Oracle consumer interface is the part of the GCCDiscrete that is responsbile for communicating with the Oracle contract. The communication is limited to reading newly added proofs from Oracle and executing hook methods for each of them. The hook method is a mint function that creates proved funds for the address.

Funds minted by consumer have two restrictions:

- needs to be proven by separate Oracle service
- cannot exceed the limit of provable funds configured during consumer deployment

public :: consumeProofs

Consumes not-yet consumed proofs from oracle for a given address and mints swapped funds.

```
function consumeProofs(address addr) public returns (bool)
```

public :: consumeLimitedProofs

Consumes not-yet consumed proofs from oracle for a given address with a limit on how many new proofs are parsed.

```
function consumeLimitedProofs(address addr, uint256 limit)
public returns (bool)
```

public :: getConsumableFullAmount

Returns the limit on how many tokens can contract generate using Oracle service.

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

public :: getConsumableUsedAmount

Returns number of tokens that were already minted by the contract.

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

public :: getConsumedProofs

Returns list of proofs which were already consumed for given address.

```
function getConsumedProofs(address addr) public view returns
(address[] memory, bytes32[] memory, uint64[] memory)
```

public :: fewConsumedProofs

Returns list of proofs which were already consumed for given address with possibiliy to use self-made cursor.

```
function fewConsumedProofs(address addr, uint cursor, uint
limit) public view returns (address[] memory, bytes32[]
memory, uint64[] memory)
```

Oracle Reader Interface

Oracle reader interface is the part of the GCCDiscrete that is responsbile for communicating with the Oracle contract. The communication is limited to acquiring publicly-available information from Oracle contract.

This interface is part of Oracle. More information about keywords used here can be found in Oracle documentation.

public :: getFilterLength

Returns the number of all filters used up to this point.

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

public :: getFilter

Get metadata about nth-filter used.

```
function getFilter(uint256 index) public view returns (string
memory, string memory, string memory, uint256)
```

public :: nowFilter

Get metadata about current filter used.

```
function nowFilter() public view returns (string memory,
string memory, string memory, uint256)
```

public :: getProof

Returns proof for given address-txid pair. Returns null if it does not exist.

```
function getProof(address addr, bytes32 txid) public view
```

```
returns (address, bytes32, uint64)
```

public :: getProofs

Returns all proofs for given adderss.

```
function getProofs(address addr) public view returns
(address[] memory, bytes32[] memory, uint64[] memory)
```

public :: getProofs

Returns few proofs for given address with custom cursor.

```
function getProofs(address addr, uint cursor, uint limit)
public view returns (address[] memory, bytes32[] memory,
uint64[] memory)
```