Summary

This proposal defines the smart contract interface necessary to implement the Tezos Decentralized Identifier (DID) Manager, which is used by the Tezos DID Method for on-chain establishment of DPKI and service discovery.

Abstract

The Tezos Decentralized Identifier (DID) Manager is used to establish decentralized public key infrastructure (DPKI) and service discovery for Tezos-based DIDs on-chain. Tezos-based DIDs allow any Tezos accounts, irrespective of their transaction history, to leverage compatible standards and protocols within the decentralized identity ecosystem, such as W3C Verifiable Credentials (VCs). All Tezos accounts start with implied DIDs, which require no transactions to use. This specification defines a smart contract interface which must be used in conjunction with implicit DID documents and off-chain updates to complete DID resolution as described in the Tezos DID Method.

TZIP-19 defines:

- Michelson annotations necessary to implement a conforming DID Manager which supports rotation of a single verification method and a single service endpoint.
- Contract metadata (TZIP-16) for off-chain resolution updates.
- Further functional requirements for conformity including signature checks during rotation.

Motivation

For users to control their DIDs in a self-custodial manner without intermediaries, they may consider DID methods which utilize a public blockchain such as the mainnet instance of the Tezos blockchain protocol. This approach is also useful in a consortium setting in which a blockchain shared by the authorized members who wish to broker direct peer-to-peer communications or exchange verifiable off-chain information.

For additional background, please consider DID Use Cases, VC Use Cases, and the DIDComm Messaging Specification.

Specification

Entrypoints and Functional Requirements

A smart contract conforming to TZIP-19 must implement the following entrypoints conforming to the specified functional requirements.

`rotate_authentication`

```
type verification_method = address;

type rotation_event = {
    public_key           : key,
    current_value_digest : bytes,
    next_value_digest    : bytes,
```

```
    current_chain        : bytes,
    rotation_count       : nat
};

let rotate_authentication : (verification_method, rotation_event, signature,
storage) => storage;
```

Functional requirements:

- The result of calling `get_authentication` is equivalent to the result of `HASH_KEY` applied to `rotation_event.public_key`.
- The target of the `signature` is the result of `PACK` applied on `rotation_event`.

`rotate_service`

```
type service = {
    type             : string,
    service_endpoint : string
};

// rotation_event is reused from the rotate_authentication section.
let rotate_service : (service, rotation_event, signature, storage) => storage;
```

Functional requirements:

- The result of calling `get_authentication` is equivalent to the result of `HASH_KEY` applied to `rotation_event.public_key`.
- The target of the `signature` is the result of `PACK` applied on `rotation_event`.

Metadata (off-chain views)

In order to actually retrieve the service endpoint and the verification method from the DID Manager, we need some form of getter. Because a Michelson smart contract cannot, at the moment, return a value -- and because we do not want to peak directly at the storage of the DID Manager -- TZIP-19 makes use of contract metadata (defined in TZIP-16).

```
{
  "name":"DID Manager",
  "interfaces":[
    "TZIP-19"
  ],
  "views":[
    {
      "implementations":[
        {
          "michelsonStorageView":{
            "annotations":[ ],
            "returnType":{
```

```
              "prim":"pair",
              "args":[
                {"prim": "string", "annots": ["%service_endpoint"]},
                {"prim": "string", "annots": ["%type"]}
              ]
            },
            "code":[
              {"prim": "CDR"},
              {"prim": "CAR"},
              {"prim": "CDR"},
              {"prim": "CDR"}
            ],
            "parameter":{
              "prim": "unit"
            }
          }
        }
      ],
      "name":"GetService",
      "pure":true
    },
    {
      "implementations":[
        {
          "michelsonStorageView":{
            "annotations":[ ],
            "returnType":{
              "prim":"address",
              "args":[ ],
              "annots": ["%verification_method"]
            },
            "code":[
              {"prim": "CDR"},
              {"prim": "CDR"}
            ],
            "parameter":{
              "prim": "unit"
            }
          }
        }
      ],
      "name":"GetVerificationMethod",
      "pure":true
    }
  ]
}
```

> The code for each view is provided as examples and is not part of the specifications.

Rationale

Backwards Compatibility