Cardano on-the-wire specification

Nicholas Clarke

February 28, 2023

Contents

1 Introduction

This document describes the binary serialisation formats used for the Cardano blockchain.

It proceeds as follows: section 2 describes the core types which must be serialised for the purposes of Cardano communication. We then detail the requirements on the binary format in 3. In section 4 we describe the explicit binary serialisation of the core types satisfying these requirements.

Appendix A details the current binary format.

1.1 Notation

Lists Lists of type A are written as $[A]$.

Non-empty lists Non-empty lists of type A are written as $[A]^+$.

Sets Sets of type A are written as $\{A\}$.

Maps A map from items of type A to items of type B is written as $A \rightharpoonup B$.[1]

Records We denote records as:

```
⟨record⟩    = {
            ,  fieldName :: type
            ,  fieldName2 :: type2 }
```

Variants We denote variants as:

```
⟨variant⟩   = Option1
            |  Option2
            |  Option3.
```

Optional values We denote an optional value of type A as $()|A$

We will use capitalised words for variant constructors and lower-case terms for field names, as per standard Haskell syntax.

1.2 Relation to the abstract specification and to the Cardano implementation

This document straddles the middle-ground between the rule-based specification given in [1] and the concrete implementation at [2]. As such, it tries to compromise between the differing presentations of how the blockchain is constructed.

In general, we take the following approach:

- Where possible, we take our nomenclature from [1]. So, for example, we refer to the type of transactions as $\mathcal{T}$ rather than **Tx**. Where there is no name given in [1] we generally invent a new identifier in the similar style.
- Structure, on the other hand, typically derives from the code in [2], with some exceptions:
  - We drop `newtype` wrappers.
  - We flatten nested records where doing so would not lead to significant confusion (for example, by requiring convoluted or length names to contextualise fields).
  - In general, we tend to be fairly free with translating between isomorphic representations where it increases the readability of this spec.
  - We monomorphize any polymorphic variables to the type they are instantiated to when they are serialised.
  - We ignore `AsBinary` and its ilk. These provide 'typed' wrappers around `ByteString` in order to defer encoding/decoding. They only confuse an abstract definition.

2 Core datatypes

The types in this section are derived from a combination of the blockchain spec and Haskell datatypes in the codebase.

[1] We use this terminology for consistency with [1], where these are partial functions.

2.1 Primitives

Primitive types:

$\mathbb{B}$ Boolean value

Word8 8-bit word

Word32 32-bit word

Word64 64-bit word

String Arbitrary UTF8-encoded string type. We do not draw any distinctions as to whether this is implemented as a **String** or **Text**.

Bytes Arbitrary string of bytes.

$\mathbb{I}$ arbitrary precision integers.

2.2 Basic Cardano Types

We start with sets of identifiers. These are represented in code as Blake2b-256 hashes. We also sometimes deal with these hashes directly.

- Transaction identifiers $\mathcal{I}_{\mathcal{T}}$
- Block identifiers $\mathcal{I}_{\mathcal{B}}$
- Update identifiers $\mathcal{I}_{\mathcal{UP}}$
- Concrete hashes **Hash**

We also have addresses, which are represented as Blake2b-224 hashes.

- Address (agent) identifiers $\mathcal{I}_{\mathcal{A}}$
- Stakeholder identifiers $\mathcal{I}_{\mathcal{K}}$

Epochs are identified by their index as a **Word64**. Slots are identified as a pair of epoch identifier and a **Word16** index within the slot.

- Epoch indices $\mathcal{I}_{\mathcal{E}}$
- Slot indices $\mathcal{I}_{\mathcal{S}}$

Cardano public keys are elliptic curve DH keys on curve 25519.

- Public keys $\chi_{\mathbf{pub}}$
- Signatures $\chi_{\mathbf{sig}}$

2.3 Addresses

We have a set of addresses $\mathcal{A}$. An address has an identifier, a type, and some additional data held in an `attributes` field. In particular, each address contains a stake distribution $\mathcal{A}_{\mathbf{distr}}$.

$$
\begin{array}{lll}
\langle \mathcal{A}_{distr} \rangle & = & \text{Bootstrap} \\
& | & \text{SingleKey } \mathcal{I}_{\mathcal{A}} \\
& | & \text{MultiKey } \mathcal{I}_{\mathcal{A}} \rightharpoonup \mathbf{Word64}.
\end{array}
$$

$$
\begin{array}{lll}
\langle \mathcal{A}_{type} \rangle & = & \text{PubKey} \\
& | & \text{Script} \\
& | & \text{Redeem} \\
& | & \text{Unknown } \mathbf{Word8}.
\end{array}
$$

$$
\begin{array}{lll}
\langle \mathcal{A}_{attr} \rangle & = & \{ \\
& , & \text{pkDerivationPath} :: ()|\mathbf{Bytes} \\
& , & \text{stakeDistr} :: \mathcal{A}_{\mathbf{distr}} \\
& , & \text{unparsed} :: \mathbf{Word8} \rightharpoonup \mathbf{Bytes} \ \}.
\end{array}
$$

$$
\begin{array}{lll}
\langle \mathcal{A} \rangle & = & \{ \\
& , & \text{root} :: \mathcal{I}_{\mathcal{A}} \\
& , & \text{attributes} :: \mathcal{A}_{\mathbf{attr}} \\
& , & \text{type} :: \mathcal{A}_{\mathbf{type}} \ \}.
\end{array}
$$

Figure 1: Address Types

2.4 Transactions

We have a set of transactions $\mathcal{T}$.

$$
\begin{array}{lll}
\langle \mathcal{T}_{in} \rangle & = & \text{Valid } \mathcal{I}_{\mathcal{T}} \times \mathbf{Word32}. \\
& | & \text{Invalid } \mathbf{Word8} \times \mathbf{Bytes}.
\end{array}
$$

$$
\begin{array}{lll}
\langle \mathcal{T} \rangle & = & \{ \\
& , & \text{inputs} :: [\mathcal{T}_{\mathbf{in}}] \\
& , & \text{outputs} :: [\mathcal{I}_{\mathcal{A}} \times \mathbf{Word64}]^{+} \ \}
\end{array}
$$

$$
\begin{array}{lll}
\langle \mathcal{T}_{proof} \rangle & = & \{ \\
& , & \text{txCount} :: \mathbf{Word32} \\
& , & \text{merkleRoot} :: \mathbf{Hash} \\
& , & \text{witnessHash} :: \mathbf{Hash} \ \}
\end{array}
$$

Figure 2: Transactions

We also have transaction witnesses, which provide proof that the transaction has authority to spend its inputs.

$$
\begin{array}{lll}
\langle \mathcal{TW}_{pk} \rangle & = & \{ \\
 & , & \text{key} :: \chi_{\mathbf{pub}} \\
 & , & \text{signature} :: \chi_{\mathbf{sig}} \ \} \\
 & & \\
\langle \mathcal{TW}_{script} \rangle & = & \{ \\
 & , & \text{validator} :: \mathbf{Word16} \times \mathbf{Bytes} \\
 & , & \text{redeemer} :: \mathbf{Word16} \times \mathbf{Bytes} \ \} \\
 & & \\
\langle \mathcal{TW}_{redeem} \rangle & = & \{ \\
 & , & \text{key} :: \chi_{\mathbf{pub}} \\
 & , & \text{signature} :: \chi_{\mathbf{sig}} \ \} \\
 & & \\
\langle \mathcal{TW} \rangle & = & \text{PublicKey } \mathcal{TW}_{\mathbf{pk}} \\
 & | & \text{Script } \mathcal{TW}_{\mathbf{script}} \\
 & | & \text{Redeem } \mathcal{TW}_{\mathbf{redeem}} \\
 & | & \text{Unknown } \mathbf{Word8\ Bytes}
\end{array}
$$

Figure 3: Transaction witnesses

2.5 Shared Seed Computation

Shared seed computation deals with commitments, openings, shares and certificates.

We start with some basic types used for VSS. VSS uses its own public key cryptography scheme.

- VSS public keys $\mathcal{VSS}_{\chi_{\mathbf{pub}}}$
- VSS secret $\mathcal{VSS}_{\mathbf{sec}}$
- VSS encrypted and decrypted shares $\mathcal{VSS}_{\mathbf{enc}}$ and $\mathcal{VSS}_{\mathbf{dec}}$
- VSS secret proof $\mathcal{VSS}_{\mathbf{proof}}$

$$
\begin{array}{lcl}
\langle \mathcal{SSC}_{comm} \rangle & = & \chi_{\mathbf{pub}} \times ((\mathcal{VSS}_{\chi_{\mathbf{pub}}} \rightharpoonup \mathcal{VSS}_{\mathbf{enc}}) \times \mathcal{VSS}_{\mathbf{proof}}) \times \chi_{\mathbf{sig}} \\
\\
\langle \mathcal{SSC}_{open} \rangle & = & \mathcal{I}_{\mathcal{K}} \rightharpoonup \mathcal{VSS}_{\mathbf{sec}} \\
\\
\langle \mathcal{SSC}_{share} \rangle & = & \mathcal{I}_{\mathcal{K}} \rightharpoonup (\mathcal{I}_{\mathcal{K}} \rightharpoonup [\mathcal{VSS}_{\mathbf{dec}}]^{+}) \\
\\
\langle \mathcal{SSC}_{cert} \rangle & = & \{ \\
 & , & \text{vsskey} :: \mathcal{VSS}_{\chi_{\mathbf{pub}}} \\
 & , & \text{signingKey} :: \chi_{\mathbf{pub}} \\
 & , & \text{expiry} :: \mathcal{I}_{\mathcal{E}} \\
 & , & \text{signature} :: \chi_{\mathbf{sig}} \ \} \\
\\
\langle \mathcal{SSC} \rangle & = & \text{Commitments } [\mathcal{SSC}_{\mathbf{comm}}] \times [\mathcal{SSC}_{\mathbf{cert}}] \\
 & | & \text{Openings } \mathcal{SSC}_{\mathbf{open}} \times [\mathcal{SSC}_{\mathbf{cert}}] \\
 & | & \text{Shares } \mathcal{SSC}_{\mathbf{share}} \times [\mathcal{SSC}_{\mathbf{cert}}] \\
 & | & \text{Certificates } [\mathcal{SSC}_{\mathbf{cert}}] \\
\\
\langle \mathcal{SSC}_{proof} \rangle & = & \text{CommitmentsProof } \mathbf{Hash} \times \mathbf{Hash} \\
 & | & \text{OpeningsProof } \mathbf{Hash} \times \mathbf{Hash} \\
 & | & \text{SharesProof } \mathbf{Hash} \times \mathbf{Hash} \\
 & | & \text{CertificatesProof } \mathbf{Hash}
\end{array}
$$

Figure 4: Shared seed computation

2.6 Delegation

The heavyweight delegation system is used to allow stakeholders to authorise other parties to issue blocks on their behalf.

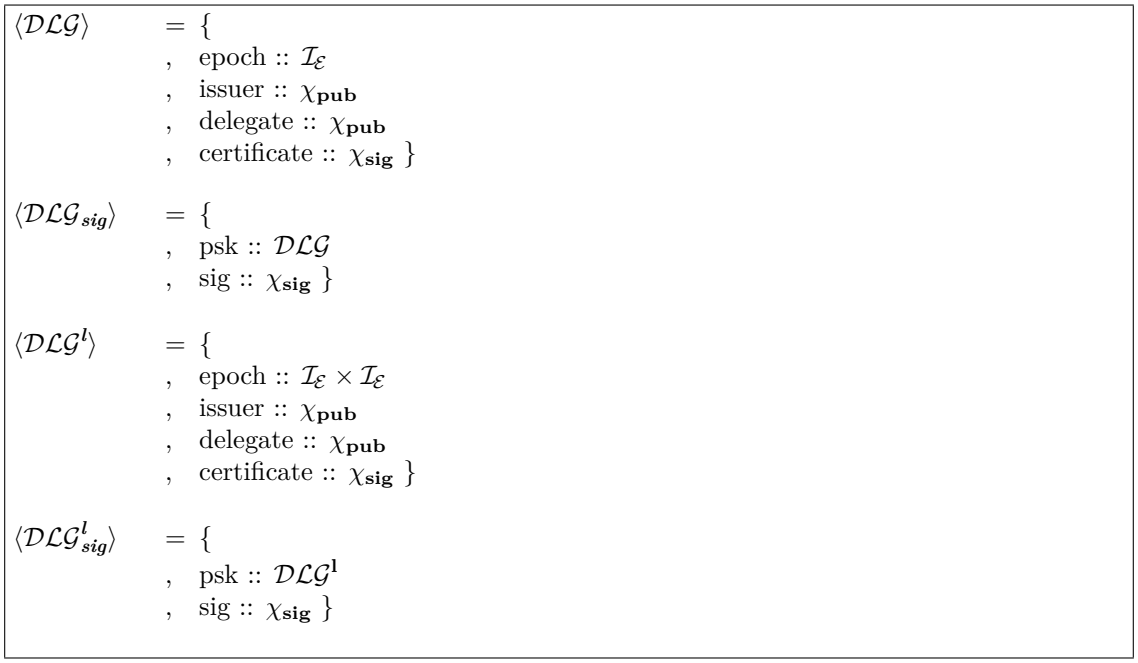

Figure 5: Delegation

2.7 Updates

This section covers the types used to orchestrate updates of the system. We also introduce the block version and transaction fee policy in this section, since they are only serialised as part of the update system.

$$
\begin{array}{lll}
\langle \mathcal{B}_{ver} \rangle & = & \mathbf{Word16} \times \mathbf{Word16} \times \mathbf{Word8} \\
\\
\langle \mathcal{T}_{feepol} \rangle & = & \text{Linear } \mathbb{I}\ \mathbb{I} \\
 & | & \text{Unknown } \mathbf{Word8}\ \mathbf{Bytes} \\
\\
\langle \mathcal{B}_{ver\Delta} \rangle & = & \{ \\
 & , & \text{scriptVersion} :: ()|\mathbf{Word16} \\
 & , & \text{slotDuration} :: ()|\mathbb{I} \\
 & , & \text{maxBlockSize} :: ()|\mathbb{I} \\
 & , & \text{maxHeaderSize} :: ()|\mathbb{I} \\
 & , & \text{maxTxSize} :: ()|\mathbb{I} \\
 & , & \text{maxProposalSize} :: ()|\mathbb{I} \\
 & , & \text{mpcThd} :: ()|\mathbf{Word64} \\
 & , & \text{heavyDelThd} :: ()|\mathbf{Word64} \\
 & , & \text{updateVoteThd} :: ()|\mathbf{Word64} \\
 & , & \text{updateProposalThd} :: ()|\mathbf{Word64} \\
 & , & \text{updateImplicit} :: ()|\mathbf{Word64} \\
 & , & \text{softforkRule} :: ()|(\mathbf{Word6}4 \times \mathbf{Word6}4 \times \mathbf{Word6}4) \\
 & , & \text{txFeePolicy} :: ()|\mathcal{T}_{\mathbf{feepol}} \\
 & , & \text{unlockStakeEpoch} :: ()|\mathcal{I}_{\mathcal{E}}\ \} \\
\\
\langle \mathcal{UP}_{data} \rangle & = & \{ \\
 & , & \text{appDiffHash} :: \mathbf{Hash} \\
 & , & \text{pkgHash} :: \mathbf{Hash} \\
 & , & \text{updaterHash} :: \mathbf{Hash} \\
 & , & \text{mdHash} :: \mathbf{Hash}\} \\
\\
\langle \mathcal{UP}_{prop} \rangle & = & \{ \\
 & , & \text{blockVersion} :: \mathcal{B}_{\mathbf{ver}} \\
 & , & \text{blockVersionMod} :: \mathcal{B}_{\mathbf{ver\Delta}} \\
 & , & \text{softwareVersion} :: \mathbf{String} \times \mathbf{Word32} \\
 & , & \text{data} :: \mathbf{String} \rightharpoonup \mathcal{UP}_{\mathbf{data}} \\
 & , & \text{attributes} :: \mathbf{Word8} \rightharpoonup \mathbf{Bytes} \\
 & , & \text{from} :: \chi_{\mathbf{pub}} \\
 & , & \text{signature} :: \chi_{\mathbf{sig}}\ \} \\
\\
\langle \mathcal{UP}_{vote} \rangle & = & \{ \\
 & , & \text{voter} :: \chi_{\mathbf{pub}} \\
 & , & \text{proposalId} :: \mathcal{I}_{\mathcal{UP}} \\
 & , & \text{vote} :: \mathbb{B} \\
 & , & \text{signature} :: \chi_{\mathbf{sig}}\ \} \\
\\
\langle \mathcal{UP} \rangle & = & \{ \\
 & , & \text{proposal} :: ()|\mathcal{UP}_{\mathbf{prop}} \\
 & , & \text{votes} :: [\mathcal{UP}_{\mathbf{vote}}]\ \}
\end{array}
$$

Figure 6: Updates

2.8 Blocks

A block consists of a block header and a block body. The block header consists of verification for the various components in the block body.

$$
\begin{array}{lll}
\langle \mathcal{B}_{sig} \rangle & = & \text{Signature } \chi_{\mathbf{sig}} \\
& | & \text{ProxySigLight } \mathcal{DLG}^{\mathbf{l}}_{\mathbf{sig}} \\
& | & \text{ProxySigHeavy } \mathcal{DLG}_{\mathbf{sig}} \\
\\
\langle \mathcal{B}_{cons} \rangle & = & \{ \\
& , & \text{slotId} :: \mathcal{I}_{\mathcal{S}} \\
& , & \text{leaderKey} :: \chi_{\mathbf{pub}} \\
& , & \text{difficulty} :: \mathbf{Word64} \\
& , & \text{signature} :: \mathcal{B}_{\mathbf{sig}} \ \} \\
\\
\langle \mathcal{B}_{headex} \rangle & = & \{ \\
& , & \text{blockVersion} :: \mathcal{B}_{\mathbf{ver}} \\
& , & \text{softwareVersion} :: \mathbf{String} \times \mathbf{Word32} \\
& , & \text{extraProof} :: \mathbf{Hash} \\
& , & \text{attributes} :: \mathbf{String} \rightharpoonup \emptyset \ \} \\
\\
\langle \mathcal{B}_{proof} \rangle & = & \{ \\
& , & \text{txProof} :: \mathcal{T}_{\mathbf{proof}} \\
& , & \text{sscProof} :: \mathcal{SSC}_{\mathbf{proof}} \\
& , & \text{dlgProof} :: \mathbf{Hash} \\
& , & \text{updProof} :: \mathbf{Hash} \ \} \\
\\
\langle \mathcal{B}_{head} \rangle & = & \{ \\
& , & \text{prevBlock} :: \mathcal{I}_{\mathcal{B}} \\
& , & \text{bodyProof} :: \mathcal{B}_{\mathbf{proof}} \\
& , & \text{consensusData} :: \mathcal{B}_{\mathbf{cons}} \\
& , & \text{extraData} :: \mathcal{B}_{\mathbf{headex}} \ \} \\
\\
\langle \mathcal{B}_{body} \rangle & = & \{ \\
& , & \text{txPayload} :: [\mathcal{T} \times [\mathcal{TW}]] \\
& , & \text{sscPayload} :: \mathcal{SSC} \\
& , & \text{dlgPayload} ::[\mathcal{DLG}] \\
& , & \text{updPayload} :: \mathcal{UP} \ \} \\
\\
\langle \mathcal{B} \rangle & = & \{ \\
& , & \text{header} :: \mathcal{B}_{\mathbf{head}} \\
& , & \text{body} :: \mathcal{B}_{\mathbf{body}} \\
& , & \text{extra} :: \mathbf{Word8} \rightharpoonup \mathbf{Bytes} \ \}
\end{array}
$$

Figure 7: Blocks

2.8.1 Epoch Boundary Blocks

In addition to regular blocks, epoch boundary blocks contain a list of slot leaders for a given epoch. They are not conventionally distributed as part of the blockchain, but can be requested as part of catch-up and as such form part of the on-the-wire protocol.

$$
\begin{array}{ll}
\langle \mathcal{B}'_{cons} \rangle & = \{ \\
& , \ \text{epoch} :: \mathcal{I}_{\mathcal{E}} \\
& , \ \text{chainDifficulty} :: \mathbf{Word64} \ \} \\
 & \\
\langle \mathcal{B}'_{head} \rangle & = \{ \\
& , \ \text{prevBlock} :: \mathcal{I}_{\mathcal{B}} \\
& , \ \text{bodyProof} :: \mathbf{Hash} \\
& , \ \text{consensusData} :: \mathcal{B}'_{\mathbf{cons}} \\
& , \ \text{extraData} :: \mathbf{Word8} \rightharpoonup \mathbf{Bytes} \ \} \\
 & \\
\langle \mathcal{B}' \rangle & = \{ \\
& , \ \text{header} :: \mathcal{B}'_{\mathbf{head}} \\
& , \ \text{body} :: [\mathcal{I}_{\mathcal{K}}]^{+} \\
& , \ \text{extra} :: \mathbf{Word8} \rightharpoonup \mathbf{Bytes} \ \}
\end{array}
$$

Figure 8: Epoch Boundary Blocks

3 Requirements on the binary format

3.1 Cryptographic properties

3.2 Dependencies on the binary format

4 Binary specification

References

[1] *Rule-based specification of the blockchain logic*. Erik de Castro Lopo, Nicholas Clarke & Arnaud Spiwack.

[2] *Cryptographic currency implementing Ouroboros PoS protocol*. `https://github.com/input-output-hk/cardano-sl/`

[3] *RFC 7049 Concise Binary Object Representation*. `http://cbor.io`

[4] *Concise data definition language (CDDL): a notational convention to express CBOR data structures* `https://tools.ietf.org/html/draft-ietf-cbor-cddl-00`

A Current binary format

This section documents the current binary serialisation format, as of 2018-06-03.

The current blockchain is serialized using CBOR[3]. Consequently we present the current description as a CDDL[4] document. Terms in use below should be interpreted in that context

and with reference to Appendix E of [4], which defines a standard prelude available to such things.

```
; Cardano Byron blockchain CBOR schema

block = [0, ebblock]
      / [1, mainblock]

mainblock = [ "header" : blockhead
            , "body" : blockbody
            , "extra" : [attributes]
            ]

ebblock = [ "header" : ebbhead
          , "body" : [+ stakeholderid]
          , extra : [attributes]
          ]

u8 = uint .lt 256
u16 = uint .lt 65536
u32 = uint
u64 = uint

; Basic Cardano Types

blake2b-256 = bytes .size 32

txid = blake2b-256
blockid = blake2b-256
updid = blake2b-256
hash = blake2b-256

blake2b-224 = bytes .size 28

addressid = blake2b-224
stakeholderid = blake2b-224

epochid = u64
slotid = [ epoch: epochid, slot : u64 ]

pubkey = bytes
signature = bytes

; Attributes - at the moment we do not bother deserialising these, since they
; don't contain anything

attributes = {* any => any}

; Addresses
```

```
addrdistr = [1] / [0, stakeholderid]

addrtype = &("PubKey" : 0, "Script" : 1, "Redeem" : 2) / (u64 .gt 2)
addrattr = { ? 0 : addrdistr
           , ? 1 : bytes}
address = [ #6.24(bytes .cbor ([addressid, addrattr, addrtype])), u64 ]

; Transactions

txin = [0, #6.24(bytes .cbor ([txid, u32]))] / [u8 .ne 0, encoded-cbor]
txout = [address, u64]

tx = [[+ txin], [+ txout], attributes]

txproof = [u32, hash, hash]

twit = [0, #6.24(bytes .cbor ([pubkey, signature]))]
     / [1, #6.24(bytes .cbor ([[u16, bytes], [u16, bytes]]))]
     / [2, #6.24(bytes .cbor ([pubkey, signature]))]
     / [u8 .gt 2, encoded-cbor]

; Shared Seed Computation

vsspubkey = bytes ; This is encoded using the 'Binary' instance
                  ; for Scrape.PublicKey
vsssec = bytes ; This is encoded using the 'Binary' instance
               ; for Scrape.Secret.
vssenc = [bytes] ; This is encoded using the 'Binary' instance
                 ; for Scrape.EncryptedSi.
                 ; TODO work out why this seems to be in a length 1 array
vssdec = bytes ; This is encoded using the 'Binary' instance
               ; for Scrape.DecryptedShare
vssproof = [bytes, bytes, bytes, [* bytes]] ; This is encoded using the
                                            ; 'Binary' instance for Scrape.Proof

ssccomm = [pubkey, [{vsspubkey => vssenc},vssproof], signature]
ssccomms = #6.258([* ssccomm])

sscopens = {stakeholderid => vsssec}

sscshares = {addressid => [addressid, [* vssdec]]}

ssccert = [vsspubkey, pubkey, epochid, signature]
ssccerts = #6.258([* ssccert])

ssc = [0, ssccomms, ssccerts]
    / [1, sscopens, ssccerts]
    / [2, sscshares, ssccerts]
    / [3, ssccerts]
```

```
sscproof = [0, hash, hash]
         / [1, hash, hash]
         / [2, hash, hash]
         / [3, hash]

; Delegation

dlg = [ epoch : epochid
      , issuer : pubkey
      , delegate : pubkey
      , certificate : signature
      ]

dlgsig = [dlg, signature]

lwdlg = [ epochRange : [epochid, epochid]
        , issuer : pubkey
        , delegate : pubkey
        , certificate : signature
        ]

lwdlgsig = [lwdlg, signature]

; Updates

bver = [u16, u16, u8]

txfeepol = [0, #6.24(bytes .cbor ([bigint, bigint]))]
         / [u8 .gt 0, encoded-cbor]

bvermod = [ scriptVersion : [? u16]
          , slotDuration : [? bigint]
          , maxBlockSize : [? bigint]
          , maxHeaderSize  : [? bigint]
          , maxTxSize : [? bigint]
          , maxProposalSize : [? bigint]
          , mpcThd : [? u64]
          , heavyDelThd : [? u64]
          , updateVoteThd : [? u64]
          , updateProposalThd : [? u64]
          , updateImplicit : [? u64]
          , softForkRule : [? [u64, u64, u64]]
          , txFeePolicy : [? txfeepol]
          , unlockStakeEpoch : [? epochid]
          ]

updata = [ hash, hash, hash, hash ]
```

```
upprop = [ "blockVersion" : bver
         , "blockVersionMod" : bvermod
         , "softwareVersion" : [ text, u32 ]
         , "data" : #6.258([text, updata])
         , "attributes" : attributes
         , "from" : pubkey
         , "signature" : signature
         ]

upvote = [ "voter" : pubkey
         , "proposalId" : updid
         , "vote" : bool
         , "signature" : signature
         ]

up = [ "proposal" :  [? upprop]
     , votes : [* upvote]
     ]

; Blocks

difficulty = [u64]

blocksig = [0, signature]
         / [1, lwdlgsig]
         / [2, dlgsig]

blockcons = [slotid, pubkey, difficulty, blocksig]

blockheadex = [ "blockVersion" : bver
              , "softwareVersion" : [ text, u32 ]
              , "attributes" : attributes
              , "extraProof" : hash
              ]

blockproof = [ "txProof" : txproof
             , "sscProof" : sscproof
             , "dlgProof" : hash
             , "updProof" : hash
             ]

blockhead = [ "protocolMagic" : u32
            , "prevBlock" : blockid
            , "bodyProof" : blockproof
            , "consensusData" : blockcons
            , "extraData" : blockheadex
            ]

blockbody = [ "txPayload" : [* [tx, [* twit]]]
```

```
              , "sscPayload" : ssc
              , "dlgPayload" : [* dlg]
              , "updPayload" : up
              ]

; Epoch Boundary Blocks

ebbcons = [ epochid, difficulty ]

ebbhead = [ "protocolMagic" : u32
          , "prevBlock" : blockid
          , "bodyProof" : hash
          , "consensusData" : ebbcons
          , "extraData" : [attributes]
          ]
```