# Hedera High-Level Specification

## Document Overview

The purpose of this document is to provide an abstract semi-formal specification of a Hedera stablecoin, which is used as the basis for a formal specification. This document focuses on modeling these core requirements, referencing the Token Taxonomy Framework (TTF) where appropriate. Similarly to TTF, this is *not* a specification of the implementation, although it provides some guidelines for the implementers.

## Goal

The goal of the project is to define and formally verify properties of a sample *fiat-backed stablecoin*.

## Assumed DLT Semantics

We assume an EVM-like semantics for the underlying distributed ledger, which, in particular, has the following properties:

1. **Transactions.** Any transaction that throws (an exception) will revert all performed state-changes from that transaction. In the specification below, if any requirement (pre-condition) of a function is not met, the function implicitly "throws" and no state-changes occur.
2. **Integers.** Integers are internally represented as fixed-width bitstrings that behave according to modular arithmetic (i.e., arithmetic expressions that exceed the maximum value are wrapped around).
3. **Max integer.** The largest value of type Int, denoted MAX_INT, is 2^256 - 1 (corresponding to a 256-bit type). The same type Int is used for total supply, account balances, allowances, etc.
4. **Defaults.** By default, all uninitialized fields assume 0 value (or false or ””, etc. depending on the type). Furthermore, uninitialized maps assume the aforementioned default values for keys that do not exist.
5. **Addresses.** Addresses are represented as 384-bit integers**.** 0x is treated as a special, empty, address.
6. **Caller.** Each function call receives an implicit caller’s account address in `caller` variable. Since `caller` is always a legitimate account address, it is different from 0x.
7. **Synchronicity.** All function calls are assumed to be synchronous when they get executed on the distributed ledger, i.e., where the consensus takes place.
8. **Replay.** The specification assumes that message and transaction replay attacks are prevented by the distributed ledger or software between the ledger and the token.

# Conformance to TTF Taxonomy

- Base type
  - Token type: **Fungible - tF**
    - Tokens don't have identities; interchangeable
  - Token unit: **Fractional/Divisible - d**
    - Tokens can be subdivided
  - Value type: **Reference**
    - Tokens value should be pegged to a fiat value, e.g., USD
  - Representation type: **Common**
    - The token should be a bank-like ledger
  - Supply: **Gated**
    - The upper bound determined by cash
  - Template Type: **Single**
    - No dependency on other tokens
- Token formula and definition
  - Behaviors
    - Behavior
      - **Transferable - t**
      - **Mintable - m**
      - **Burnable - b**
      - **Roles - r**
        - Owner
          - transferOwnership()
          - claimOwnership()
          - changeSupplyManager()
          - changeAssetProtectionManager()
          - [any capabilities available to other roles]
        - Supply manager
          - mint()
          - burn()
        - Asset protection manager
          - freeze()
          - unfreeze()
          - wipe()
          - setKycPassed()
          - unsetKycPassed()
      - **Delegable - g**
        - Enables approveAllowance() and transferFrom()
      - **Compliant - c**
        - Checks for transfer()

                - **Subdividable - d**
                    - Can be divided into fractions
                - **Logable - l**
            - Behavior group
                - **Supply control - SC**
    - Outside of TTI scope (may be used as a proposal to extend TTI)
        - constructor()
        - getters
        - freeze()
        - unfreeze()
        - wipe()
        - setKycPassed()
        - unsetKycPassed()
        - isPrivilegedRole()
        - increaseAllowance()
        - decreaseAllowance()

# Deviation from TTF Taxonomy

1. **Ownership**. TTF assumes that every HCS Token Contract has an owner. It does not prescribe a way of transferring the ownership. Ethereum token contracts, typically, use the Ownable smart contract where the current token contract owner can simply set a new owner. A possible risk is a change to an unintended address. To mitigate this issue the specification below follows a two step process:
    a. The HCS Token Contract owner proposes a new owner (operation can be called by the HCS Token Contract owner any number of times).
    b. The proposed owner must claim ownership to become a new HCS Token Contract owner. There is no time limit on when the ownership may be claimed since the current owner may change the proposed owner at any time.
2. **Compliant**. TTF Compliant behavior specifies three functions: `CheckTransferAllowed()`, `CheckMintAllowed()`, `CheckBurnAllowed()`. The specification below implements only the first function since minting and burning can only be done by the supply manager who is assumed to have a privileged role.
3. **Other** (not prescribed by TTF)
    a. Constructor
    b. Getters.
    c. Roles other than the owner (and related functionalities).
    d. Functions for increasing and decreasing the allowance.

# HCS Token Contract Design Decisions

1. **Roles.** Although TTF allows for arbitrarily many roles (p. 52-57), here roles are hardcoded in the HCS Token Contract due to the simplicity of this solution. In other words, it is impossible to modify roles in the HCS Token Contract after the code has been deployed unless a new deployment occurs.
2. **KYC.** KYC status of each account is stored in the HCS Token Contract. This information must be available any time on-chain to support basic token functionalities, such as token transfers. If some HCS Token Contract implementations do not require KYC checks, the checks can be simply removed.
3. **Conformance.** Regarding conformance to standards this semi-formal specification generally follows TTF. It follows ERC20 if there are no matching definitions in TTF. Requirements from the Hedera team have priority over the standards.

# Token Specification

This semi-formal specification is of a descriptive kind; not prescriptive, meaning it specifies the conditions that must hold for each function (before and after execution), but it does *not* prescribe specific steps for how the function should perform computations. In other words, the conditions do *not* manipulate any state directly, but specify how the state must change in a declarative way.

The conditions are expressed in the language of mathematical logic and have the same semantics. The conditions follow similar syntax which is used in modeling languages, such as UML/OCL, Alloy, etc. For example, the symbol equals, =, should be understood as mathematical equality (at a given point in time), not an assignment.

## Fields

**Reading guidelines:** for each field in the HCS Token Contract we specify its name, followed by the colon, followed by field's type. For clarity, we also provide default values that follow the equality symbols. As in Solidity, maps assume default empty values for keys that do not exist. For example, if the map Balances is empty, then Balances[Owner] would return 0 instead of undefined or instead of throwing an exception if the key Owner does not exist in the map.

### HCS Token Contract Fields

- TokenName : String = “”
- TokenSymbol : String = “”
- TokenDecimal : Int = 0
- TotalSupply : Int = 0

- Owner : Address = 0x
- SupplyManager: Address = {}
- AssetProtectionManager: Address = {}
- Balances: Map::Address->Int = {}
- Allowances: Map::Address->(Map::Address->Int) = {}
- Frozen: Map::Address->Bool = {}
- KycPassed: Map::Address->Bool = {}
- ProposedOwner: Address = 0x

# Functions

**Reading guidelines:** functions “operate” over the fields. For each function we specify *logical formulas* that must hold before the function can be executed (pre-conditions) and after it gets executed (post-conditions). If a function gets called and the pre-condition is not satisfied, then the function is assumed to throw an exception and the whole transaction reverts. Anything not mentioned in post-conditions remains unchanged.

For each function, pre- and post-conditions are enumerated in numbered lists. Some typical conditions are:

- equality, e.g., Owner = 0x, meaning that Owner must be equal to 0x,
- inequality, e.g., supplyManager != 0x, meaning that supplyManager must be different from 0x,
- predicates that evaluate to true, e.g., CheckTransferAllowed(addr), meaning that transfer to addr must be allowed,
- predicates that evaluate to false, e.g., !Frozen[addr], meaning that address is not frozen.

To avoid ambiguity, some post-conditions need to distinguish between original and modified values to model value updates explicitly. We construct the updated value name by taking the original value name and following it by the apostrophe symbol (‘). For example, if the original name is TotalSupply and we want to increase the total supply by 10, in the post-condition it would be expressed as: TotalSupply’ = TotalSupply + 10. Intuitively, it works in a similar way as a variable update. Distinguishing the new and updated values by name follows the declarative nature of the specification. If there is no ambiguity, post-conditions simply use the original name.

Where relevant, we specify events. If there are no events specified, then the function does not emit any events. The syntax of events follows that of Ethereum, i.e., event name is followed by a list of arguments enclosed in parentheses.

If a function returns some value, this is expressed as a special post-condition: result = … . Please note that, as with other invariants, this is a logical equality where result is just a name for the return value.

## Base type

- Constructor
    - Signature: void constructor(String tokenName, String tokenSymbol, Int tokenDecimal, Int totalSupply, Address supplyManager, Address assetProtectionManager)
    - Pre-conditions
        i. Owner = 0x // the existing Owner has not been set
        ii. tokenDecimal >= 0
        iii. totalSupply >= 0
        iv. caller != 0x // the newly created Owner is non-zero
        v. supplyManager != 0x
        vi. assetProtectionManager != 0x
    - Post-conditions
        i. TokenName = tokenName
        ii. TokenSymbol = tokenSymbol
        iii. TokenDecimal = tokenDecimal
        iv. TotalSupply = totalSupply
        v. Owner = caller
        vi. SupplyManager = supplyManager
        vii. AssetProtectionManager = assetProtectionManager
        viii. Balances = { SupplyManager->TotalSupply } // SupplyManager gets the TotalSupply of tokens
        ix. Allowances = {}
        x. Frozen = {} // no account is frozen by default
        xi. KycPassed = { Owner->true, SupplyManager->true, AssetProtectionManager->true }
        xii. ProposedOwner = 0x
    - Events
        i. Constructed(tokenName, tokenSymbol, tokenDecimal, totalSupply, supplyManager, assetProtectionManager)

## Getters

- Get token name
    - Signature: String name()
    - Pre-conditions
        i. Owner != 0x
    - Post-conditions
        i. result = TokenName
- Get token symbol
    - Signature: String symbol()
    - Pre-conditions

        i. Owner != 0x
    - Post-conditions
        i. result = TokenSymbol
- Get token decimal
    - Signature: Int decimals()
    - Pre-conditions
        i. Owner != 0x
    - Post-conditions
        i. result = TokenDecimal
- Get total supply
    - Signature: Int totalSupply()
    - Pre-conditions
        i. Owner != 0x
    - Post-conditions
        i. result = TotalSupply
- Get account's balance
    - Signature: Int balanceOf(Address addr)
    - Pre-conditions
        i. Owner != 0x
    - Post-conditions
        i. result = Balances[addr]
- Get account's allowance
    - Signature: Int allowance(Address addr, Address spender)
    - Pre-conditions
        i. Owner != 0x
    - Post-conditions
        i. result = Allowances[addr][spender]
- Get owner
    - Signature: Address owner()
    - Pre-conditions
        i. Owner != 0x
    - Post-conditions
        i. result = Owner
- Get supply manager
    - Signature: Address supplyManager()
    - Pre-conditions
        i. Owner != 0x
    - Post-conditions
        i. result = SupplyManager
- Get asset protection manager
    - Signature: Address assetProtectionManager()
    - Pre-conditions
        i. Owner != 0x

    - Post-conditions
        i. result = AssetProtectionManager
- Get proposed owner
    - Signature: Address proposedOwner()
    - Pre-conditions
        i. Owner != 0x
    - Post-conditions
        i. result = ProposedOwner
- Is account frozen
    - Signature: bool isFrozen(Address addr)
    - Pre-conditions
        i. Owner != 0x
    - Post-conditions
        i. result = Frozen[addr]
- Did the account pass KYC/AML
    - Signature: bool isKycPassed(Address addr)
    - Pre-conditions
        i. Owner != 0x
    - Post-conditions
        i. result = KycPassed[addr]

## Delegable

- Approve
    - Signature: void approveAllowance(Address spender, Int value)
    - TTF Reference: Delegable.ApproveAllowance() [p. 48]
    - Pre-conditions
        i. Owner != 0x
        ii. value >= 0
        iii. CheckTransferAllowed(caller)
        iv. CheckTransferAllowed(spender)
    - Post-conditions
        i. Allowances[caller][spender] = value
    - Events
        i. Approve(caller, spender, value)

## Mintable

- Mint
    - Signature: void mint(Int value)
    - TTF Reference: Mintable.mint() [p. 42]
    - Pre-conditions
        i. Owner != 0x // constructor has been called
        ii. caller = SupplyManager || caller = Owner

        iii. value >= 0
        iv. TotalSupply + value <= MAX_INT // prevents overflow
        v. TotalSupply >= Balances[SupplyManager]
    - Post-conditions
        i. TotalSupply' = TotalSupply + value // the new supply is increased by value
        ii. Balances[SupplyManager]' = Balances[SupplyManager] + value
    - Events
        i. Mint(SupplyManager, value)

## Burnable

- Burn
    - Signature: void burn(Int value)
    - TTF Reference: Burnable.burn() [p. 80]
    - Pre-conditions
        i. Owner != 0x
        ii. caller = SupplyManager || caller = Owner
        iii. value >= 0
        iv. Balances[SupplyManager] >= value
        v. TotalSupply >= Balances[SupplyManager]
    - Post-conditions
        i. TotalSupply' = TotalSupply - value // the new supply is decreased by value
        ii. Balances[SupplyManager]' = Balances[SupplyManager] - value
    - Events
        i. Burn(SupplyManager, value)

## Transferable

- Transfer
    - Signature: void transfer(Address to, Int value)
    - TTF Reference: Transferable.transfer() [p. 38]
    - Pre-conditions
        i. Owner != 0x
        ii. value >= 0
        iii. Balances[caller] >= value
        iv. CheckTransferAllowed(caller)
        v. CheckTransferAllowed(to)
    - Post-conditions
        i. Balances[caller]' = Balances[caller] - value
        ii. Balances[to]' = Balances[to] + value
    - Events
        i. Transfer(caller, to, value)

- TransferFrom
    - Signature: void transferFrom(Address from, Address to, Int value)
    - TTF Reference: Transferable.transferFrom() [p. 39]
    - Pre-conditions
        i. Owner != 0x
        ii. value >= 0
        iii. Balances[from] >= value
        iv. Allowances[from][caller] >= value
        v. CheckTransferAllowed(caller)
        vi. CheckTransferAllowed(from)
        vii. CheckTransferAllowed(to)
    - Post-conditions
        i. Balances[from]' = Balances[from] - value
        ii. Allowances[from][caller]' = Allowances[from][caller] - value
        iii. Balances[to]' = Balances[to] + value
    - Events
        i. Transfer(from, to, value)
        ii. Approve(from, caller, Allowances[from][caller]')

## Roles

- ProposeOwner
    - Signature: void proposeOwner(Address addr)
    - TTF Reference: none
    - Pre-conditions
        i. Owner != 0x
        ii. caller = Owner
        iii. addr != 0x
        iv. CheckTransferAllowed(addr)
    - Post-conditions
        i. ProposedOwner = addr
    - Events
        i. ProposeOwner(addr)
- Claim ownership
    - Signature: void claimOwnership()
    - TTF Reference: none
    - Pre-conditions
        i. Owner != 0x
        ii. caller = ProposedOwner
        iii. CheckTransferAllowed(caller)
    - Post-conditions
        i. Owner = caller
        ii. ProposedOwner = 0x

    - Events
        i. ClaimOwnership(caller)
        ii. ProposeOwner(0x)
- ChangeSupplyManager
    - Signature: void changeSupplyManager(Address addr)
    - TTF Reference: none / Roles
    - Pre-conditions
        i. Owner != 0x
        ii. caller = Owner
        iii. addr != 0x
        iv. CheckTransferAllowed(addr)
    - Post-conditions
        i. SupplyManager = addr
    - Events
        i. ChangeSupplyManager(addr)
- ChangeAssetProtectionManager
    - Signature: void changeAssetProtectionManager(Address addr)
    - TTF Reference: none / Roles
    - Pre-conditions
        i. Owner != 0x
        ii. caller = Owner
        iii. addr != 0x
        iv. CheckTransferAllowed(addr)
    - Post-conditions
        i. AssetProtectionManager = addr
    - Events
        i. ChangeAssetProtectionManager(addr)

## Compliant

- Detect transfer restriction
    - Signature: bool checkTransferAllowed(Address addr)
    - TTF Reference: Compliant.CheckTransferAllowed() [p. 75]
    - Pre-conditions
        i. Owner != 0x
    - Post-conditions
        i. result = (!Frozen[addr] && KycPassed[addr])

## Unclassified

- Freeze
    - Signature: void freeze(Address addr)
    - TTF Reference: none
    - Pre-conditions

        i. Owner != 0x
        ii. caller = AssetProtectionManager || caller = Owner
        iii. !isPrivilegedRole(addr)
    - Post-conditions
        i. Frozen[addr] // upon completion, Frozen[addr] must be true
    - Events
        i. Freeze(addr)
- Unfreeze
    - Signature: void unfreeze(Address addr)
    - TTF Reference: none
    - Pre-conditions
        i. Owner != 0x
        ii. caller = AssetProtectionManager || caller = Owner
    - Post-conditions
        i. !Frozen[addr] // upon completion, Frozen[addr] must be false
    - Events
        i. Unfreeze(addr)
- Wipe
    - Signature: void wipe(Address addr)
    - TTF Reference: none
    - Pre-conditions
        i. Owner != 0x
        ii. caller = AssetProtectionManager || caller = Owner
        iii. Frozen[addr]
    - Post-conditions
        i. TotalSupply' = TotalSupply - Balances[addr] // total supply decreased
        ii. Balances[addr]' = 0 // balance "updated" to 0
    - Events
        i. Wipe(addr, Balances[addr])
- Set KYC passed
    - Signature: void setKycPassed(Address addr)
    - TTF Reference: none
    - Pre-conditions
        i. Owner != 0x
        ii. caller = AssetProtectionManager || caller = Owner
    - Post-conditions
        i. KycPassed[addr]
    - Events
        i. SetKycPassed(addr)
- Unset passed KYC
    - Signature: void unsetKycPassed(Address addr)
    - TTF Reference: none
    - Pre-conditions

        i. Owner != 0x
        ii. caller = AssetProtectionManager || caller = Owner
        iii. !isPrivilegedRole(addr)
    - Post-conditions
        i. !KycPassed[addr]
    - Events
        i. UnsetKycPassed(addr)
- Is privileged role
    - Signature: bool isPrivilegedRole(Address addr)
    - TTF Reference: none
    - Pre-conditions
        i. Owner != 0x
    - Post-conditions
        i. result = (addr = Owner || addr = AssetProtectionManager ||
            addr = SupplyManager)
- Increase allowance
    - Signature: void increaseAllowance(Address spender, Int value)
    - TTF Reference: none
    - Pre-conditions
        i. Owner != 0x
        ii. value >= 0
        iii. CheckTransferAllowed(caller)
        iv. CheckTransferAllowed(spender)
        v. Allowances[caller][spender] + value <= MAX_INT
    - Post-conditions
        i. Allowances[caller][spender]' = Allowances[caller][spender] + value
    - Events
        i. IncreaseAllowance(caller, spender, Allowances[caller][spender]')
- Decrease allowance
    - Signature: void decreaseAllowance(Address spender, Int value)
    - TTF Reference: none
    - Pre-conditions
        i. Owner != 0x
        ii. value >= 0
        iii. CheckTransferAllowed(caller)
        iv. CheckTransferAllowed(spender)
        v. Allowances[caller][spender] >= value
    - Post-conditions
        i. Allowances[caller][spender]' = Allowances[caller][spender] - value
    - Events
        i. DecreaseAllowance(caller, spender, Allowances[caller][spender]')

# Formal Specification Guidelines

Formal specifications may make some simplifying assumptions or model certain constructions explicitly. Below we list a few of these:

1. **String values.** Fields and values of type String may be modeled as integers for performance reasons. The only relevant operation is equality or inequality check.
2. **Caller.** Unlike in blockchain applications, the caller may need to be passed explicitly as a function argument.
3. **Constructor.** Not all languages provide native support for constructors. As such, the specification includes the function constructor() which checks if the Owner is non-zero. A non-zero Owner means that the HCS token contract has already been constructed.
4. **Exceptions.** Not all languages provide native support for exceptions. As such, a formal specification may need to include a value ‘throw’ to model invalid function execution.

# Guidelines for Implementers

This semi-formal specification may be too explicit in some respects (e.g., conditions to call a function), whereas too inspecific in others (e.g., the handling of events) since there may be multiple ways of implementing a given functionality. Below we provide some guidelines which may be useful for implementers:

1. **Synchronicity.** We assume synchronous execution of HCS Token Contract functions on the distributed ledger, i.e., as in executing a function by a miner and then adding its trace to the ledger. From the point of view of the developer-facing API, however, it may be more practical to work with an asynchronous API since network operations (and achieving finality on the ledger) may take relatively significant time. We recommend following this model since it is considered a standard in distributed ledgers (e.g., Ethereum, EOS, IOTA). In this model developer-facing API is asynchronous and reacts to data that get added to the ledger. As such, it requires connection either to a local or remote client.
2. **Nonce.** Depending on the semantics of the underlying distributed ledger, if it does not prevent message and transaction replays, these could be prevented by adding a nonce. Nonce is typically a number per user account and it is incremented whenever a new transaction/message is sent. Transactions/messages which use a previously used number should be rejected.
3. **Events.** Events are used to log data to the blockchain and to communicate data to clients. The semi-formal specification provides event names and arguments albeit they are largely immaterial from the point of view of formal verification. A specific way of logging and broadcasting events should be defined in a separate specification for developers. It would depend on the underlying DLT and the developer-facing API. If necessary, events may additionally carry the nonces.

4. **Return types.** State-modifying HCS Token Contract functions either throw an exception (upon failure) or log an event (upon success). These functions do not normally return values. In case when some pre-conditions can be quickly checked synchronously (e.g., if values are non-zero), one could develop synchronous function wrappers which do not require a DLT query to fail. Such wrappers could return Boolean or enum values to signal these quick failures.
5. **Constructor.** Many modern languages, e.g., Java, C++ provide native support for constructors. As such, conditions that check if an object has been constructed (by checking if Owner is non-zero) may be omitted in the actual implementation.
6. **Addresses.** The semi-formal specification makes no assumptions about how accounts addresses are generated. Developer's specification should make this more precise. Specific details should depend on the underlying DLT.
7. **Maps.** We assumed that: 1) uninitialized maps are empty maps (i.e., maps with no entries), and that 2) if a key does not exist in the map, then a default value is returned (e.g., if a key 0xACC does not exist in the map Frozen, then Frozen[0xACC] returns false). If the underlying language does not follow such a semantics, developers should first check if the key exists, and if it doesn't, they should insert (and return) the default value. This could be done either per function that modifies a map, or there could be a single method like `initializeAccount(Address account)` which would initialize an account before it can be used.
8. **Nulls.** If the implementation language distinguishes between null/undefined/uninitialized and empty (default-initialized) values, then nulls need to be checked for and taken care of appropriately.
9. **Assignments.** The semi-formal specification uses names of variables followed by apostrophes due to the declarative nature of the specification. In the actual implementation in an imperative language (e.g., Java) one can think of operations over these variables as of assignments (i.e., modifications of variables' values).
10. **Auditability.** If there is a need for auditability of transactions (e.g., to track transfers that lead to burns) this kind of information could be derived by analyzing the ledger by going through transactions/messages, their originators, nonces, and events.
11. **Roles.** As a design choice for the specification, there are four separate roles hardcoded into the HCS Token Contract. Depending on specific requirements for how the token should be managed, roles could be collapsed (to have fewer actors controlling various functionalities of the token), or extended (to allow more actors control the functionalities). Below are some design choices regarding roles:
    a. Only one user per role type (as it is proposed in the spec now),
    b. Multiple users per role type, where a multi-sig is used (e.g., multiple users need to agree to mint new tokens),
    c. Multiple users who are assigned separate roles of a given type (in the sense of instances of a specific role type, e.g., multiple supply managers)