# **ORC Yellow Paper: Technical Specification for Bitcoin Asset Protocols**

**Version: 1.0**

**Authors: Jack Liao**

**Date: May 13, 2025**

**Hosted Platform: opreturncomment.com**

## **Abstract**

This yellow paper presents the technical specifications for the ORC20 and ORC721 protocols, which enable the creation and management of fungible tokens and non-fungible tokens (NFTs) on the Bitcoin blockchain. Utilizing Bitcoin's OP\_RETURN opcode for data storage, these protocols avoid UTXO bloat while providing robust token functionality. Unlike Ordinals and Runes, which rely on UTXO attachments for data, the ORC protocols store all token data in unspendable outputs, creating a clean separation between Bitcoin's monetary system and asset metadata. This paper outlines the protocol specifications, implementation details, indexing requirements, and integration pathways for Lightning Network compatibility, establishing a foundation for Bitcoin-based tokenization that aligns with Bitcoin's design philosophy while enabling new use cases.

## **Table of Contents**

1. Introduction

2. Protocol Design Principles

3. ORC20: Fungible Token Standard

4. ORC721: Non-Fungible Token Standard

5. Transaction Structure and Encoding

6. Indexing System Architecture

7. Lightning Network Integration

8. Compatibility with Existing Standards

9. Scaling Considerations

10. Security Analysis

11. Implementation Roadmap

12. Conclusion and Future Work

13. References

14. Appendices

## 

## **1. Introduction**

### **1.1 Background**

Bitcoin's permissionless, decentralized design has established it as the world's most secure blockchain. However, its scripting limitations have historically constrained more complex use cases, such as tokenization. The introduction of OP\_RETURN in 2014 (BIP 0080) provided a standardized method for embedding small amounts of data in transactions, enabling metadata storage without affecting the UTXO set.

Multiple approaches to tokenization on Bitcoin have emerged, including:

● **Colored Coins** (2012-2014): Early attempts to "color" specific bitcoins to represent assets

● **Counterparty** (2014): A protocol using OP\_RETURN for asset creation and transfers

● **Omni Layer** (formerly Mastercoin): A platform enabling custom assets on Bitcoin

● **Ordinals** (2023): A system for inscribing data directly into satoshis

● **Runes** (2023): A fungible token protocol using OP\_RETURN with UTXO references

Each approach has made trade-offs between functionality, efficiency, and Bitcoin's design principles. The ORC protocols build upon these experiences, specifically addressing the limitations of recent standards like Ordinals and Runes.

### **1.2 Motivation**

Current Bitcoin-based token standards face significant challenges:

● **UTXO Bloat**: Ordinals and Runes attach data to spendable outputs, increasing the UTXO set size

● **Limited Functionality**: Runes supports only fungible tokens, lacking NFT capabilities

● **Inefficient Encoding**: BRC-20 uses verbose JSON, wasting limited OP\_RETURN space

● **Layer 2 Challenges**: Existing standards lack native Lightning Network compatibility

● **Indexing Complexity**: Off-chain indexers face challenges with consistency and reliability

The ORC protocols address these challenges by:

1. Storing all token data in unspendable OP\_RETURN outputs

2. Supporting both fungible (ORC20) and non-fungible (ORC721) tokens

3. Using efficient encoding to maximize the 80-byte OP\_RETURN capacity

4. Designing for Lightning Network compatibility from inception

5. Providing a robust indexing solution via opreturncomment.com

### 

### **1.3 Design Philosophy**

The ORC protocols adhere to the following principles:

● **Bitcoin Alignment**: Respect Bitcoin's design philosophy and technical constraints

● **UTXO Efficiency**: Minimize impact on the UTXO set to preserve Bitcoin's scalability

● **Simplicity**: Favor simple, well-defined operations over complex functionality

● **Permissionless Innovation**: Enable anyone to create and transfer tokens without gatekeepers

● **Layer 2 Compatibility**: Design for compatibility with Lightning Network and other L2 solutions

● **Upgrade Path**: Prepare for potential future Bitcoin protocol upgrades

## **2. Protocol Design Principles**

### **2.1 Separation of Concerns**

The ORC protocols maintain a clear separation between:

● **Value Transfer**: Bitcoin's native UTXO system for monetary value

● **Metadata Storage**: OP\_RETURN outputs for token data

● **State Tracking**: Off-chain indexing system for current token state

This separation ensures that token operations don't compromise Bitcoin's fundamental monetary properties while enabling rich asset functionality.

### **2.2 Efficient Data Storage**

The 80-byte OP\_RETURN limit requires efficient data encoding:

● **Minimal JSON**: Shortened key names and optimized structures

● **Binary Encoding**: Using binary formats for identifiers and quantities where appropriate

● **Off-Chain References**: IPFS hashes for larger metadata

● **Multi-Part Transactions**: Linking multiple transactions for complex operations

### **2.3 Security Model**

The ORC protocols inherit Bitcoin's security model with additional considerations:

● **Transaction Validation**: Standard Bitcoin consensus rules for validity

● **Ownership Verification**: Public key cryptography for authentication

● **Issuer Controls**: Special permissions for token issuers

● **Indexer Consensus**: Rules for resolving inconsistencies between indexers

### **2.4 Interoperability**

The protocols are designed for interoperability:

● **ERC-20/ERC-721 Similarity**: Familiar interfaces for existing token developers

● **Cross-Protocol Compatibility**: Methods for representing tokens from other Bitcoin standards

● **Layer 2 Integration**: Design patterns for Lightning Network compatibility

● **Cross-Chain Bridges**: Potential for interoperability with other blockchains

## **3. ORC20: Fungible Token Standard**

### **3.1 Token Properties**

ORC20 tokens have the following properties:

|  |  |  |  |
| --- | --- | --- | --- |
| **Property** | **Description** | **Required** | **Example** |
| name | Human-readable token name | Yes | "Example Token" |
| symbol | Short token identifier | Yes | "EXT" |
| supply | Total token quantity | Yes | 1000000 |
| decimals | Divisibility factor | No | 8 (default) |
| issuer | Address controlling token | Yes | bc1q... |
| mintable | Whether additional issuance is allowed | No | false (default) |
| tokenId | Unique identifier | Yes (generated) | [SHA-256 hash] |

### 

### 

### 

### 

### 

### **3.2 Operations**

#### **3.2.1 Token Creation (Deploy)**

Creates a new token with defined properties:

{

"p":"ORC20",

"op":"deploy",

"name":"Example Token",

"sym":"EXT",

"sup":"1000000",

"dec":"8",

"mint":false

}

The transaction that contains this OP\_RETURN output establishes:

● The token's existence and properties

● The initial supply, allocated to the issuer

● The tokenId, derived from the transaction

#### **3.2.2 Token Transfer**

Moves tokens between addresses:

{

"p":"ORC20",

"op":"transfer",

"id":"[tokenId]",

"amt":"100",

"to":"bc1q..."

}

The sender is implicitly the address that creates the transaction. The indexer verifies that the sender has sufficient balance before updating the state.

#### 

#### 

#### 

#### 

#### **3.2.3 Token Burn**

Permanently removes tokens from circulation:

{

"p":"ORC20",

"op":"burn",

"id":"[tokenId]",

"amt":"50"

}

Burning reduces the sender's balance and the total circulating supply. It cannot be reversed.

#### **3.2.4 Token Mint (Optional)**

Creates additional tokens if allowed by the token properties:

{

"p":"ORC20",

"op":"mint",

"id":"[tokenId]",

"amt":"1000",

"to":"bc1q..."

}

Only the issuer can mint new tokens, and only if the token was created with "mint":true.

### **3.3 State Transitions**

Every valid ORC20 operation triggers a state transition:

● **Deploy**: Initialize token properties and allocate supply to issuer

● **Transfer**: Deduct balance from sender, add to recipient

● **Burn**: Deduct balance from sender, reduce circulating supply

● **Mint**: Increase recipient balance, increase circulating supply

The indexer maintains this state and enforces rules like:

● Balances cannot be negative

● Only the issuer can mint

● Total supply must match the sum of all balances

### 

### **3.4 Error Handling**

Invalid operations are processed by the indexer as follows:

● **Insufficient Balance**: Transfer/burn is rejected with no state change

● **Invalid TokenId**: Operation is ignored

● **Unauthorized Mint**: Mint is rejected with no state change

## **4. ORC721: Non-Fungible Token Standard**

### **4.1 NFT Properties**

ORC721 tokens have the following properties:

|  |  |  |  |
| --- | --- | --- | --- |
| **Property** | **Description** | **Required** | **Example** |
| name | NFT collection name | Yes | "Example Collection" |
| description | NFT description | No | "A unique digital item" |
| image | Visual representation | No | "ipfs://Qm..." |
| attributes | Trait properties | No | [{"trait":"color","value":"blue"}] |
| issuer | Address controlling NFT | Yes | bc1q... |
| nftId | Unique identifier | Yes (generated) | [transaction ID]:[index] |

### **4.2 Operations**

#### **4.2.1 NFT Creation (Deploy)**

Creates a new NFT with defined properties:

{

"p":"ORC721",

"op":"deploy",

"name":"Example NFT",

"desc":"A unique digital item",

"img":"ipfs://Qm...",

"attr":[{"trait":"color","value":"blue"}]

}

The transaction establishes:

● The NFT's existence and properties

● The initial ownership (the issuer)

● The nftId, derived from the transaction

#### **4.2.2 NFT Transfer**

Transfers ownership of an NFT:

{

"p":"ORC721",

"op":"transfer",

"id":"[nftId]",

"to":"bc1q..."

}

The sender must be the current owner of the NFT.

#### **4.2.3 NFT Metadata Update**

Updates an NFT's metadata:

{

"p":"ORC721",

"op":"update",

"id":"[nftId]",

"desc":"Updated description",

"img":"ipfs://Qm2..."

}

Only the issuer can update metadata, regardless of current ownership.

#### **4.2.4 NFT Burn**

Permanently destroys an NFT:

{

"p":"ORC721",

"op":"burn",

"id":"[nftId]"

}

The sender must be the current owner of the NFT.

### **4.3 Collections**

NFTs can be organized into collections:

{

"p":"ORC721",

"op":"deploy",

"name":"Example Collection",

"sym":"EXNFT",

"col":"col1"

}

Collection properties include:

● Collection identifier

● Shared metadata attributes

● Royalty information

● Creator details

### **4.4 Metadata Management**

NFT metadata can be stored:

● **On-Chain**: Core properties in the OP\_RETURN output

● **Off-Chain**: Larger media on IPFS or similar systems

● **Hybrid**: IPFS hashes on-chain with content off-chain

The metadata update operation enables dynamic NFTs that can evolve over time, while maintaining an immutable record of changes on-chain.

## **5. Transaction Structure and Encoding**

### **5.1 OP\_RETURN Structure**

All ORC protocol data is stored in OP\_RETURN outputs with this structure:

OP\_RETURN <data>

Where <data> is the encoded protocol payload.

### 

### **5.2 Payload Encoding**

The payload uses minimal JSON encoding to maximize efficiency:

● Short property names (e.g., "p" instead of "protocol")

● Compact value representation

● No unnecessary whitespace

● Binary encoding for complex data when appropriate

### **5.3 Transaction Templates**

#### **5.3.1 Basic ORC Transaction**

Input: [funding inputs]

Output 1: OP\_RETURN <ORC data>

Output 2: [change address] [change amount]

#### **5.3.2 Multi-Part Transaction (for complex operations)**

Input: [funding inputs]

Output 1: OP\_RETURN <ORC data part 1>

Output 2: [continuation address] [dust amount]

Input: [continuation input]

Output 1: OP\_RETURN <ORC data part 2>

Output 2: [change address] [change amount]

Multi-part transactions use continuation addresses to link related operations.

### **5.4 Size Limitations**

The 80-byte OP\_RETURN limit requires efficient encoding strategies:

|  |  |  |
| --- | --- | --- |
| **Operation** | **Typical Size** | **Strategy for Larger Data** |
| ORC20 Deploy | 60-80 bytes | Minimal property names |
| ORC20 Transfer | 40-60 bytes | Compact ID encoding |
| ORC721 Deploy | 70-200 bytes | IPFS for metadata, multi-part transactions |
| ORC721 Transfer | 40-60 bytes | Compact ID encoding |
| ORC721 Update | 70-200 bytes | IPFS for metadata, multi-part transactions |

### **5.5 Future Expansion**

If Bitcoin increases or removes the OP\_RETURN size limit, the protocol can expand to include:

● More detailed metadata directly on-chain

● More complex operations in single transactions

● Enhanced functionality without splitting across multiple transactions

## **6. Indexing System Architecture**

### **6.1 System Components**

The opreturncomment.com indexing system consists of:

● **Bitcoin Node Connector**: Interfaces with Bitcoin nodes

● **Block Parser**: Extracts ORC data from blocks

● **Transaction Processor**: Validates operations and updates state

● **State Database**: Stores current token/NFT state

● **API Server**: Exposes data via REST API

● **WebSocket Service**: Provides real-time updates

### **6.2 Indexing Process**

The indexing process follows these steps:

1. **Block Ingestion**: New blocks are received from the Bitcoin network

2. **Transaction Filtering**: Transactions with OP\_RETURN outputs are identified

3. **Data Extraction**: ORC protocol data is parsed from OP\_RETURN outputs

4. **Validation**: Operations are validated against current state

5. **State Update**: Valid operations trigger state changes

6. **Event Generation**: State changes trigger events for subscribers

### 

### 

### 

### 

### **6.3 State Management**

The indexer maintains the following state:

● **Token Registry**: Details of all ORC20 tokens

● **NFT Registry**: Details of all ORC721 NFTs

● **Balance Ledger**: Token balances for all addresses

● **Ownership Registry**: Current owners of all NFTs

● **Transaction History**: Record of all ORC operations

### **6.4 Consensus Rules**

If multiple indexers exist, they follow these consensus rules:

● **Bitcoin Primacy**: Bitcoin's blockchain is the source of truth

● **Deterministic Processing**: Operations must be processed in block order

● **Consistent Rules**: All indexers apply the same validation rules

● **State Reconciliation**: Regular checkpoints for consistency verification

### **6.5 API Services**

The opreturncomment.com API provides these services:

● **Token Queries**: Information about ORC20 tokens

● **NFT Queries**: Information about ORC721 NFTs

● **Address Queries**: Token balances and NFT ownership

● **Transaction Submission**: Broadcast signed transactions

● **Event Subscription**: WebSocket notifications for state changes

## **7. Lightning Network Integration**

### **7.1 Channel Design for Tokens**

ORC tokens can be represented in Lightning channels:

● **Balance Encoding**: Channel state includes token balances

● **HTLC Extension**: Hash Time-Locked Contracts for token transfers

● **Multi-Asset Channels**: Single channel for multiple token types

● **Commitment Transactions**: Modified to represent token ownership

### 

### 

### **7.2 Token Transfer Protocol**

The token transfer protocol works as follows:

1. **Invoice Creation**: Recipient generates invoice for token amount

2. **Path Finding**: Sender finds route with sufficient token liquidity

3. **HTLC Creation**: Sender initiates token HTLC along the path

4. **Preimage Revelation**: Recipient reveals preimage to claim tokens

5. **Settlement**: HTLCs settle, updating channel balances

### **7.3 NFT Transfer Protocol**

NFTs can be transferred via Lightning:

● **NFT-Specific HTLC**: Special contract for NFT transfers

● **Atomic Swaps**: Exchange NFTs for BTC or tokens

● **Conditional Transfers**: Time-locked or condition-based transfers

● **Proof of Ownership**: Verification within Lightning channels

### **7.4 On-Chain Settlement**

Channel states eventually settle on-chain:

● **Balance Resolution**: Final token balances in closing transactions

● **NFT Ownership**: Final NFT ownership in closing transactions

● **Dispute Resolution**: Protocol for resolving disagreements

### **7.5 Technical Challenges**

Lightning integration faces these challenges:

● **Channel Capacity**: Managing multiple assets within BTC-defined capacity

● **Routing Complexity**: Finding routes with multi-asset liquidity

● **Update Frequency**: Handling frequent state updates

● **Protocol Extensions**: Required modifications to Lightning implementations

The ORC protocols address these challenges through:

● Compact state representation

● Efficient routing algorithms

● Batched updates

● Phased implementation approach

## 

## **8. Compatibility with Existing Standards**

### **8.1 Ordinals Compatibility**

The ORC721 protocol can represent Ordinals inscriptions:

● **Wrapper Mechanism**: Ordinals can be "wrapped" as ORC721 NFTs

● **Two-Way Bridge**: Lock inscription, mint ORC721, and reverse

● **Metadata Mapping**: Convert Ordinals metadata to ORC721 format

● **Collection Grouping**: Group related inscriptions into collections

Implementation details:

{

"p":"ORC721",

"op":"wrap",

"ord":"[inscription ID]",

"proof":"[ownership proof]"

}

### **8.2 Runes Compatibility**

The ORC20 protocol can represent Runes tokens:

● **Wrapper Mechanism**: Runes can be wrapped as ORC20 tokens

● **Two-Way Bridge**: Lock Runes, mint ORC20, and reverse

● **Property Mapping**: Convert Runes parameters to ORC20 format

● **Batch Conversion**: Convert multiple Runes in single operation

Implementation details:

{

"p":"ORC20",

"op":"wrap",

"rune":"[rune ID]",

"proof":"[ownership proof]"

}

### 

### 

### **8.3 BRC-20 Compatibility**

The ORC20 protocol can represent BRC-20 tokens:

● **Wrapper Mechanism**: BRC-20 can be wrapped as ORC20 tokens

● **Two-Way Bridge**: Lock BRC-20, mint ORC20, and reverse

● **Property Mapping**: Convert BRC-20 parameters to ORC20 format

● **Verification System**: Validate BRC-20 ownership before wrapping

Implementation details:

{

"p":"ORC20",

"op":"wrap",

"brc20":"[BRC-20 ticker]",

"proof":"[ownership proof]"

}

### **8.4 Name Collision Management**

To handle tokens with identical names:

**Namespace System**: Optional prefixes to distinguish tokens  
 [namespace].[name]

●

● **Unique Identifiers**: Each token has a unique ID independent of name

● **First-Come-First-Served**: Basic principle for name allocation

● **Verification System**: Optional badges for recognized projects

Example search capabilities:

● Exact matching: ORC20.search("Bitcoin")

● Namespace filtering: ORC20.search("official.\*")

● Creator filtering: ORC20.searchByCreator("[address]")

## 

## 

## 

## **9. Scaling Considerations**

### **9.1 Current Limitations**

The ORC protocols face these scaling challenges:

● **OP\_RETURN Size**: 80-byte limit constrains operation complexity

● **Transaction Throughput**: Bitcoin's block size limits overall capacity

● **Indexer Performance**: Processing high volumes of transactions

● **API Capacity**: Handling numerous queries and updates

### **9.2 Short-Term Solutions**

Until fundamental constraints change:

● **Efficient Encoding**: Maximize use of available space

● **IPFS Integration**: Store large data off-chain

● **Multi-Part Transactions**: Split complex operations

● **Caching Layer**: Reduce database load for common queries

● **Horizontal Scaling**: Distribute indexer across multiple servers

### **9.3 Medium-Term Solutions**

With optimization and ecosystem growth:

● **Layer 2 Integration**: Move frequent operations to Lightning Network

● **Batched Operations**: Combine multiple token transfers in single transactions

● **State Channels**: Conduct multiple operations off-chain with single settlement

● **Database Sharding**: Partition data for better performance

● **CDN Integration**: Distribute API load globally

### **9.4 Long-Term Vision**

If Bitcoin protocol evolves:

● **Expanded OP\_RETURN**: Utilize larger data capacity

● **Covenant Support**: Enhanced script capabilities for token operations

● **UTXO Commitments**: Improve verification efficiency

● **Cross-Chain Interoperability**: Bridge to other blockchain ecosystems

● **Advanced Smart Contracts**: More complex token behaviors

## 

## **10. Security Analysis**

### **10.1 Threat Model**

The ORC protocols consider these threat vectors:

● **Double-Spending**: Attempting to spend tokens multiple times

● **Replay Attacks**: Repeating valid transactions

● **Metadata Manipulation**: Altering token/NFT properties

● **Indexer Attacks**: Disrupting the indexing system

● **Front-Running**: Exploiting pending transactions

● **Protocol Confusion**: Mixing different protocols

### **10.2 Security Measures**

To address these threats:

● **Bitcoin Consensus**: Leverage Bitcoin's security for transaction validity

● **Signature Verification**: Ensure operations are properly authorized

● **IPFS Content Addressing**: Immutable references to off-chain data

● **Multiple Indexers**: Distribute indexing to prevent centralization

● **Transaction Design**: Mitigate front-running through proper structure

● **Protocol Prefixes**: Clear identification of protocol messages

### **10.3 Vulnerabilities and Mitigations**

Potential vulnerabilities include:

● **Indexer Centralization**: Risk of opreturncomment.com control

○ *Mitigation*: Open-source indexer, multiple independent operators

● **State Inconsistency**: Indexers may diverge in their view of state

○ *Mitigation*: Consistent rules, regular checkpoints, reconciliation process

● **Data Availability**: IPFS content may become unavailable

○ *Mitigation*: Multiple pinning services, redundant storage

● **Front-Running**: Miners may reorder transactions

○ *Mitigation*: Atomic operations, time-insensitive designs

### **10.4 Security Best Practices**

For developers and users:

● **Verify Transactions**: Confirm operations are properly recorded

● **Use Secure Wallets**: Protect private keys controlling tokens

● **Check Metadata Integrity**: Verify IPFS content matches hashes

● **Monitor Balances**: Regularly check token balances for discrepancies

● **Follow Updates**: Stay informed about protocol developments

## **11. Implementation Roadmap**

### **11.1 Development Phases**

**Phase 1 (Weeks 1-2): Foundation**

● Complete protocol specification

● Set up development infrastructure

● Create OP\_RETURN encoding/decoding module

● Design database schema

**Phase 2 (Weeks 3-4): Core Protocol**

● Implement ORC20 token operations

● Implement ORC721 NFT operations

● Develop basic indexing functionality

● Create transaction validation system

**Phase 3 (Weeks 5-6): Advanced Features**

● Implement Lightning Network integration

● Add IPFS connector for metadata

● Develop wrapping mechanisms for compatibility

● Build API services

**Phase 4 (Weeks 7-8): Deployment and Launch**

● Conduct comprehensive testing

● Deploy indexer to production

● Publish documentation

● Launch public services

### **11.2 Testnet Deployment**

Before mainnet launch:

● Deploy full protocol stack to Bitcoin testnet

● Conduct extensive testing with realistic workloads

● Document test results and performance metrics

● Address any issues discovered during testing

### 

### **11.3 Mainnet Launch**

The mainnet deployment includes:

● Migrating tested code to production environment

● Initializing production databases

● Configuring monitoring and alerting

● Establishing backup and recovery procedures

● Launching public API services on opreturncomment.com

### **11.4 Post-Launch Support**

After launch:

● Monitor system performance and security

● Provide developer support via documentation and forums

● Address bugs and performance issues

● Gather feedback for future improvements

## **12. Conclusion and Future Work**

### **12.1 Summary**

The ORC protocols represent a significant advancement in Bitcoin asset functionality, providing:

● Efficient token and NFT capabilities without UTXO bloat

● Lightning Network compatibility for high-throughput operation

● Backward compatibility with existing standards

● Scalable infrastructure via opreturncomment.com

These protocols open Bitcoin's memo space for innovation while respecting Bitcoin's design philosophy and security model.

### **12.2 Future Research Areas**

Topics for continued exploration:

● Enhanced privacy mechanisms for token transfers

● Advanced programmability within Bitcoin's constraints

● Cross-chain bridges to other blockchain ecosystems

● Formal verification of protocol security properties

### 

### **12.3 Ecosystem Development**

Beyond the protocols themselves:

● Developer tools and SDKs

● Mobile and web wallet integration

● Exchange and marketplace support

● DApp development framework

### **12.4 Governance and Evolution**

Long-term protocol governance will involve:

● ORC Improvement Proposal (OIP) process

● Community feedback mechanisms

● Backward compatibility considerations

● Version management strategy

## 

## 

## 

## 

## 

## 

## 

## 

## 

## 

## **13. References**

1. Nakamoto, S. (2008). "Bitcoin: A Peer-to-Peer Electronic Cash System"

2. BIP 0080 - OP\_RETURN Data Standardization

3. Casey Rodarmor (2023). "Ordinals: Satoshi in the Machine"

4. Casey Rodarmor (2023). "Runes: A Bitcoin Native Framework for Fungible Tokens"

5. Lightning Network Specification (BOLT)

6. IPFS: Inter-Planetary File System Protocol

7. ERC-20 Token Standard

8. ERC-721 Non-Fungible Token Standard

## **14. Appendices**

### **Appendix A: JSON Schema Definitions**

{

"ORC20Deploy": {

"type": "object",

"required": ["p", "op", "name", "sym", "sup"],

"properties": {

"p": { "enum": ["ORC20"] },

"op": { "enum": ["deploy"] },

"name": { "type": "string", "maxLength": 32 },

"sym": { "type": "string", "maxLength": 8 },

"sup": { "type": "string", "pattern": "^[0-9]+$" },

"dec": { "type": "string", "pattern": "^[0-9]+$" },

"mint": { "type": "boolean" }

}

}

}

### 

### 

### 

### 

### 

### 

### **Appendix B: Code Examples**

**JavaScript: Creating ORC20 Token**

const bitcoin = require('bitcoinjs-lib');

function createORC20Token(tokenParams, keyPair, network) {

const { name, symbol, supply, decimals = 8, mintable = false } = tokenParams;

// Create OP\_RETURN data

const payload = JSON.stringify({

p: "ORC20",

op: "deploy",

name: name,

sym: symbol,

sup: supply.toString(),

dec: decimals.toString(),

mint: mintable

});

// Create transaction

const tx = new bitcoin.TransactionBuilder(network);

// Add input (assume we have UTXO data)

tx.addInput('prevTxId', 0);

// Add OP\_RETURN output

const dataScript = bitcoin.script.compile([

bitcoin.opcodes.OP\_RETURN,

Buffer.from(payload)

]);

tx.addOutput(dataScript, 0);

// Add change output (example)

tx.addOutput(keyPair.getAddress(), 99000);

// Sign transaction

tx.sign(0, keyPair);

return tx.build().toHex();

}

### 

### **Appendix C: API Documentation Preview**

**Token Endpoints**

GET /api/tokens

{

"tokens": [

{

"tokenId": "abc123",

"name": "Example Token",

"symbol": "EXT",

"supply": "1000000",

"decimals": "8",

"issuer": "bc1q...",

"created": "2025-01-01T00:00:00Z"

}

],

"pagination": {

"page": 1,

"perPage": 10,

"total": 120

}

}

### **Appendix D: Test Vectors**

**ORC20 Deploy**

Input:

{

"name": "Example Token",

"symbol": "EXT",

"supply": "1000000",

"decimals": "8",

"mintable": false

}

Expected OP\_RETURN (hex):

6a4c507b2270223a224f524332302222c226f70223a226465706c6f79222c226e616d65223a224578616d706c6520546f6b656e222c2273796d223a22455854222c22737570223a2231303030303030222c22646563223a2238222c226d696e74223a66616c73657d

Social link:

Git: <https://github.com/opreturncomment>