Master Chef Staking Spec

This document outlines the intended functionality of `staking.rsh`

Context

xBacked needs a dynamic staking contract for distributing incentives. These incentives could be in the form of project tokens, Algos, or even stablecoins. The incentives should also be rewarded using different weights. For example, the split of rewards might be 70% Algos and 30% projects tokens. These weightings can be updated in flight.

High level contract functionality

- The deployer sets initial parameters

```
const InitializationParameters = Struct([
  // the token to be staked
  ['stakingASAInfo', Tuple(Token, UInt)],
  // the reward token (typically the project token)
  ['rewardASAInfo', Array(Tuple(Token, UInt), NUM_EXTRA_REWARD_ASSETS)],
  // once contract is deprecated, this is the amount of time users have to claim
  ['initialDeprecateTimeout', UInt],
  // initial time stamp of when the rewards are deprecated
  ['initialDeprecateAt', UInt],
  // initial reward weihtins
  ['initialRewardRatios', Array(UInt, NUM_SUPPORTED_REWARD_ASSETS)],
  // initial per block reward rate
  ['initialRewardRate', UInt],
  // initial amount of rewards deposited by the deployer
  ['initialRewards', Array(UInt, NUM_SUPPORTED_REWARD_ASSETS)]
]);
```

- The contract allows 3 different tokens as rewards
  - Network tokens
  - The token being staked
  - A token set by the deployer (typically the project token)
- The Admin of the staking contract (initially the deployer) can perform the following actions
  - `updateAdminParameters` - update initial settings for the contract
  - `depositRewards` - deposit further rewards into the contract
- A user of the staking contract can complete the following actions
  - `deposit` - deposit `stakingASAInfo` tokens to being staking
  - `withdraw` - withdraw staked tokens

    - `cacheRewards` - update the amount of rewards they can claim. this is done within other actions, but a plain function is provided in case a user wants to update their rewards without performing another action
    - `withdrawRewards` - withdraw pending rewards accrued from staking
- Using `untrackedFunds` the contract allows **anyone (or anything)** to deposit more rewards into the contract
    - This could be another smart contract which transfers tokens to the staking contract address to be used as rewards
- Deprecation
    - Deprecation of the staking contract happens at a specified time. This time is **mutable** and can be updated by the administrator
    - The `deprecateAt` parameter is the block time the contract will be deprecated. At this time, rewards will stop accruing
    - The `deprecateTimeout` parameter is the amount of blocks users have to claim accrued rewards before the contract can be closed
- Program Invariant
    - Validates `adminParameters` are correct
    - Validates deprecation condition `(adminParameters.deprecateAt === 0 || lastRewardBlock <= adminParameters.deprecateAt)`

Reward Formulas & Rules

Found in `reward_utils.rsh`

`calculateRewardRate`

If the reward rate is a function of `remainingRewards` and `remainingBlocks` until deprecation. Rewards are capped by a `MAX_REWARD_RATE` to prevent an overflow / contract locking

`calculateRewardPerToken`

We use the current `rewardRate`, the `timeDelta` in blocks since the last reward update and the sum of all deposits within the contract

```
const newlyEmittedRewards = muldiv(
  saturationMul(timeDelta, rewardRate),
  SCALE_FACTOR,
  currentTotalDeposit
);
```

`rewardsEarned`

Calculates the difference between the current reward per token, and the reward per token when the user last claimed

```
const newRewards = muldiv(
  newRewardPerToken - rewardPerTokenPaid,
  amountDeposited,
  SCALE_FACTOR
);
```

Trade offs, Behavior & Quirks

- Saturation arithmetic is implemented so that all incrementing values will be capped at `UInt.max`, an example of where this could occur is in the `newlyEmittedRewards` calculation
- Amount of rewards 'deposited' in the contract should match the ratio set, or stakers will be paid out at the highest amount that satisfies the reward ratios
- `NUM_EXTRA_REWARD_ASSETS` should be adjustable to allow the contract to support more reward assets, however there are a number of areas where manual adjustments are required;
  - `const rewardASA1 = rewardASAs[0];` line 241 we need to give each of the reward assets distinct variables, to be used in payment expressions
  - `.paySpec([stakingASA, rewardASA1])` line 435 each asset needs to be listed explicitly, as spread syntax is not accepted here, e.g. `.paySpec([stakingASA, ...rewardAssets])` is not possible
- Reward rate is static, even when the `adminParamaters` is updated with a reward rate of 0 and a non zero `deprecateAt` time, as the reward rate is calculated once in the call to `AdminAPI.updateAdminParameters` and then treated the same as a specifically set rate across the rest of the contract
- `AnyAPI.updateLastConsensus` is used to update the `lastConsensusTime` value of the contract without mutating other contract state
- `getTime()` is an alias for `lastConsensusTime()` so that if another method becomes available to obtain the current block time of the chain, it can be replaced seamlessly
- Ideally, `StakingUserAPI.withdrawRewards` should also cache rewards prior to checking the staker's reward balance, but as things are we cannot implement this without extreme verification duration, feedback would be appreciated on this. It is important to note, this does not prevent the staker from being entitled to all rewards, just that they need to make multiple calls to this endpoint.