Version 2.8

# PROJECT

# Intelligent Daemon System

# Detailed Design

# &

# Architecture

# Table of Contents

# REVISION HISTORY

| Version | Description | Date | Author |
|---|---|---|---|
| 1.11 | Appendix L and 5.7.3 point were added. 3.1.1, 3.1.2, 3.8, 5.1, 5.7 points were updated | 21/10/2015-30/10/2015 | Olga Kuznetsova |
| See revision history of previous versions in the DD v1.11 | | | |
| 2.0 | Major updates related to “Warm Storage” solution. Appendix F, Point 2 and DB structure (Point 3) were updated. 1.2, 5.1 sub-points were updated. Table of Contents was refreshed. | 04/11/2015 | Olga Kuznetsova |
| 2.1 | Appendix L and 1.1, 2, 3 points were updated | 05/11/2015-12/11/2015 | Olga Kuznetsova |
| 2.2 | Appendix M was added. Appendix A and 1.1, 1.2, 2.1, 3, 5.1 points were updated | 16/11/2015-19/11/2015 | Olga Kuznetsova |
| 2.3 | New point 2.11 “Configuration and Logs” was added. 2.10.2, 3.1, 4.1, 5.1, 5.7.3, 6.2.4 points were updated. Acronyms … of Current Doc was updated. | 20/11/2015-01/12/2015 | Olga Kuznetsova |
| 2.4 | Points 1.2, 2.12.7, 2.8, 3.9.1, 3.9.2, 4.7, 5.7.2, 9.1 were updated | 2/12/2015 | Trupti Birje |
| 2.5 | 2.3 and 2.4 versions were merged | 2/12/2015 | Olga Kuznetsova, Trupti Birje |
| 2.6 | Points 3.9.1, 4.1.1, 4.1.2, 4.1.3, 4.1.4, 4.1.5 and Appendix L were updated | 3/12/2015-18/12/2015 | Trupti Birje |
| 2.7 | “Definitions, Acronyms and Abbreviations”, Glossary, Appendixes L and E, 2.2.3, 2.10.2, 2.10.4, 2.11.2, 2.12, 2.12.7, 2.8, 3.1.1, 3.1.2, 3.9.1, 4, 4.1.1, 4.1.3, 5.1.1, 5.1.3 points were updated | 3/12/2015-18/12/2015 | Olga Kuznetsova |

| | | | |
|---|---|---|---|
| 2.8 | 2.6 and 2.7 versions were merged | 18/12/2015 | Olga Kuznetsova, Trupti Birje |

## Definitions, Acronyms and Abbreviations

| | |
|---|---|
| ASN.1 | Abstract Syntax Notation One [1.14] |
| API | Application Programming Interface |
| BER | Basic Encoding Rules |
| BIP | Bitcoin Improvement Proposal |
| BSD license | Berkeley Source Distribution license [1.11] |
| BTC, Btc | Bitcoin |
| CER | Canonical Encoding Rules |
| CDDL | Common Development and Distribution License |
| DB | Data Base |
| DER | Distinguished Encoding Rules [1.15] |
| ECDSA | Elliptic Curve Digital Signature Algorithm |
| FOS | Free Open Source |
| GNU LGPL v2 | GNU Library General Public License version 2 [1.12] |
| GPL | General Public License |
| GUI | Graphic User Interface |
| JSON | JavaScript Object Notation = an open language-independent data format that uses human-readable text to transmit data objects consisting of attribute–value pairs. |
| MQ | Message Queue |
| Multi-sig | Multi signature |
| NAF | Non-Adjacent Form |
| P2P | Peer-to-Peer |
| P2PKH | Pay-to-Public-Key-Hash |

| | |
|---|---|
| P2SH | Pay-to-Script-Hash |
| PK | Primary Key |
| PSQL SP | PostgreSQL stored procedures |
| RDB | Relational Data Base |
| REST | Representational State Transfer = architectural style |
| RPC | Remote Procedure Call |
| SHA | Secure Hash Algorithm |
| Single-sig | Single signature |
| SW | Software |
| Trx | Transaction |
| txid, TXID | Transaction Identifier = hash in hex of signed transaction |
| UTXO | Unspent Transaction Output |
| Web App. | Web Application |
| Opcode | Operation code |
| 4S | Shamir's Secret Sharing Scheme |
| WIF | Wallet Import Format |

## Acronyms and Abbreviations of the Current Document

| | |
|---|---|
| ATrxMSS | Accounting Transaction Management SubSystem |
| BTrxMSS | Bank Transaction Management SubSystem |
| ContrMSS | Contracts Management SubSystem |
| CW API | Common Ware API |
| DmnCS | Daemon Core System |
| ETrxMSS | Exchange Transaction Management SubSystem |
| ECDSA API | Elliptic Curve Digital Signature Algorithm API |
| FOS DmnCC | FOS Daemon Core Component |
| IntDS | Intelligent Daemon System |
| KeysMC | Keys Management Component |
| MCG API | Mnemonic Code Generator API |
| MTrxMSS | Message Transaction Management SubSystem |
| MntS | Monitoring System |
| PubKey | Public Key |
| PriKey | Private Key |
| STrxMSS | Single-sig Transaction Management SubSystem |
| TrxMC | Transaction Management Component |
| WDmnCC | Wrapper of Daemon Core Component |
| EncrPK | Encrypted Private Key |
| 4S API | Shamir’s Secret Sharing Scheme API |

# 1. References

## 1.1 Online References

| Reference | Website URL | Description |
|---|---|---|
| [1.1] | http://en.wikipedia.org/wiki/Representational_state_transfer | REST |
| [1.2] | https://bitcoin.org/en/download | Bitcoin Core |
| [1.3] | http://nginx.org/en/ | Nginx official website |
| [1.4] | https://www.rabbitmq.com/ | RabbitMQ official website |
| [1.5] | http://tomcat.apache.org/ | Tomcat official website |
| [1.6] | http://www.postgresql.org/ | PostgreSQL official website |
| [1.7] | http://www.oracle.com/technetwork/java/api-141528.html | Java API specifications |
| [1.8] | http://slony.info/ | Slony-I official website |
| [1.9] | http://www.pgpool.net/mediawiki/index.php/Main_Page | Pgpool-II official website |
| [1.10] | http://www.keepalived.org/download.html | Keepalived official website |
| [1.11] | http://www.linfo.org/bsdlicense.html | BSD license |
| [1.12] | http://www.gnu.org/licenses/old-licenses/lgpl-2.0.en.html | GNU LGPL v2 |
| [1.13] | http://dotnetcodr.com/2014/06/05/rabbitmq-in-net-data-serialisation/ | RabbitMQ data serialization example |
| [1.14] | https://en.wikipedia.org/wiki/Abstract_Syntax_Notation_One | ASN.1 description |
| [1.15] | https://en.wikipedia.org/wiki/X.690#DER_encoding | X.690 specifying several ASN.1 encoding formats: BER, CER, DER |
| [1.16] | http://blog.mybatis.org/p/about.html<br>http://mybatis.org/mybatis-3/dependency-info.html | myBatis |
| [1.17] | http://code.google.com/p/spock/ | Spock framework |
| [1.18] | http://junit.org/ | JUnit framework |

| | | |
|---|---|---|
| [1.19] | http://www.jmock.org/ | JMock framework |
| [1.20] | http://www.postgresql.org/docs/8.0/static/plpgsql.html | PL/pgSQL |
| [1.21] | http://www.postgresqltutorial.com/introduction-to-postgresql-stored-procedures/ | Introduction to PostgreSQL Stored Procedures |
| [1.22] | https://jersey.java.net/ | Jersey Framework |
| [1.23] | https://jax-rs-spec.java.net/ | JAX-RS API |
| [1.24] | http://ant.apache.org/ | Apache Ant home page |
| [1.25] | http://ant.apache.org/ivy/ | Apache Ivy home page |
| [1.26] | http://json.org/ | JSON representation |

## 1.2 Bitcoin System Online References

| Reference | Website URL | Description |
|---|---|---|
| [2.1] | https://en.bitcoin.it/wiki/Transaction | Bitcoin Transactions |
| [2.2] | https://en.bitcoin.it/wiki/Script | Scripting system for Btc transactions |
| [2.3] | https://en.bitcoin.it/wiki/Block_chain | Block Chain description |
| [2.4] | https://en.bitcoin.it/wiki/Genesis_block | Genesis Block definition |
| [2.5] | https://en.bitcoin.it/wiki/Blocks | Blocks description |
| [2.6] | https://github.com/OpenAssets/open-assets-protocol/blob/master/specification.mediawiki | Open Assets Transactions |
| [2.7] | https://en.bitcoin.it/wiki/Contracts | Contracts Transactions |
| [2.8] | https://en.bitcoin.it/wiki/Bitcoin_Improvement_Proposals | BIP definition |
| [2.9] | https://github.com/bitcoin/bips/ | BIPs list in the github |
| [2.10] | https://en.bitcoin.it/wiki/IP_Transactions | IP Transaction |
| [2.11] | https://en.bitcoin.it/wiki/Transaction_fees | Transaction Fees info |
| [2.12] | https://en.bitcoin.it/wiki/Original_Bitcoin_client/API_calls_list | Btc client API |
| [2.13] | https://en.bitcoin.it/wiki/Protocol_documentation | Btc protocol overview |
| [2.14] | https://en.bitcoin.it/wiki/Technical_background_of_Bitcoin_addresses | Btc Addresses v1 overview |
| [2.15] | https://en.bitcoin.it/wiki/Address | Btc Address definition |
| [2.16] | https://bitcoin.org/en/developer-reference#get-tx | Btc Core RPCs |
| [2.17] | https://github.com/bitcoin/bips/blob/master/bip-0039.mediawiki | BIP-0039 |
| [2.18] | https://github.com/bitcoin/bips/blob/master/bip-0039/english.txt | Wordlist for BIP-0039 |
| [2.19] | https://github.com/bitcoin/bips/blob/master/bip-0065.mediawiki | BIP-0065 |
| [2.20] | https://bitcoinxt.software/ | Bitcoin XT |

| | | |
|---|---|---|
| [2.21] | http://chimera.labs.oreilly.com/books/1234000001802/ch04.html#base58 | Mastering Bitcoins chapter 4 |
| [2.22] | https://en.bitcoin.it/wiki/OP_CHECKSIG | Sighash types |
| [2.23] | https://en.bitcoin.it/wiki/File:Bitcoin_OpCheckSig_InDetail.png | Trx verification Steps: OP_CHECKSIG (SIGHASH_ALL only) |
| [2.24] | https://github.com/bitcoin/bips/blob/master/bip-0065.mediawiki#freezing-funds | BIP-0065 description |
| [2.25] | https://github.com/bitcoin/bips/blob/master/bip-0061.mediawiki#freezing-funds | BIP-0061 description |
| [2.26] | http://codesuppository.blogspot.com.au/2014/01/how-to-parse-bitcoin-blockchain.html | How to parse bitcoin blockchain |

## 1.3 Algorithms and Math online references

| Reference | Website URL | Description |
|---|---|---|
| [3.1] | http://en.wikipedia.org/wiki/Elliptic_Curve_Digital_Signature_Algorithm | ECDSA |
| [3.2] | http://en.wikipedia.org/wiki/Public-key_cryptography | Public-key cryptography |
| [3.3] | https://en.bitcoin.it/wiki/Base58Check_encoding | Base58Check_encoding |
| [3.4] | https://en.wikipedia.org/wiki/Shamir%27s_Secret_Sharing | Shamir’s Secret Sharing Scheme |
| [3.5] | https://tools.ietf.org/html/rfc6979 | Deterministic ECDSA |
| [3.6] | https://en.wikipedia.org/wiki/Unix_time | Unix Epoch timestamp |
| [3.7] | https://en.wikipedia.org/wiki/Lagrange_polynomial | Lagrange polynomial |
| [3.8] | https://en.wikipedia.org/wiki/Congruence_relation | Congruence relation |
| [3.9] | https://en.wikipedia.org/wiki/Modular_arithmetic | Modular arithmetic |
| [3.10] | https://en.wikipedia.org/wiki/Affine_coordinate_system | Affine coordinate system |
| [3.11] | https://en.wikipedia.org/wiki/Base64 | Base 64 Encoding |
| [3.12] | https://en.wikipedia.org/wiki/Base58 | Base 58 Encoding |

## 1.4 Offline References

| Reference | Document Name |
|---|---|
| [4.1] | Brier, ´E., D´ech`ene, I., Joye, M.: Unified point addition formulæ for elliptic curve cryptosystems. In Nedjah, N., de Macedo Mourelle, L., eds.: Embedded Cryptographic Hardware: Methodologies and Architectures. Nova Science Publishers (2004) 247–256 |
| [4.2] | Douglas Stebila, Nicolas Th´eriault: Unified Point Addition Formulæ and Side-Channel Attacks. Institute for Quantum Computing, University of Waterloo, Waterloo, ON, Canada, Department of Combinatorics and Optimization, University of Waterloo, Waterloo, ON, Canada |
| [4.3] | Di Wang, Supervisor: Dr. Nicolas T. Courtois; Secure Implementation of ECDSA Signatures in Bitcoin. MSc in Information Security. University College London, September 17, 2014 |
| [4.4] | E. Brier and M. Joye. Weierstraß elliptic curves and side-channel attacks. Public Key Cryptography, pages 335–345, 2002. |
| [4.5] | Billy Bob Brumley. Efficient Elliptic Curve Algorithms for Compact Digital Signatures. HELSINKI UNIVERSITY OF TECHNOLOGY, Department of Computer Science and Engineering & Laboratory for Theoretical Computer Science, Espoo, November 27, 2006 |
| [4.6] | Ernst G. Straus. Addition chains of vectors (problem 5125). American Mathematical Monthly, 71:806–808, 1964. |

## 1.5 Reference Documents

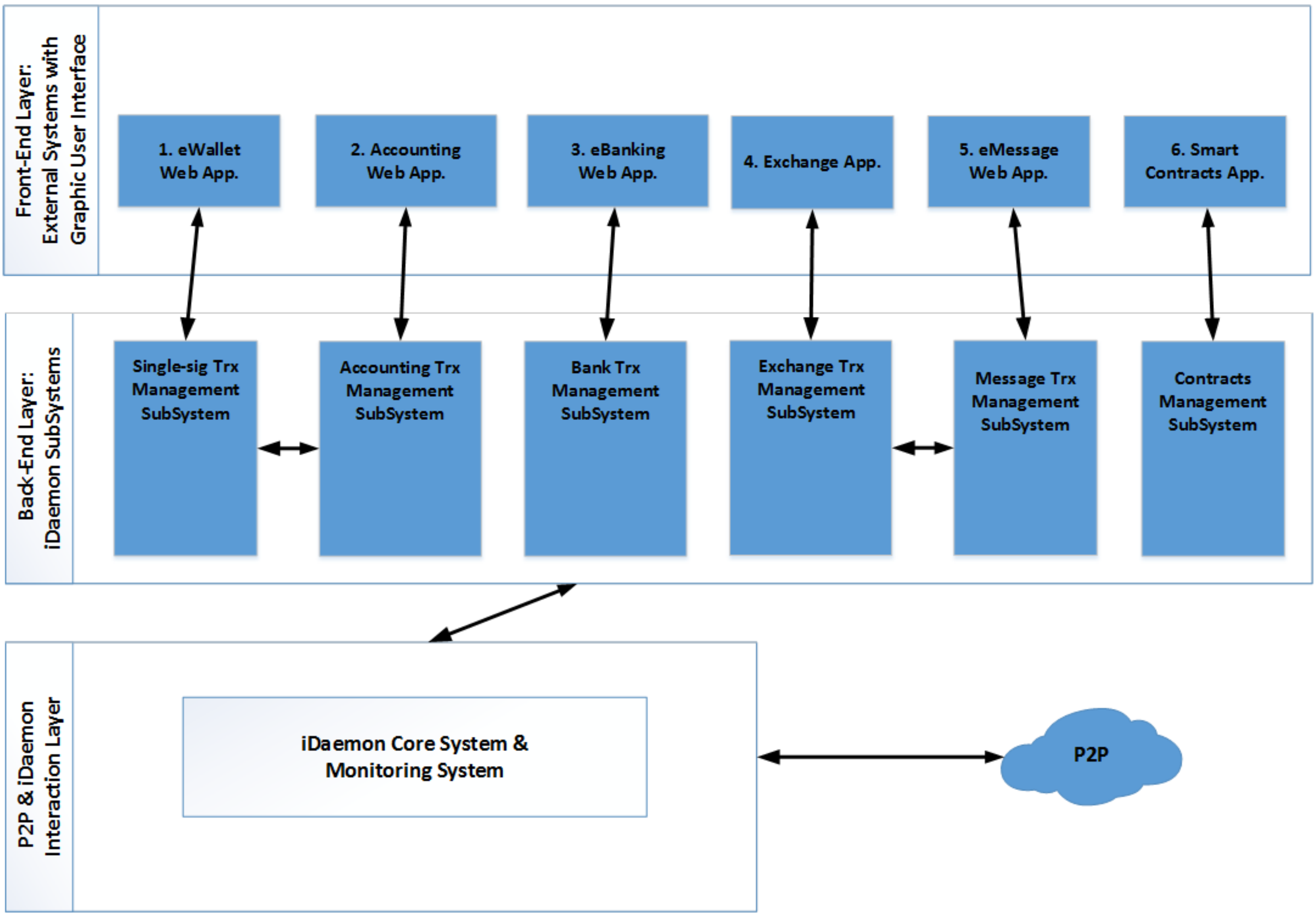
| Reference | Document Name | Description |
|---|---|---|
| [5.1] | “Intelligent Daemon System” Project Charter | Project Charter |
| [5.2] | “Intelligent Daemon System” Specification of Functionality | Functional Requirements |
| [5.3] | “eWallet Web Application” Detailed Design | Detailed Design |

# 2. Intelligent Daemon System Architecture

## 2.1 High Level Architecture

**Intelligent Daemon System (IntDS)** architecture is based on SOA and Microservice architecture approach to support different types of consumers (Web applications, Mobile applications, 3rd party's applications, etc.) The System should handle HTTP/HTTPS requests by executing business logic; accessing a database; exchanging messages; and returns a JSON [1.26]/XML response to a consumer system. The System SW configuration should use IPv6 protocol as much as possible. The System will use IPv4 protocol in case configuration problems with IPv6.

The diagram below (Pic. 2.1.1) shows the skeleton of IntDS layers.



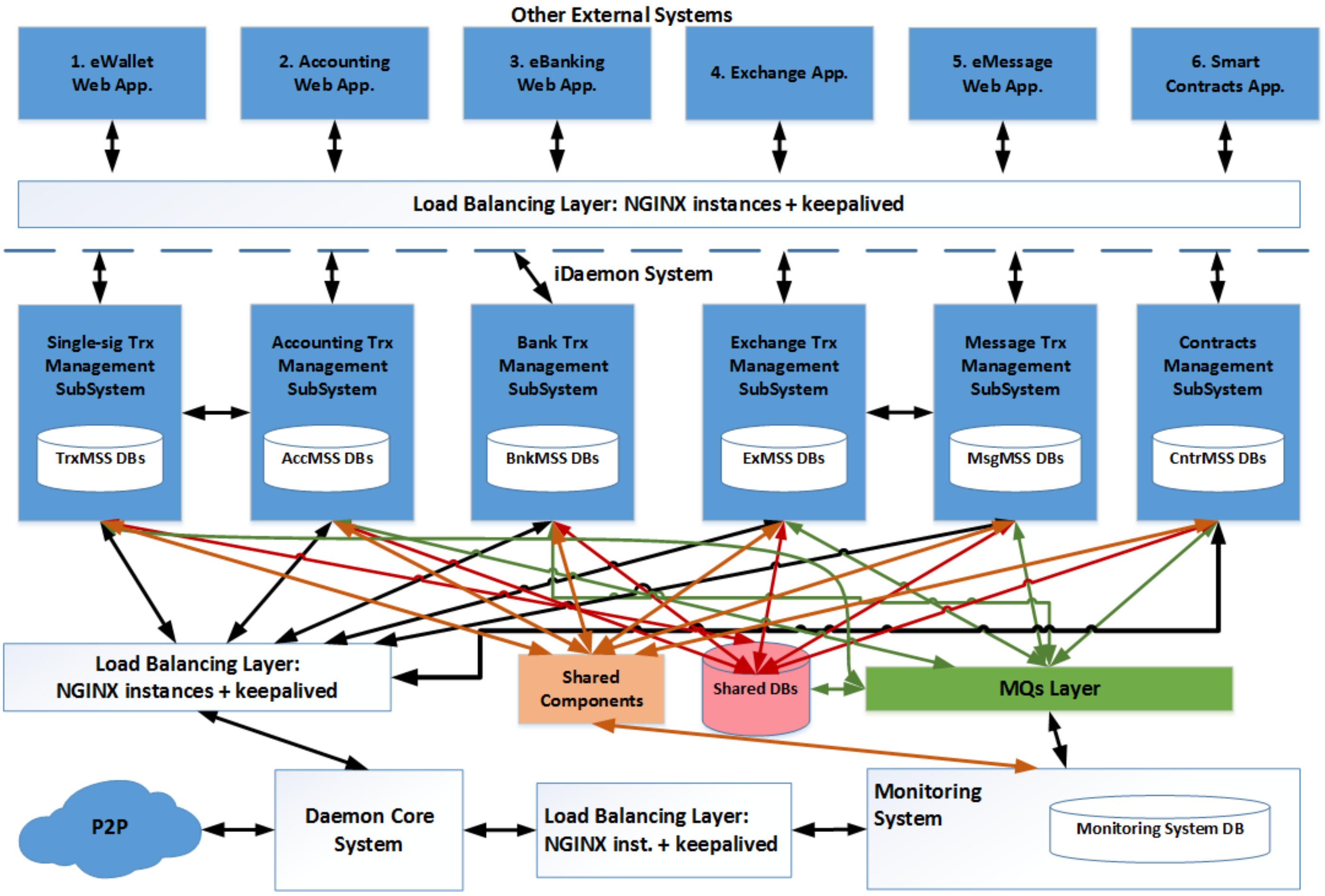

Pic. 2.1.1

### 2.1.1 Architecture Diagram

IntDS high level architecture includes (Pic. 2.1.2):

1. **Transaction Management SubSystems layer**: each sub-system implements main business logic related to particular External system or application. Currently, there are 6 sub-systems which implements different types of Btc transactions (see Appendix B):
   - **Single-sig Transaction Management SubSystem (STrxMSS)** supports “eWallet” web application and has possibility to support similar applications by using simple “Single-sig Trxs” type.
   - **Accounting Transaction Management SubSystem (ATrxMSS)** supports “Accounting” applications by using Financial Single-sig/Multi-sig Trxs types. ATrxMSS has possibility to interact with STrxMSS or another sub-system if necessary.
   - **Bank Transaction Management SubSystem (BTrxMSS)** supports “eBanking” web application and has possibility to support similar applications by using Financial Single-sig/Multi-sig Trxs types.
   - **Exchange Transaction Management SubSystem (ETrxMSS)** supports “Exchange” application. ETrxMSS uses different types of Btc transactions as:
     - Financial Single-sig Trxs
     - Financial Multi-sig Trxs
     - Open Assets Trxs
     - IP Trxs
     - Contracts
   - **Message Transaction Management SubSystem (MTrxMSS)** supports “eMessage” web application and has possibility to support similar applications by using “Message Trxs” type. MTrxMSS has possibility to interact with ETrxMSS or another sub-system if necessary.
   - **Contracts Management SubSystem (ContrMSS)** supports any “Smart Contacts” applications by using transactions of “Contract” type.

   Additional sub system can be added in current design according to new business logic requests if necessary.
2. **Monitoring System (MntS)**: provides Block Chain monitoring capabilities for each sub system. MntS consists of DB and Multithread application.
3. **Daemon Core System (DmnCS)**: coordinates how the network processes transactions. DmnCS consists of Load Balancing Layer, FOS Daemon Core Component (DmnCC) and Wrapper of DmnCC which is REST[1.1] services.
4. **MQs Layer**: provides asynchronous interactions between each sub system and MntS via MQ broker.
5. **Shared DBs**: stores Data which can be used by each sub system depending on business logic.
6. **Shared Components**: provides shared functionality and libraries for each sub system.




Pic. 2.1.2

**IntDS has Shared Libraries** (as java jar files) which can be plugged into application or used as standalone if it is needed:

1. **Common Ware API (CW API)**: implements services logic as connection types (DB, Daemon, MQ, etc.), log record types, performance records, common utilities, etc. CW API is used in all sub systems.
2. **4S API**: implements a splitting and reconstructing of secret string according to Shamir's Secret Sharing Scheme [3.4]
3. **ECDSA API**: implements Elliptic Curve Digital Signature Algorithm to generate a public/private keys pair.
4. **Mnemonic Code Generator API (MCG API)**: implements "Mnemonic code" generator with a pre-defined wordlist.

### 2.1.2 Technologies and Applications

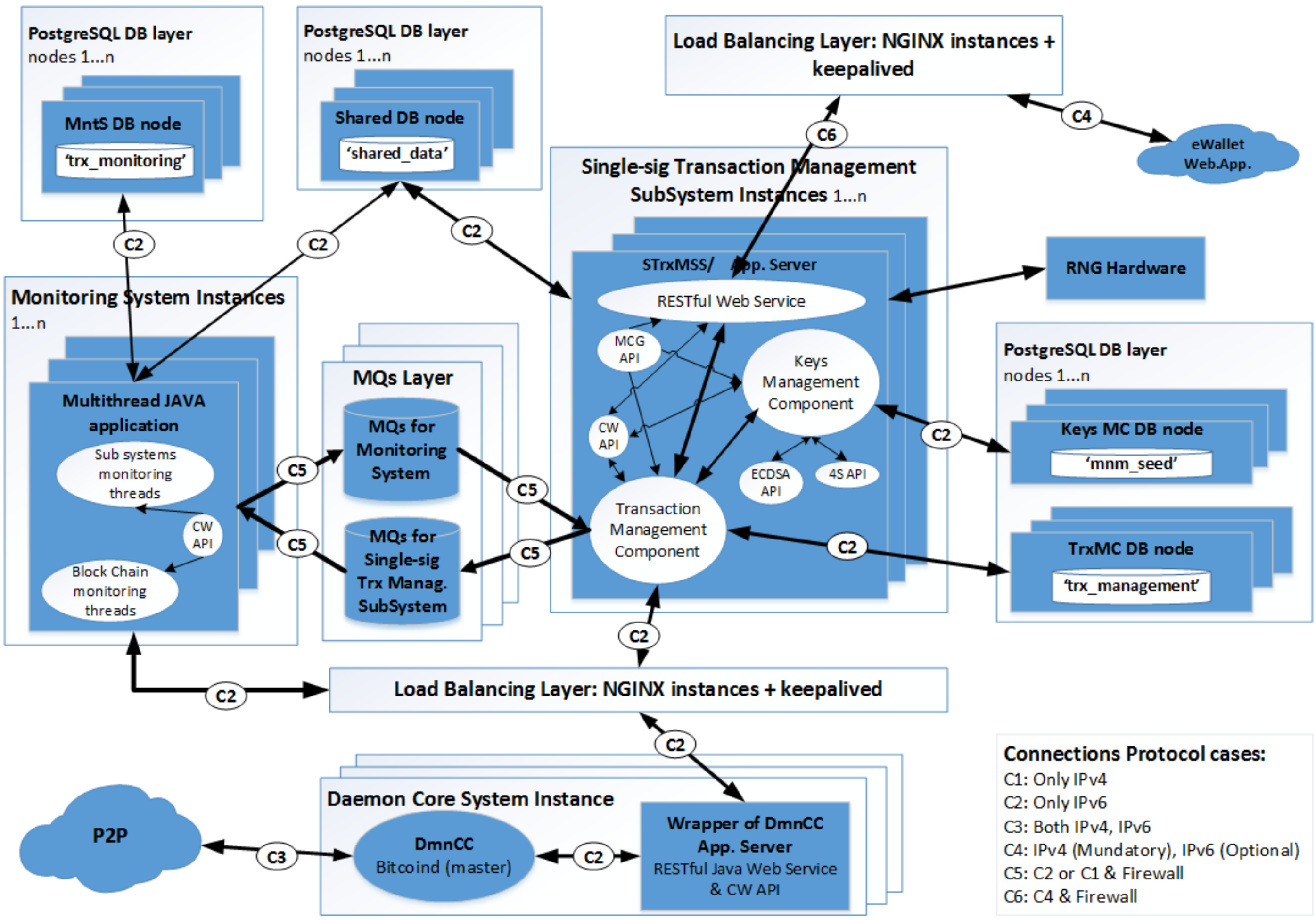
The technology stack used in the development of the Intelligent Daemon System includes the following pieces of software:

| # | Title | Latest Version (Oct. 2015) | Description |
|---|---|---|---|
| 1 | Java [1.7] | 8 | Programming language |
| 2 | JDK | 1.8.0_60 | Java Development Kit |
| 3 | JAX-RS [1.23] | 2.0 | JAX-RS is Java API for RESTful Services |
| 4 | Jersey [1.22] | 2.22.1 | Jersey RESTful Web Services framework is open source, production quality, framework for developing RESTful Web Services in Java that provides support for JAX-RS APIs and serves as a JAX-RS (JSR 311 & JSR 339) Reference Implementation. |
| 5 | Ant [1.24] | 1.9.6 | Apache Ant is a Java library and command-line tool. The main usage of Ant is the build of Java applications. Ant can also be used effectively to build non Java applications, for instance C or C++ applications. |
| 6 | Ivy [1.25] | 2.4.0 | Apache Ivy is a tool for managing (recording, tracking, resolving and reporting) project dependencies. Tight integration with Apache Ant |
| 7 | Groovy | 2.4 | Programming language |
| 8 | Spock [1.17] | 1.0 | Spock is a testing and specification framework for Java applications |
| 9 | JUnit [1.18] | 4.12 | Framework for tests and integration tests |
| 10 | Jmock [1.19] | 2.6.1 | Framework for tests and integration tests |
| 11 | PostgreSQL [1.6] | 9.4.5 | A free DB server |
| 12 | Slony-I [1.8] | 2.2.4 | Slony-I is a "master to multiple slaves" replication system for PostgreSQL supporting cascading (e.g. - a node can feed another node which feeds another node...) and failover. |
| 13 | Pgpool-II [1.9] | 3.4.0 | Multipurpose connect proxy for PostgreSQL |
| 14 | Tomcat [1.5] | 8.0.27 | FOS Java Servlet Container |
| 15 | NGINX [1.3] | 1.9.5 | HTTP and reverse proxy server that provides load balancing functionality on HTTP level. |
| 16 | Keepalived [1.10] | 1.2.19 | A routing software written in C for load balancing and high-availability to Linux system. |
| 17 | Bitcoind [1.2] | 0.11 | FOS program that implements the Bitcoin protocol for command line and RPC use. |
| 18 | RabbitMQ [1.4] | 3.5.6 | A free message broker |
| 19 | IPv4 | 4 | Internet Protocol version 4. IPv4 addresses may be written in any notation expressing a 32-bit integer value, but for human convenience, they are most often written in the dot-decimal notation, which consists of four octets of the address expressed individually in decimal and separated by periods. |
| 20 | IPv6 | 6 | Internet Protocol version 6. IPv6 addresses are represented as eight groups of four hexadecimal digits separated by colons, for example 2001:0db8:85a3:0042:1000:8a2e:0370:7334 |

## 2.2 Single-sig Transaction Management SubSystem

### 2.2.1 High Level of STrxMSS Architecture

The Single-sig Transaction Management System (STrxMSS): implements main business logic related to Btc Wallets, Btc Single-sig Transactions, Private/Public key pairs, Warm Storage. All the requests from eWallet Web App. are coming to STrxMSS via RESTful Web Services. STrxMSS involves other systems (DmnCS and MntS) to fulfil the request. STrxMSS uses Shared DB data and Shared libraries.
The diagram below (Pic. 2.2.1) shows the high level architecture of the proposed solution.




Pic. 2.2.1

### 2.2.2 Layers of STrxMSS Architecture

The STrxMSS consists of three layers:

- Load Balancing Layer
- Business Layer

- Databases Layer

**Load Balancing Layer** consists of multiple NGINX instances which are used as reverse proxy running on separate nodes. The synchronization between NGINX instances for handling failover is done using “Keepalived” utility. All the incoming HTTPS requests will be coming to Load Balancing Layer. Load balancer will redirect the request to appropriate Apache Tomcat Server instance via HTTP.

**Business Layer** consists of multiple instances of STrxMSS Java Web Application deployed on Apache Tomcat Server running on separate nodes. STrxMSS Java Web Application consists of:

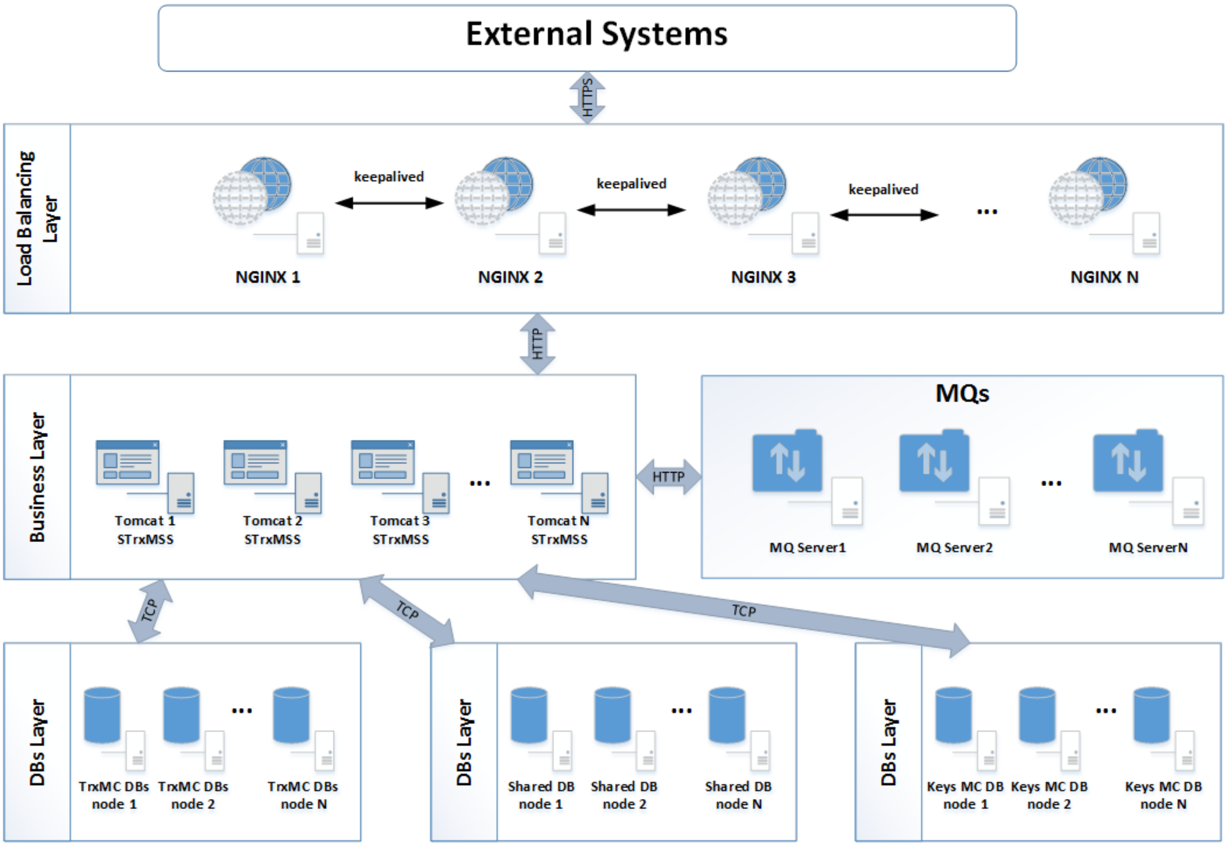
- RESTful Web Service
- Transaction Management Component (TrxMC)
- Keys Management Component (KeysMC)

STrxMSS Web Application is using CW API, 4S API, MGC API and ECDSA API as java libraries in the application build path.
STrxMSS is using RNG Hardware as random number generator.
STrxMSS Business Layer provides integration with MQs Layer to asynchronously interact with Monitoring System. STrxMSS Business Layer integrated with DBs Layer to store transactions and keys data. STrxMSS Business Layer is using data from shared DBs.

**Databases Layer** consists of three sets of DBs:

- Databases related to KeysMC business logic
- Databases related to TrxMC business logic
- Shared Databases related to IntDS errors and scripts types logic

The diagram below (Pic. 2.2.2) shows the principal architecture of the STrxMSS.




Pic. 2.2.2

### 2.2.3 Transaction Management Component

Transaction Management Component (TrxMC) provides core functionality related to Btc wallets balance and transactions.

1) TrxMC is integrated with KeysMC to sign transactions or create new Btc address for user wallet.
2) TrxMC fulfil requests to the Daemon Core System according to received requests from RESTful Web Service.
3) TrxMC creates response for RESTful Web Service according to request business logic.
4) TrxMC asynchronously interact with Monitoring System via MQs Layer. There are two main interactions:

- Sending of messages about new created transactions and Btc addresses which should be monitored. This is done using "strxmss_to_mnts_new_trxs" and "strxmss_to_mnts_new_btcaddr" queues in message broker.
- Receiving of messages about transactions to be monitored for confirmations from "mnts_to_strxmss_inb_trxs", "mnts_to_strxmss_reject_msg" and "mnts_to_strxmss_outb_trxs" queues.

MQs specification is described in "*MQs Layer*".

5) TrxMC is integrated with DB Layer which consists of several nodes. Each DB node consists of one Database:

- “trx_management” DB stores data related to Btc wallets balances, Btc transactions, etc.

6) TrxMC provide “Warm Storage” functionality.

### Warm Storage

IntDS will implement “Freezing Funds” solution according to BIP-0065 [2.24]. User will transfer BTC funds from a Wallet to the new Wallet’s BtcAddress with future unlocking date time in the Trx. Funds of this Trx will be frozen in the block chain without possibility to spend it until unlocking date.

**Note**: 1. Current project stage will implement “Warm Storage” functionality only in the GUI level by using IS_LOCKED boolean flag.
2. Back-end “Warm Storage” functionality will be developed and implemented in the future stage of project, when BIP-0065 will be approved from the Draft status.

Under construction…

The difference between “Cold Storage” and “Warm Storage” is:

*“Warm Storage” temporarily stores user’s funds and gives them back to the user when desired.*

*“Cold Storage” does the same. However it implements many more levels of security. Ex. private keys of 5 different people, required to sign a particular transaction at a particular time in a pre-defined physical location. This can (& most probably will) be an offline system with high level of physical security.*

## 2.2.4 Keys Management Component

Keys Management Component (KeysMC) provides functionality:

- to generate private key from mnemonic seed
- to generate public key and Btc address
- to securely store mnemonic seed parts
- to sign transactions

KeysMC is integrated with DB Layer which consists of several nodes. Each DBs node consists of one Database:

- “mnm_see” DB stores System’s parts of mnemonic seed

Each mnemonic seed part received by using Shamir’s Secret Sharing Scheme [3.4]. The whole seed can be restored by using user’s parts with system’s parts. User’s parts of seed will be unknown for IntDS till user insert it via GUI of external system. Private Key can be restored from the whole seed.

### 2.2.5 RESTful Web Sevice

All the requests from other external systems are coming to the STrxMSS via RESTful Web Service. The interface implementation must satisfy RESTful specification requirements. RESTful Web Service will return JASON objects according to other systems requests. The description of public API provided by STrxMSS is described in “*Single-sig Transaction Management SubSystem Interface*”

## 2.3 Accounting Transaction Management SubSystem

This point can be done in the scope of future development. Will need some researching activity.

## 2.4 Bank Transaction Management SubSystem

This point can be done in the scope of future development. Will need some researching activity.

## 2.5 Exchange Transaction Management SubSystem

This point can be done in the scope of future development. Will need some researching activity.

## 2.6 Message Transaction Management SubSystem

This point can be done in the scope of future development. Will need some researching activity.

## 2.7 Contracts Management SubSystem

This point can be done in the scope of future development. Will need some researching activity.

## 2.8 Monitoring System

The Monitoring System (MntS) consists of Business Layer and DBs Layer.

**Business Layer** consists of Multithread Java Application which fulfil requests periodically to the Daemon Core System for receiving data about current Block Chain situation and transactions. There are two types of threads which should implement business logic of monitoring.

Block Chain monitoring threads:

- **Out. Thread** monitors Outbound transactions in the Block Chain
- **Inb. Thread** monitors Inbound transactions in the Block Chain
- **Sync. Thread** monitors the sync and download of the blockchain
- **Sys. Start Thread** monitors the building of local blockchain when system starts for the first time.

General monitoring thread:

- **Log Thread** monitors log files of Daemon Core related to P2P Reject messages.

Sub systems monitoring threads:

- **STrxMSS Thread** monitors the receiving of messages from STrxMSS and adds data into “trx_monitoring” DB (tables with “STRXMSS_” name prefix)
- **ATrxMSS Thread** monitors the receiving of massages from ATrxMSS and adds data into “trx_monitoring” DB (tables with “ATRXMSS_” name prefix)
- **BTrxMSS Thread** monitors the receiving of massages from BTrxMSS and adds data into “trx_monitoring” DB (tables with “BTRXMSS_” name prefix)
- **ETrxMSS Thread** monitors the receiving of massages from ETrxMSS and adds data into “trx_monitoring” DB (tables with “EATRXMSS_” name prefix)
- **MTrxMSS Thread** monitors the receiving of massages from MTrxMSS and adds data “trx_monitoring” DB (tables with “MTRXMSS_” name prefix)
- **ContrMSS Thread** monitors the receiving of massages from ContrMSS and adds data into “trx_monitoring” DB (tables with “CONTRMSS_” name prefix)

MntS Business Layer provides integration with MQs Layer to asynchronously interact with each sub system. There are two main interactions:

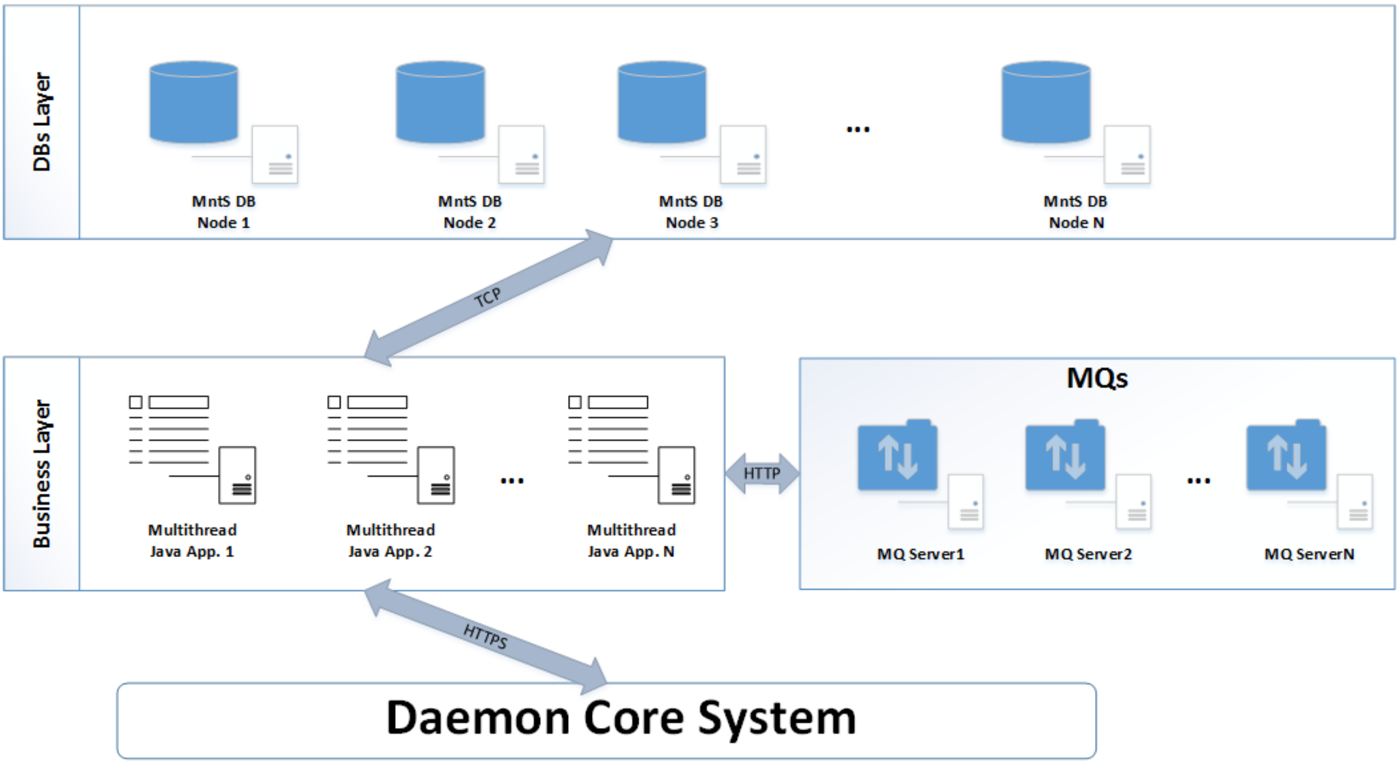
- Sending of messages about Inbound/Outbound transactions to be monitored for confirmations.
- Receiving of messages from each sub system about new created transactions and Btc addresses which should be monitored.

Out. and Inb. threads validate any new transactions which were received from block chain before sending data into MQ for each sub system. MQs specification is described in “*MQs Layer*”.

**Databases Layer** consists of two sets of DBs:

- Databases related to MntS business logic which consists of several nodes. Each DB node has “trx_monitoring” DB.
- Shared Databases related to IntDS errors and scripts types logic

The diagram below (Pic. 2.8.1) shows the principal architecture of the MntS.




Pic. 2.8.1

## 2.9 Daemon Core System

The Daemon Core System (DmnCS) provides functionality to synchronize with P2P network. The DmnCS consists of two layers:

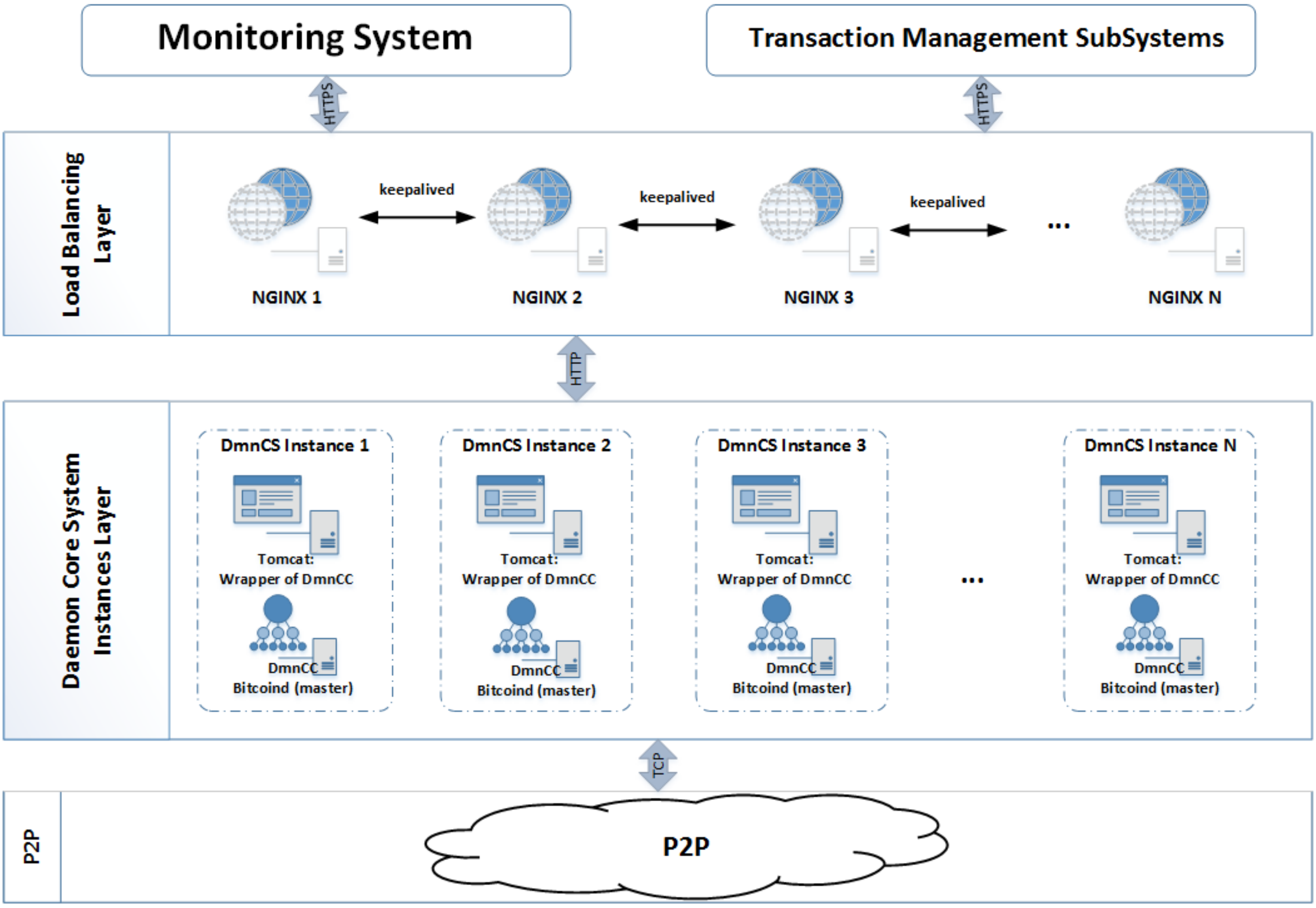
- DmnCS Instances Layer
- Load Balancing Layer

**DmnCS Instances Layer** provides core functionality to synchronize with P2P network. It consists of multiple DmnCS instances running on separate nodes. Each of the DmnCS instances includes:

- FOS Daemon Core Component (DmnCC) which is master part of bitcoind
- Wrapper of DmnCC which is RESTful Web Service deployed onto Tomcat Application Server

**Load Balancing Layer** consists of multiple NGINX instances which are used as reverse proxy running on separate nodes. The synchronization between NGINX instances for handling failover is done using “Keepalived” utility. All the incoming requests will be coming to Load Balancing Layer. Load balancer will redirect the request to appropriate daemon instance.

The diagram below (Pic. 2.9.1) shows the principal architecture of the Daemon Core System.




Pic. 2.9.1

### 2.9.1 FOS Daemon Core Component

FOS Daemon Core Component (DmnCC) is master part of FOS C++ bitcoind application. DmnCC should have part of OS bitcoind application functionality related to communications with P2P network. All functionality related to private/public keys creation is moved into KeysMC of STrxMSS or other sub-system. DmnCC does not have private/public keys creation logic anymore. DmnCC should care that new system transactions will be injected into Block Chain via Company Daemon Miner rather than via randomly found unknown Miner.

***Note:*** *There is parallel implementation of FOS Daemon (BIP1001) “Bitcoin XT” [2.20]. The capacity of block is 8Mb instead of 1Mb. This is not in the scope of current design. It should be considered in future project stages.*

### 2.9.2 Wrapper of Daemon Core Component

Wrapper of Daemon Core Component is RESTful Web Service deployed onto Tomcat Application Server.

All the requests from MntS and STrxMSS systems are coming to the Wrapper. The interface implementation must satisfy RESTful specification requirements. RESTful Web Service will return JASON objects according to systems requests. API repeats main functions of FOS Daemon Core Component. The description of public API provided by Wrapper is described in "*Daemon Core System Interface*".

# 2.10 Shared Libraries

## 2.10.1 Common Ware API

Common Ware API (**CW API**) is a Java Application: *commonWareAPI.jar* file. CW API should be included in the class path of the IntDS Java Applications as common library.

The table below describes CW API packages structure.

| Package name | Description | Classes | Classes Description |
|---|---|---|---|
| com.cwapi.support | classes responsible for support and services | EmailSender | Class allows the sending of simple text email. EmailSender class used JavaMail 1.4.7 API (mail.jar). API is licensed under the CDDL v1.1 and GPU v2 with Classpath Exception. |
| | | LogWriter | Class writes records to a log file of a different specified type. |
| | | PropertyLoader | Class loads properties from configuration file: IntDSConfig.properties |
| | | Constants | Class holds constants for this package. |
| | | | |
| com.cwapi.utils | shared utilities classes | DateTimeUtils | Class holds date and time utilities. |
| | | StringUtils | Class holds string utilities. |
| | | TypeConvertor | Class holds type casting and conversion utilities. |
| | | Utils | Class holds generic utilities. |
| | | | |
| com.cwapi.hashfnc | classes support different hash functions | SHA256 | Class holds converters to a hash by using SHA256 and SHA1 function. |
| | | RIPEMD160 | Class holds converters to a hash by using RIPEMD160 function. |

| | | | |
|---|---|---|---|
| | | Base58 | Class holds converters to Base58 and Base58Check encoding |
| | | | |

Under construction.

### 2.10.2 Shamir’s Secret Sharing Scheme API

Shamir’s Secret Sharing Scheme [3.4] API (4S API) is a Java Application: *ssssAPI.jar* file. The table below describes two main methods of 4S API.

| Method Name | Input Parameters | Return Value | Description |
|---|---|---|---|
| split | **secret** (String),<br>**N** (int) – number of total pieces, N=3 by default<br>**K** (int) – number of pieces for restoring the Secret, K < N, K=2 by default | JASON Object{<br>“**N**”: (int) number of total pieces,<br>“**K**”: (int) number of pieces for restoring Secret,<br>“**modulus**”: (int) random modulus for restoring the Secret,<br>“**partsArray**”: (JASON Array) (<br>{“ind”: 1, “arrValue”: “somevalue1”},<br>{“ind”: 2, “arrValue”: “somevalue2”},<br>…<br>{“ind”: K, “arrValue”: “somevalueK”},<br>), where array size is K<br>} | A Secret is divided into "N" pieces of data. Any "K" of those pieces can be used to reconstruct the Secret. |
| merge | **N** (int) – number of total pieces,<br>**K** (int) – number of pieces for restoring the Secret,<br>**modulus** (int) - random modulus for restoring the Secret<br>**partsArray** (ArrayList<SSSSPoint>) - array list of the Secret pieces, where size is K, SSSSPoint is Object with members:<br>xValue (int) – X coordinate and integer started from 1<br>yValue (String) - Y coordinate in the string representation | Secret (String) | Function reconstructs the Secret from pieces. |

### 2.10.3 ECDSA API

Elliptic Curve Digital Signature Algorithm API (ECDSA API) is Java Application: *ecdsaAPI.jar* file.

The table below describes functions of ECDSA API business logic.

| Function Name | Input Parameters | Return Value | Description |
|---|---|---|---|
| Private Key generator | -- | Private Key (int) - 256-bit integer (BigInteger in Java) in the range $[1, 2^{256}]$ | Function generates randomly 256-bit integer in the range $[1, 2^{256}]$ |

| | | | |
|---|---|---|---|
| Deterministic Private Key maker | **Mnemonic Seed** (String) – string consists of mnemonic code + ASCII-coded number | Private Key (int) - 256-bit integer (BigInteger in Java) in the range [1, $2^{256}$] | Function makes 256-bit integer in the range [1, $2^{256}$] by using deterministic approach - SHA256(Mnemonic Seed) |
| Points Additions | ECDSAPoint **P(x1, y1)** – ECDSAPoint object (java bean) with affine coordinates x1 and y1; ECDSAPoint **P(x2, y2)** – ECDSAPoint object (java bean) with affine coordinates x2 and y2. Arguments are points on the elliptic curve | ECDSAPoint P(x3, y3) - ECDSAPoint object (java bean) with affine coordinates x3 and y3, which is point on the elliptic curve and result of points additions operation. | Function implements “Points Additions” formula. Function produce P(x3, y3) as result of points additions |
| Points Doubling | ECDSAPoint **P(x1, y1)** – ECDSAPoint object (java bean) with affine coordinates x1 and y1, which is point on the elliptic curve | ECDSAPoint P(x3, y3) - ECDSAPoint object (java bean) with affine coordinates x3 and y3, which is point on the elliptic curve | Function implements “Points Doubling” formula. Function produce P(x3, y3) as result of point doubling |
| Unified formula | ECDSAPoint **P(x1, y1)** – ECDSAPoint object (java bean) with affine coordinates x1 and y1; ECDSAPoint **P(x2, y2)** – ECDSAPoint object (java bean) with affine coordinates x2 and y2. Arguments are points on the elliptic curve | ECDSAPoint P(x3, y3) - ECDSAPoint object (java bean) with affine coordinates x3 and y3, which is point on the elliptic curve | Function implements “Unified” formula which can be used instead “Points Additions” or “Points Doubling”. |
| Simultaneous Scalar Multiplication | ECDSAPoint **P(x1, y1)** – ECDSAPoint object (java bean) with affine coordinates x1 and y1; ECDSAPoint **P(x2, y2)** – ECDSAPoint object (java bean) with affine coordinates x2 and y2. P(x1, y1) and P(x2, y2) are points on the elliptic curve<br>**k** (int, BigInteger in Java) – multiplication number for point P(x1, y1);<br>**l** (int, BigInteger in Java) - multiplication number for point P(x2, y2) | ECDSAPoint P(x3, y3) - ECDSAPoint object (java bean) with affine coordinates x3 and y3, which is point on the elliptic curve and result of Simultaneous Scalar Multiplication operation. P(x3, y3) = k P(x1, y1) + l P(x2, y2) | Function implements Simultaneous Scalar Multiplication according to Straus's algorithm |
| Compressed Pub Key maker | **privKey** (int, BigInteger in Java) – Private Key, 256-bit integer in the range 1<= PrvKey<=$2^{256}$ | Public Key (String) - hexadecimal string of Pub Key in the Compressed form | Function makes compressed form of Public Key for the script |
| Uncompressed Pub Key maker | **privKey** (int, BigInteger in Java) – Private Key, 256-bit integer in the range 1<= PrvKey<=$2^{256}$ | Public Key (String) - hexadecimal string of Pub Key in Uncompressed form | Function makes uncompressed form of Public Key for the script |
| Generate Signature | **message** (byte[]) - the SHA-1 hash of the message/transaction that should be signed;<br>ECDSAPrivateKey **privKey** – private key | (r, s) (BigInteger[]{r, s}) - Signature as big integer pair (r, s) | Function generates a signature as a pair of integers for the given message/transaction using the private key. |
| Generate DER-encoded Signature | **message** (byte[]) - the SHA-1 hash of the message/transaction that should be signed; | sig (String) - DER-encoding of signature par (r, s) for the script | Function generates a DER-encoded signature for the given message/transaction using the private key. |

| | | | |
|---|---|---|---|
| | ECDSAPrivateKey **privKey** – private key | | |
| Signature Verification | **(r, s)** (BigInteger[]{r, s}) - Signature as big integer pair (r, s); **message** (byte[]) - the SHA-1 hash of the message/transaction for which signature should be verified; ECDSAPoint **Q(x, y)** – Public Key, ECDSAPoint object (java bean) with affine coordinates x and y. | Verification flag (boolean) – true if signature is valid and corresponds to given Public Key, otherwise false | Function verify that given signature match the given Public Key. |
| Signature Verification in scripts | **pubKeyStr** (String) – Public Key in the Script representation; **sigStr** (String) – DER-encoded signature in the Script representation; **message** (byte[]) - the SHA-1 hash of the message/transaction for which signature should be verified | Verification flag (boolean) – true if signature is valid and corresponds to given Public Key, otherwise false | Function verify that given signature match the given Public Key. The input parameters are given in the Script representation. |

### 2.10.4 Mnemonic Code Generator API

Mnemonic Code Generator API is a Java Application: *mnmCodeAPI.jar* file.

| Function Name | Input Parameters | Return Value | Description |
|---|---|---|---|
| Generate Mnemonic Code as string array | • Language (String) – 3 chars string to select what language wordlist to use, English as “eng” by default<br>• no_of_words (int) – number of words required in the mnemonic code in the range 12 to 24. Default number is 12. For validations, refer section 6.1.1 | mnm_code (String array) – Array of strings containing the required number of words from the selected wordlist | Generates a sequence of words as part of the mnemonic code from a pre-defined wordlist. |
| Generate Mnemonic Code as string | • Language (String) – 3 chars string to select what language wordlist to use, English as “eng” by default<br>• no_of_words (int) – number of words required in the mnemonic code in the range 12 to 24. Default number is 12. For validations, refer section 6.1.1 | mnm_code (String) – String containing the required number of words from the selected wordlist | Generates a sequence of words as part of the mnemonic code from a pre-defined wordlist. |

# 2.11 Configurations and Logs

## 2.11.1 Main Configuration File

MntSConfig.properties

Configurable parameters must be included:

Company fee: 7000 Satoshi =0.00007 Btc

Change can not be less than 546 Satoshi = 0.00000546 Btc

MQ Consumer Thread sleep time in milliseconds.

Under construction…

## 2.11.2 Log types and rules

The LogWriter class (see CW API) responsible for creation of log files for every java components. The location of log files directory is the same as application location:

../[application location]/..

../CFGDATA/LogsDir/*.log

There are different types of logs depending on function logic. Different logs can be recognized by application or component prefix and log type.

*Log file name = [*Application or Component Prefix*]-[Log Type]current.log*

List of logs:

| **Component Prefix** | **Log Type** | **Log File** | **Description** |
|---|---|---|---|
| MntS | ERROR | *MntS-ERRORcurrent.log* | Monitoring System general errors logs |
| MntS | DEBUG | *MntS-DEBUGcurrent.log* | Monitoring System general debug logs. These logs will be written if debug setting is switched on in the MntSConfig.properties |
| MntS | INFO | *MntS-INFOcurrent.log* | Monitoring System general information logs |
| MntS | DB-ERR | *MntS-DB-ERRcurrent.log* | Monitoring System DB errors logs (connections etc.) |

| | | | |
|---|---|---|---|
| MntS | MQ-ERR | *MntS-MQ-ERRcurrent.log* | Monitoring System MQ errors logs (connections, message was not sent etc.) |
| STrxMSS | ERROR | *STrxMSS-ERRORcurrent.log* | Single-sig Trx Management SubSystem general errors logs |
| STrxMSS | DEBUG | *STrxMSS-DEBUGcurrent.log* | Single-sig Trx Management SubSystem general debug logs. These logs will be written if debug setting is switched on in the MntSConfig.properties |
| STrxMSS | INFO | *STrxMSS-INFOcurrent.log* | Single-sig Trx Management SubSystem general information logs |
| STrxMSS | DB-ERR | *STrxMSS-DB-ERRcurrent.log* | Single-sig Trx Management SubSystem DB errors logs (connections etc.) |
| STrxMSS | MQ-ERR | *STrxMSS-MQ-ERRcurrent.log* | Single-sig Trx Management SubSystem MQ errors logs (connections, message was not sent etc.) |
| MCGAPI | ERROR | *MCGAPI-ERRORcurrent.log* | Mnemonic Code Generator API general errors logs |
| MCGAPI | DEBUG | *MCGAPI-DEBUGcurrent.log* | Mnemonic Code Generator API general debug logs. These logs will be written if debug setting is switched on in the MntSConfig.properties |
| MCGAPI | INFO | *MCGAPI-INFOcurrent.log* | Mnemonic Code Generator API general information logs |
| KeysMC | ERROR | *KeysMC-ERRORcurrent.log* | Keys Management Component general errors logs |
| KeysMC | DEBUG | *KeysMC-DEBUGcurrent.log* | Keys Management Component general debug logs. These logs will be written if debug setting is switched on in the MntSConfig.properties |
| KeysMC | INFO | *KeysMC-INFOcurrent.log* | Keys Management Component general information logs |
| KeysMC | DB-ERR | *KeysMC-DB-ERRcurrent.log* | Keys Management Component DB errors logs (connections etc.) |
| CWAPI | ERROR | *CWAPI-ERRORcurrent.log* | Common Ware API general errors logs |
| CWAPI | DEBUG | *CWAPI-DEBUGcurrent.log* | Common Ware API general debug logs. These logs will be written if debug setting is switched on in the MntSConfig.properties |
| CWAPI | INFO | *CWAPI-INFOcurrent.log* | Common Ware API general information logs |
| CWAPI | DB-ERR | *CWAPI-DB-ERRcurrent.log* | Common Ware API DB errors logs (connections etc.) |
| ECDSA API | ECDSA-ERR | ECDSA-*ERRcurrent.log* | ECDSA API errors logs |

| | | | |
|---|---|---|---|
| 4S API | 4SAPI-ERR | 4SAPI-*ERRcurrent.log* | 4S API errors logs |

Each record in the log file will be separate line. The format of the record will be as follows:

Timestamp in format “yyyyMMddHHmmss”: <Type of problem>: <Class name>.<Method name> : <Log message>

Example:

20151228142020: ERROR: LogWriter().log: Error writing to log file “*KeysMC-INFOcurrent.log*”

If the log file becomes larger than “maximum log file size” (1000000 bytes by default) it is renamed with an archive date/time and new log file is started:

*Archived Log file name = [*Application or Component Prefix*]-[Log Type][timestamp].log*

Additional debug lines can be written in the any type of logs if debug setting is switched on in the Config.properties

Additional new types of logs can be created in development process if new type is logically needed.

## 2.12 MQs Layer

Naming rules in the prefixes of queues:

The queue name should be made according to formula:

*[Sub-system abbreviation of a Producer]_to_[Sub-system abbreviation of a Consumer]_[object/item should be monitored]*

Sub systems Queues prefixes:

- **STrxMSS Queues:** “strxmss_to_”
- **ATrxMSS Queues:** “atrxmss_to_”
- **BTrxMSS Queues:** “btrxmss_to_”
- **ETrxMSS Queues:** “etrxmss_to_”
- **MTrxMSS Queues:** “mtrxmss_to_”
- **ContrMSS Queues:** “contrmss_to_”
- **MntS Queues:** “mnts_to_”

Sub-systems and monitoring system send objects with different properties. The object needs to be serialised into a byte array so that it can be included in the message body. The serialised object needs to be deserialised in the receiving part. See example on [1.13].

### 2.12.1 MQ Specification for Single-sig Transaction Management SubSystem

STrxMSS is producer MntS is consumer in this scenario. Sending of messages about new created Outbound transactions and Btc addresses which should be monitored is done by using “strxmss_to_mnts_new_trxs” and “strxmss_to_mnts_new_btcaddr” queues in message broker.

**“strxmss_to_mnts_new_trxs”** queue: object specification of message:

| Field Title | Java Type | Length | Description |
|---|---|---|---|
| | | | |

**“strxmss_to_mnts_new_btcaddr”** queue: object specification of message:

| Field Title | Java Type | Length | Description |
|---|---|---|---|
| | | | |

Under construction…

### 2.12.2 MQ Specification for Accounting Transaction Management SubSystem

This point can be done in the scope of future development. Will need some researching activity.

### 2.12.3 MQ Specification for Bank Transaction Management SubSystem

This point can be done in the scope of future development. Will need some researching activity.

### 2.12.4 MQ Specification for Exchange Transaction Management SubSystem

This point can be done in the scope of future development. Will need some researching activity.

### 2.12.5 MQ Specification for Message Transaction Management SubSystem

This point can be done in the scope of future development. Will need some researching activity.

### 2.12.6 MQ Specification for Contracts Management SubSystem

This point can be done in the scope of future development. Will need some researching activity.

### 2.12.7 MQ Specification for Monitoring System

MntS is producer STrxMSS is consumer in this scenario.

Inbound transactions queue in message broker is “**mnts_to_strxmss_inb_trxs**”. Object specification of message:

| Field Title | Java Type | Length | Description |
|---|---|---|---|
| *daemonTxidHash* | String | | |
| *confirmations* | int | | |
| *walletId* | | | |
| *btcAddress* | | | |
| | | | |

Outbound transactions queue in message broker is “**mnts_to_strxmss_outb_trxs**”. Object specification of message:

| Field Title | Java Type | Length | Description |
|---|---|---|---|
| *daemonTxidHash* | String | | |
| *confirmations* | int | | |
| *blockHash* | String | | |

Log files and sending reject messages queue in message broker is “**mnts_to_strxmss_reject_msg**”. Object specification of message:

| Field Title | Java Type | Length | Description |
|---|---|---|---|
| *daemonTxidHash* | String | | |
| *rejectMsgId* | int | | |

# 3. Databases

IntDS does not use any ORM frameworks and JPA to interact with DBs for security and performance reasons. IntDS uses JDBC approach and DAO pattern instead.

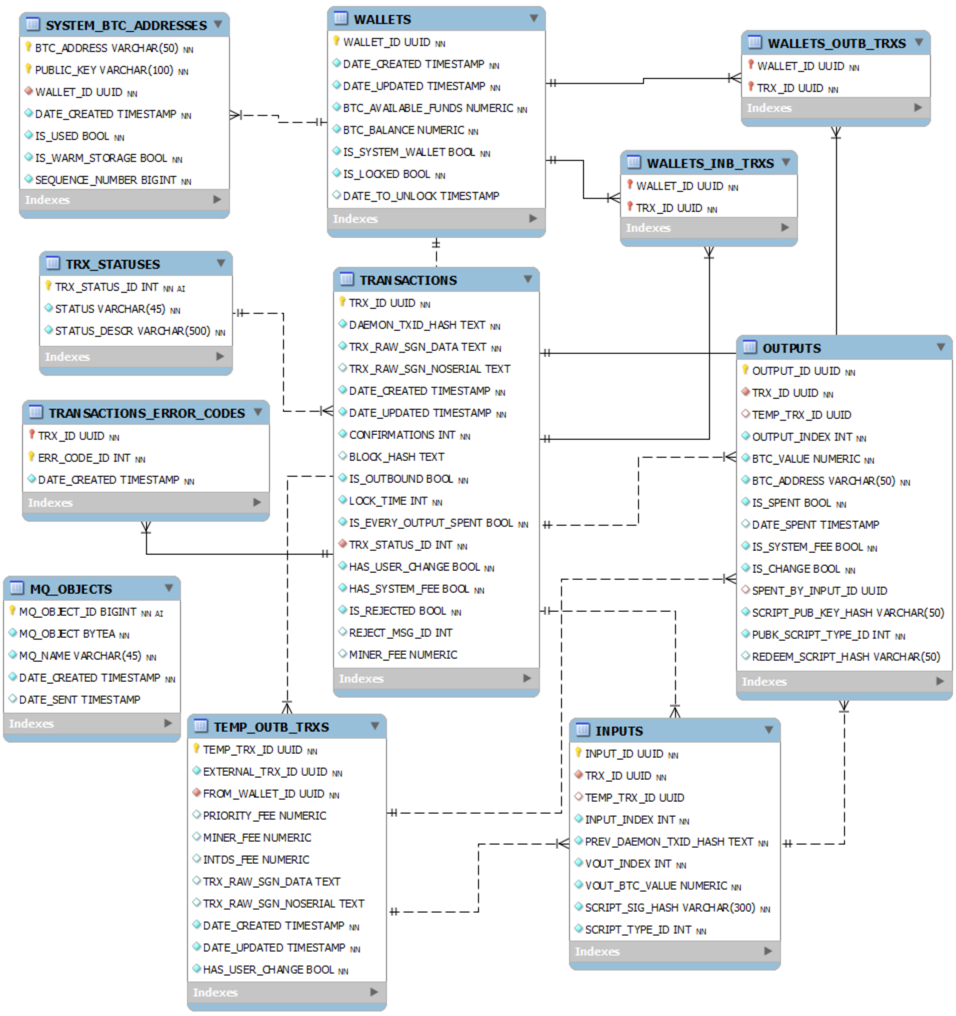
## 3.1 Single-sig Transaction Management SubSystem DBs

The Single-sig Transaction Management SubSystem (STrxMSS) has 2 Databases. **“mnm_seed”** interact with Keys Management Component. “**trx_management**” DB interact with Transaction Management Component.

### 3.1.1 Transaction Management Component DBs Diagram

**“trx_management” DB** consists of 11 tables and stores transactions data. The diagram below (Pic. 3.1.1) shows the DB structure.

*Update frequency*: fast changing data.




Pic. 3.1.1

### 3.1.2 Transaction Management Component DBs Description

**1) “trx_management” DB tables description**:

**1. MQ_OBJECTS** table holds binary objects of messages which were not sent to MQ for MntS in time because message producer had error. Separate thread should send these messages later for MntS.

*MQ_OBJECTS table consists of below fields:*

| Field Title | DB Type | Java type | Not Null | Description |
|---|---|---|---|---|
| MQ_OBJECT_ID | bigint | Long | true | **Primary Key,** Message object identifier. |
| MQ_OBJECT | bytea | byte<br>use `getBytes()`, `setBytes()`, `getBinaryStream()`, or `setBinaryStream()` methods | true | Binary object of message which should be sent to MQ |
| MQ_NAME | varchar(45) | String | true | Queue name |
| DATE_CREATED | timestamp(6) | String | true | Date-time of record creation |
| DATE_SENT | timestamp(6) | String | false | Date-time when thread sends this object to MQ |

*MQ_OBJECTS table has indexes:*

| Index Name | Fields |
|---|---|
| PRIMARY | MQ_OBJECT_ID |

**2. INPUTS** table holds data of transaction Inputs. Inputs - records which reference the funds, messages, hash info from other previous transactions.

*INPUTS table consists of below fields:*

| Field Title | DB Type | Java type | Not Null | Description |
|---|---|---|---|---|
| INPUT_ID | UUID | java.util.UUID -> setObject/getObject in JDBC | true | **Primary Key,** the Input identifier. |
| TRX_ID | UUID | java.util.UUID -> setObject/getObject in JDBC | true | Transaction identifier of the STrxMSS, PK from TRANSACTIONS table, dependency with System transaction |

| | | | | |
|---|---|---|---|---|
| TEMP_TRX_ID | UUID | java.util.UUID -> setObject/getObject in JDBC | false | Temporary Trx identifier, PK from TEMP_OUTB_TRXS table, dependency with data of temporary outbound transaction. Zero by default. |
| INPUT_INDEX | int | int | true | Index in the Inputs array |
| PREV_DAEMON_TXID_HASH | text | String | true | Identifier of the prior referenced transaction (“txid”: hash in hex of signed transaction). Transaction id is received from Block Chain via the DmnCS. A hash of completed transaction which allows other transactions to spend its outputs. Maximum size – 1,000,000 |
| VOUT_INDEX | int | int | true | Index of valid unspent Output in the referenced transaction. The referenced transaction (DAEMON_TXID_HASH) may have more than one Output so the index is used to identify which Output is being “spent” for System transaction (TRX_ID). Zero by default. |
| VOUT_BTC_VALLUE | numeric | BigDecimal | true | Btc amount of the valid unspent Output in the referenced transaction (DAEMON_TXID_HASH). Output has index = VOUT_INDEX. Zero by default. |
| SCRIPT_SIG_HASH | varchar(300) | String | true | Current Input’s scriptSig value in hash as part of transaction hash string |
| SCRIPT_TYPE_ID | int | int | true | Script pair type Identifier. PK from SCRIPT_PAIRS_TYPES |

| | | | | |
|---|---|---|---|---|
| | | | | table, dependency with script pair type from "shared_data" DB. Zero by default. Script pair types with descriptions can be found in the Appendix E |

*INPUTS table has foreign keys:*

| Foreign Key Name | Referenced Table | Field | Referenced Field |
|---|---|---|---|
| TRX_ID_INPUTS_FK | **TRANSACTIONS** | TRX_ID | TRX_ID |
| TEMP_TRX_ID_INPUTS_FK | **TEMP_OUTB_TRXS** | TEMP_TRX_ID | TEMP_TRX_ID |

*INPUTS table has unique indexes:*

| Index Name | Field |
|---|---|
| INPUT_ID_UNIQUE | INPUT_ID |
| DAEMON_TXID_UNIQUE | DAEMON_TXID |

*INPUTS table has indexes:*

| Index Name | Fields |
|---|---|
| PRIMARY | INPUT_ID |
| TRX_ID_INPUTS_FK_IDX | TRX_ID |
| TEMP_TRX_ID_INPUTS_FK_IDX | TEMP_TRX_ID |

**3. OUTPUTS** table holds data of transaction Outputs. Outputs - records which determine the new owner of the transferred Bitcoins, and which will be referenced as Inputs in future transactions as those funds are respent. Outputs are tied to transaction identifiers (TXIDs), which are the hashes of signed transactions.

*OUTPUTS table consists of below fields:*

| Field Title | DB Type | Java type | Not Null | Description |
|---|---|---|---|---|
| OUTPUT_ID | UUID | java.util.UUID -> setObject/getObject in JDBC | true | **Primary Key,** the Output identifier. Unique |
| TRX_ID | UUID | java.util.UUID -> setObject/getObject in JDBC | true | Transaction identifier of the STrxMSS, PK from TRANSACTIONS table, dependency with System transaction |

| | | | | |
|---|---|---|---|---|
| TEMP_TRX_ID | UUID | java.util.UUID -> setObject/getObject in JDBC | false | Temporary Trx identifier, PK from TEMP_OUTB_TRXS table, dependency with data of temporary outbound transaction. Zero by default. |
| OUTPUT_INDEX | int | int | true | Index in the Outputs array |
| BTC_VALLUE | numeric | BigDecimal | true | Transferred Bitcoins amount |
| BTC_ADDRESS | varchar(50) | String | true | Bitcoin address is the recipient of the funds. |
| IS_SPENT | bool | boolean | true | Each Output from one transaction can only ever be referenced once by an input of a subsequent transaction. This field is true (1) if Output was used by an input of a subsequent confirmed transaction, otherwise false (0). Default value is false (0). |
| DATE_SPENT | timestamp(6) | String | false | Null by default. Date and time when Output was spent. |
| IS_SYSTEM_FEE | bool | boolean | true | System fee for transaction is sent to System Btc address as one of transaction's Output. This output is a system transaction fee if this field is true (1), otherwise false (0). Default value is false (0). |
| IS_CHANGE | bool | boolean | true | User change should be send back to User Wallet. This Output is User change if this field is true (1), otherwise false (0). Default value is false (0). |
| SPENT_BY_INPUT_ID | UUID | java.util.UUID -> setObject/getObject in JDBC | false | Input identifier if this Output was spent by STrxMSS transaction, PK |

| | | | | |
|---|---|---|---|---|
| | | | | from INPUTS table, dependency with Input of Outbound transaction. |
| SCRIPT_PUB_KAEY_HASH | varchar(50) | String | true | Current Output’s scriptPubKey value in hash as part of transaction hash string |
| PUBK_SCRIPT_TYPE_ID | int | int | true | Script pair type Identifier. PK from SCRIPT_PAIRS_TYPES table, dependency with script pair type from “shared_data” DB. Zero by default. Script pair types with descriptions can be found in the Appendix E |
| REDEEM_SCRIPT_HASH | varchar(50) | String | false | Current Output’s redeemScript value in 20-byte hash. Redeem script value is required if the funds is spending from multi-sig btc address, otherwise null |

*OUTPUTS table has foreign keys:*

| Foreign Key Name | Referenced Table | Field | Referenced Field |
|---|---|---|---|
| TRX_ID_OUTPUTS_FK | **TRANSACTIONS** | TRX_ID | TRX_ID |
| BY_INPUT_ID_INPUTS_FK | **INPUTS** | SPENT_BY_INPUT_ID | INPUT_ID |
| TEMP_TRX_ID_OUTPUTS_FK | **TEMP_OUTB_TRXS** | TEMP_TRX_ID | TEMP_TRX_ID |

*OUTPUTS table has indexes:*

| Index Name | Fields |
|---|---|
| PRIMARY | OUTPUT_ID |
| TRX_ID_OUTPUT_FK_IDX | TRX_ID |
| BY_INPUT_ID_INPUTS_FK_IDX | SPENT_BY_INPUT_ID |
| TEMP_TRX_ID_OUTPUTS_FK_IDX | TEMP_TRX_ID |

**4. SYSTEM_BTC_ADDRESSES** table holds Btc addresses data and mapping with data from WALLETS table. Single-sig Btc address is a 160-bit hash of the public portion of a public/private ECDSA key pair. Single-sig

Btc address is generated by STrxMSS. (***Note:*** *Table holds only Single-sig Btc addresses. Multi-sig addresses are not in the scope of STrxMSS. The first digit in the Multi-sig address is a “3” to validate data.*).

*SYSTEM_BTC_ADDRESSES table consists of below fields:*

| Field Title | DB Type | Java type | Not Null | Description |
|---|---|---|---|---|
| BTC_ADDRESS | varchar(50) | String | true | **Primary key**. Btc address hash string. Bitcoin addresses are used to receive payments, message, hash info. |
| PUBLIC_KEY | varchar(100) | String | true | **Primary key**. Public key string that corresponds to a private key, but does not need to be kept secret. A public key can be used to determine if a signature is genuine without requiring the private key to be divulged. |
| WALLET_ID | UUID | java.util.UUID -> setObject/ getObject in JDBC | true | Wallet identifier, PK from WALLETS table, dependency with wallets. |
| DATE_CREATED | timestamp(6) | String | true | Date-time of record creation |
| IS_USED | bool | boolean | true | This field is true (1) if Btc address was used in the Outbound transaction, otherwise false (0). Default value is false (0). |
| IS_WARM_STORAGE | bool | boolean | true | This field is true (1) if Btc address is used in the “Warm Storage” output transaction, otherwise false (0). This BTC_ADDRESS data should be equal to data in COLD_STORAGE_TRXS_OUT table, BTC_ADDR_TO field in this case. Default value is false (0). |
| SEQUENCE_NUMBER | bigint | Long | true | Sequence Number of every Btc address for particular wallet. This number will be used to generate private key. Default value is 0. |

*SYSTEM_BTC_ADDRESSES table has foreign keys:*

| Foreign Key Name | Referenced Table | Field | Referenced Field |
|---|---|---|---|
| WALLET_ID_SYS_BTC_ADDR | **WALLETS** | WALLET_ID | WALLET_ID |

*SYSTEM_BTC_ADDRESSES table has indexes:*

| Index Name | Fields |
|---|---|
| PRIMARY | BTC_ADDRESS, PUBLIC_KEY |
| WALLET_ID_SYS_BTC_ADDR_IDX | WALLET_ID |

**5. TEMP_OUTB_TRXS** table holds data about temporary outbound transactions. IntDS creates new Trx by several stages. There is a preparation stage which calculate Miner and company fee depending on Trx size. Calculated fee should be included in Trx. External system should accept payment of fee or reject Trx. This table holds new trx data till it is confirmed or rejected. Data will be deleted after that. Data will be copied into TRANSACTIONS table if External system will confirm Trx.

*TEMP_OUTB_TRXS table consists of below fields:*

| Field Title | DB Type | Java type | Not Null | Description |
|---|---|---|---|---|
| TEMP_TRX_ID | UUID | java.util.UUID -> setObject/getObject in JDBC | true | **Primary Key,** the Temporary outbound transaction identifier. |
| EXTERNAL_TRX_ID | UUID | java.util.UUID -> setObject/getObject in JDBC | true | Transaction identifier from External system. Maping between IntDS and External system data |
| FROM_WALLET_ID | UUID | java.util.UUID -> setObject/getObject in JDBC | true | Identifier of wallet from which Btc funds will be sent. PK from WALLETS table, dependency with wallets. |
| PRIORITY_FEE | numeric | BigDecimal | false | Fee was inserted by External system, which should be paid for this transaction. Zero by default. |
| MINER_FEE | numeric | BigDecimal | false | Miner’s fee is calculated by IntDS depending on trx size, which should be paid for this transaction. Currently 0.0001 Btc per each 1 kByte. Zero if trx |

| | | | | |
|---|---|---|---|---|
| | | | | size less than 1 kByte. Zero by default. |
| INTDS_FEE | numeric | BigDecimal | false | IntDS’s fee is calculated by IntDS depending on trx size, which should be paid for this transaction. Currently 0.00007 Btc per each 1 kByte. Zero if trx size less than 1 kByte. Zero by default. |
| TRX_RAW_SGN_DATA | text | String | false | Raw byte data of the signed Transaction (hex-encoded string) after serialization. Zero by default. |
| TRX_RAW_SGN_NOSERIAL | text | String | false | Raw byte data of the signed Transaction (hex-encoded string) before serialization. |
| DATE_CREATED | timestamp(6) | String | true | Date-time of record creation |
| DATE_UPDATED | timestamp(6) | String | true | Date-time of transaction record last update |
| HAS_USER_CHANGE | bool | boolean | true | True (1) if one Output of this transaction is User’s change otherwise false (0). Default value is false (0). |

*TEMP_OUTB_TRXS table has foreign keys:*

| Foreign Key Name | Referenced Table | Field | Referenced Field |
|---|---|---|---|
| TEMP_OUTB_TRX_WALLETS_FK | **WALLETS** | FROM_WALLET_ID | WALLET_ID |

*TEMP_OUTB_TRXS table has indexes:*

| Index Name | Fields |
|---|---|
| PRIMARY | TEMP_TRX_ID |
| TEMP_OUTB_TRX_WALLETS_FK_IDX | FROM_WALLET_ID |

**6. TRANSACTIONS** table holds data about System transactions.

*TRANSACTIONS table consists of below fields:*

| Field Title | DB Type | Java type | Not Null | Description |
|---|---|---|---|---|
| TRX_ID | UUID | java.util.UUID -> setObject/getObject in JDBC | true | **Primary Key,** the Transaction identifier. |
| DAEMON_TXID_HASH | text | String | true | Transaction Identifier is a hash of completed transaction which allows other transactions to spend its outputs. Transaction Identifier is received from Block Chain via the DmnCS. Zero by default. Maximum size - 1,000,000 chars |
| TRX_RAW_SGN_DATA | text | String | true | Raw byte data of the signed Transaction (hex-encoded string) after serialization.<br>Zero by default. |
| TRX_RAW_SGN_NOSERIAL | text | String | false | Raw byte data of the signed Transaction (hex-encoded string) before serialization. |
| DATE_CREATED | timestamp(6) | String | true | Date-time of record creation |
| DATE_UPDATED | timestamp(6) | String | true | Date-time of transaction record last update |
| CONFIRMATIONS | int | int | true | Number of new blocks in Block Chain after the transaction has been included in the block and block was published to the network. Confirmations is received from Block Chain via the DmnCS. The transaction should be considered as **confirmed** if it is a six number of blocks deep. Zero by default. |
| BLOCK_HASH | text | String | false | Block Identifier is a hash of block in which transaction was included. Maximum size - 2,000,000 chars |

| | | | | |
|---|---|---|---|---|
| IS_OUTBOUND | bool | boolean | true | True (1) if transaction is outbound trx otherwise false (0). Default value is true (1). |
| LOCK_TIME | int | int | true | The block number or timestamp at which this transaction is locked, or 0 if the transaction is always locked. A non-locked transaction must not be included in blocks, and it can be modified by broadcast ting a new version before the time has expired. Default value is zero. |
| IS_EVERY_OUTPUT_SPENT | bool | boolean | true | True (1) if every Outputs of this transaction are spent otherwise false (0). Default value is false (0). |
| TRX-STATUS_ID | int | int | true | Transaction status Identifier. PK from TRX_STATUSES table, dependency with transaction statuses. Zero by default. |
| HAS_USER_CHANGE | bool | boolean | true | True (1) if one Output of this transaction is User’s change otherwise false (0). Default value is false (0). |
| HAS_SYSTEM_FEE | bool | boolean | true | True (1) if one Output of this transaction is IntDS fee otherwise false (0). Default value is false (0). |
| IS_REJECTED | bool | boolean | true | True (1) if this transaction is rejected by blockchain otherwise false (0). Default value is false (0). |
| REJECT_MSG_ID | int | int | false | Rejection message identity number, PK from BTC_REJECTION_MSG table, dependency with blockchain rejection |

| | | | | |
|---|---|---|---|---|
| | | | | messages from “shared_data” DB. Rejection messages with descriptions can be found in the Appendix M |
| MINER_FEE | numeric | BigDecimal | false | Bitcoins amount should be paid as Miner fee. Zero by default. |

*TRANSACTIONS table has foreign keys:*

| Foreign Key Name | Referenced Table | Field | Referenced Field |
|---|---|---|---|
| TRX_STATUS_ID_TRX_STATUSES_FK | **TRX_STATUSES** | TRX_STATUS_ID | TRX_STATUS_ID |

*TRANSACTIONS table has unique indexes:*

| Index Name | Field |
|---|---|
| HEXSTR_BEFORE_SIGN_UNIQUE | HEXSTR_BEFORE_SIGN |
| BLOCK_HASH_UNIQUE | BLOCK_HASH |
| TRX_ID_UNIQUE | TRX_ID |

*TRANSACTIONS table has indexes:*

| Index Name | Fields |
|---|---|
| PRIMARY | TRX_ID |
| TRX_STATUS_ID_TRX_STATUSES_IDX | TRX_STATUS_ID |

**7. TRANSACTIONS_ERROR_CODES** table is join table between TRANSACTIONS table and INTDSYSTEM_ERROR_CODES table from “shared_data” DB. Dependency with IntDS error from “shared_data” DB.

*TRANSACTIONS_ERROR_CODES table consists of below fields:*

| Field Title | DB Type | Java type | Not Null | Description |
|---|---|---|---|---|
| TRX_ID | UUID | java.util.UUID -> setObject/getObject in JDBC | true | Transaction identifier, PK from TRANSACTIONS table |
| ERR_CODE_ID | int | int | true | Error code identity number, PK from INTDSYSTEM_ERROR_CODES table, dependency with IntDS error from “shared_data” DB. Error |

| | | | | |
|---|---|---|---|---|
| | | | | with descriptions can be found in the Appendix L |
| DATE_CREATED | timestamp(6) | String | true | Date-time of record creation |

*TRANSACTIONS_ERROR_CODES table has foreign keys:*

| Foreign Key Name | Referenced Table | Field | Referenced Field |
|---|---|---|---|
| TRX_ID_TRX_ERROR_CODES_FK | **TRANSACTIONS** | TRX_ID | TRX_ID |

*TRANSACTIONS_ERROR_CODES table has indexes:*

| Index Name | Fields |
|---|---|
| PRIMARY | ERR_CODE_ID, TRX_ID |

**8. TRX_STATUSES** table holds data about transaction statuses (see statuses list in the Appendix A).

*TRX_STATUSES table consists of below fields:*

| Field Title | DB Type | Java type | Not Null | Description |
|---|---|---|---|---|
| TRX_STATUS_ID | int | int | true | Auto incremented, **Primary Key**. Transaction status identity number. |
| STATUS | varchar(45) | String | true | Transaction status name |
| STATUS_DESCR | varchar(500) | String | true | Status description |

*TRX_STATUSES table has unique indexes:*

| Index Name | Field |
|---|---|
| STATUS_UNIQUE | STATUS_NAME |

*TRX_STATUSES table has indexes:*

| Index Name | Fields |
|---|---|
| PRIMARY | TRX_STATUS_ID |

**9. WALLETS** table holds data about Btc wallets.

*WALLETS table consists of below fields:*

| Field Title | DB Type | Java type | Not Null | Description |
|---|---|---|---|---|
| WALLET_ID | UUID | java.util.UUID -> setObject/getObject in JDBC | true | **Primary Key,** the Wallet identifier. |

| | | | | |
|---|---|---|---|---|
| DATE_CREATED | timestamp(6) | String | true | Date-time of the record creation. |
| DATE_UPDATED | timestamp(6) | String | true | Date-time of the record's last update |
| BTC_AVAILABLE_FUNDS | numeric | BigDecimal | true | Credit = Sum of unspent Outputs of 'confirmed' Inbound transactions for Btc addresses of this wallet.<br>Debit = Sum of Inputs of Outbound transactions for this wallet.<br>Btc available balance = Credit - Debit. Default value ="0.0". |
| BTC_BALANCE | numeric | BigDecimal | true | Credit = Sum of all Outputs of 'confirmed' Inbound transactions for Btc addresses of this wallet.<br>Debit = Sum of Inputs of 'confirmed' Outbound transactions for this wallet.<br>Btc balance = Credit - Debit.<br>Default value ="0.0". |
| IS_SYSTEM_WALLET | bool | boolean | true | True (1) if wallet owner is IntDS otherwise false (0).<br>Default value is false (0). |
| IS_LOCKED | bool | boolean | true | True (1) if wallet is locked otherwise false (0). Default value is false (0). |
| DATE_TO_UNLOCK | timestamp(6) | String | false | Date and time when wallet will be unlocked. |

**10. WALLETS_OUTB_TRXS** table is join table between WALLETS table and TRANSACTIONS table. This table holds only Outbound transactions Identifieres.

*WALLETS_OUTB_TRXS table consists of below fields:*

| Field Title | DB Type | Java type | Not Null | Description |
|---|---|---|---|---|
| WALLET_ID | UUID | java.util.UUID -> setObject/getObject in JDBC | true | Wallet identifier, PK from WALLETS table |
| TRX_ID | UUID | java.util.UUID -> setObject/getObject in JDBC | true | Outbound transaction identifier, PK from TRANSACTIONS table |

*WALLETS_OUTB_TRXS table has foreign keys:*

| Foreign Key Name | Referenced Table | Field | Referenced Field |
|---|---|---|---|
| TRX_ID_WALLETS_TRX_FK | **TRANSACTIONS** | TRX_ID | TRX_ID |
| WALLET_ID_WALLETS_TRX_FK | **WALLETS** | WALLET_ID | WALLET_ID |

*WALLETS_OUTB_TRXS table has indexes:*

| Index Name | Fields |
|---|---|
| PRIMARY | WALLET_ID, TRX_ID |
| TRX_ID_TRANSACTIONS_FK_IDX | TRX_ID |

**11. WALLETS_INB_TRXS** table is join table between WALLETS table and TRANSACTIONS table

*WALLETS_INB_TRXS table consists of below fields:*

| Field Title | DB Type | Java type | Not Null | Description |
|---|---|---|---|---|
| WALLET_ID | UUID | java.util.UUID -> setObject/getObject in JDBC | true | Wallet identifier, PK from WALLETS table |
| TRX_ID | UUID | java.util.UUID -> setObject/getObject in JDBC | true | Inbound transaction identifier, PK from TRANSACTIONS table. This transaction is marked as Outbound in TRANSACTIONS table in case user’s change or IntDS fee. |

*WALLETS_INB_TRXS table has foreign keys:*

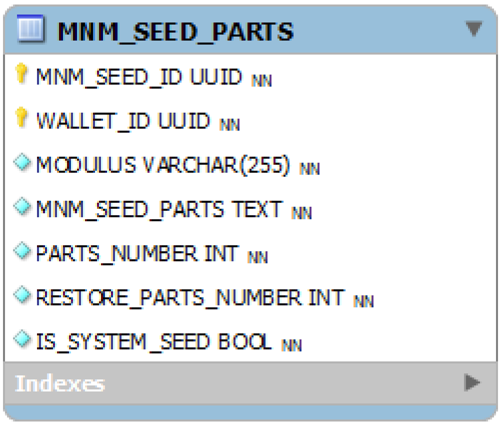
| Foreign Key Name | Referenced Table | Field | Referenced Field |
|---|---|---|---|
| TRX_ID_WALLETS_INBTRX_FK | **TRANSACTIONS** | TRX_ID | TRX_ID |
| WALLET_ID_WALLETS_INBTRX_FK | **WALLETS** | WALLET_ID | WALLET_ID |

*WALLETS_INB_TRXS table has indexes:*

| Index Name | Fields |
|---|---|
| PRIMARY | WALLET_ID, TRX_ID |
| TRX_ID_WALLETS_INBTRX_FK_IDX | TRX_ID |

### 3.1.3 Keys Management Component DB Diagram

Keys Management Component has ”**mnm_seed**” DB. The diagram below shows DB table (see Pic. 3.1.3).




Pic. 3.1.3

*Update frequency*: not changing data. *Size*: records number = number of wallets.

Mnemonic Seed can be restored by using Modulus and some parts according to Shamir's Secret Sharing Scheme [3.4]. ”**mnm_seed**” DB stores only system’s parts of mnemonic seeds.

***Note:*** *Seed will divided on the 3 parts for eWallet Web App. Seed can be restored by any 2 parts in this case.*

### 3.1.4 Keys Management Component DBs Description

**1) “mnm_seed” DB: MNM_SEED_PARTS** table holds data about system’s parts of mnemonic seed.

*MNM_SEED_PARTS table consists of below fields:*

| Field Title | DB Type | Java type | Not Null | Description |
|---|---|---|---|---|
| MNM_SEED_ID | UUID | java.util.UUID -> setObject/getObject in JDBC | true | **Primary key**. Mnemonic seed identifier. |
| WALLET_ID | UUID | java.util.UUID -> setObject/getObject in JDBC | true | **Primary key**. Wallet identifier, PK from WALLETS table in “trx_management” DB |
| MODULUS | varchar(255) | String | true | Rundomly generated Modulus value for restoring of private key from any “N= RESTORE_PARTS_NUMBER” parts according to 4S scheme [3.4]. |

| | | | | |
|---|---|---|---|---|
| MNM_SEED_PARTS | text | String | true | System parts of Mnemonic seed with indexes which are started from 2. Number of parts is K -1 where K= RESTORE_PARTS_NUMBER The format is [2]_[part2 string]_[3]_[part3 string] … [K]_[partK string]. Part with index=1 should be given to user and should not be kept in the table excluding IntDS system seed. “Underscore” is separator between parts. |
| PARTS_NUMBER | int | int | true | Total number of parts. Default value is 3. |
| RESTORE_PARTS_NUMBER | int | int | true | Necessary number of parts which can restore Mnemonic seed. RESTORE_PARTS_NUMBER <= PARTS_NUMBER Default value is 2. |
| IS_SYSTEM_SEED | bool | boolean | true | True (1) if seed owner is IntDS otherwise false (0). Default value is false (0). |

*MNM_SEED_PARTS table has unique indexes:*

| Index Name | Field |
|---|---|
| MODULUS_UNIQUE | MODULUS |

*MNM_SEED_PARTS table has indexes:*

| Index Name | Fields |
|---|---|
| PRIMARY | MNM_SEED_ID, WALLET_ID |

## 3.2 Accounting Transaction Management SubSystem DBs

This point can be done in the scope of future development. Will need some researching activity.

## 3.3 Bank Transaction Management SubSystem DBs

This point can be done in the scope of future development. Will need some researching activity.

## 3.4 Exchange Transaction Management SubSystem DBs

This point can be done in the scope of future development. Will need some researching activity.

## 3.5 Message Transaction Management SubSystem DBs

This point can be done in the scope of future development. Will need some researching activity.

## 3.6 Contracts Management SubSystem DBs

This point can be done in the scope of future development. Will need some researching activity.

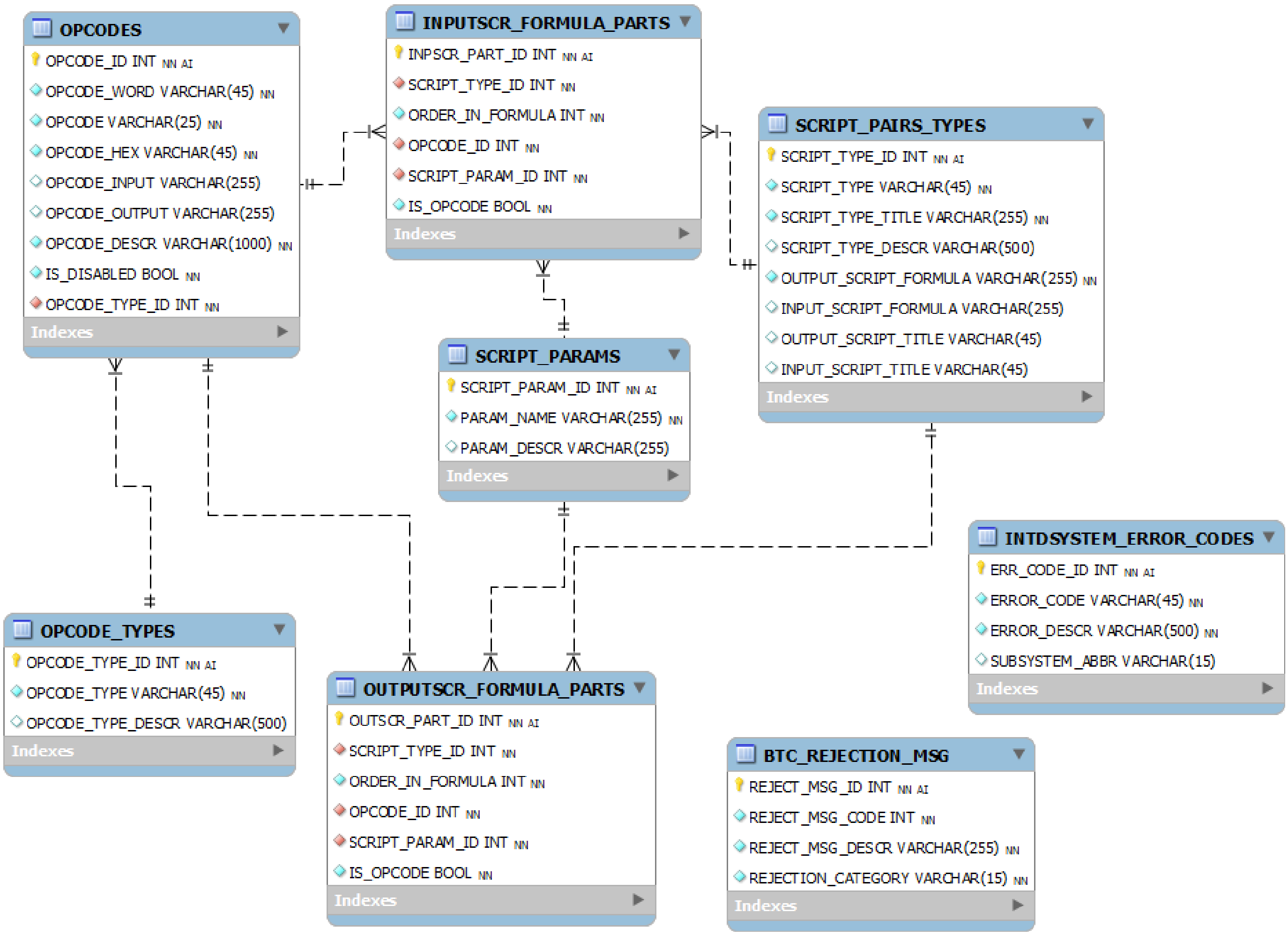
# 3.7 Shared DBs

Shared DBs stores Data which can be used by each iDaemon sub system and component.

## 3.7.1 IntDS Shared Data DB Diagram

"IntDS Shared Data" DB (**"shared_data")** consists of 8 tables and stores data related to IntDS errors and types of Locking and Unlocking scripts. Every Inputs must have Unlocking Script (`scriptSig`) and every Outputs must have Locking Script (`ScriptPubKey`). Both scripts are using formula with op-codes depending on transaction type (*see Paragraph 8* for more details).

The diagram below (Pic. 3.7.1) shows the DB structure.




Pic. 3.7.1

## 3.8 Monitoring System DB

### 3.8.1 IntDS Shared Data DB Description

**1. BTC_REJECTION_MSG** table holds data of rejection messages from blockchain. Data is described in *Appendix M*.

*BTC_REJECTION_MSG table consists of below fields:*

| Field Title | DB Type | Java type | Not Null | Description |
|---|---|---|---|---|
| REJECT_MSG_ID | int | int | true | Auto incremented, **Primary Key**. Rejection message identity number. |
| REJECT_MSG_CODE | varchar(45) | String | true | Rejection message code |
| REJECT_MSG_DESCR | varchar(500) | String | true | Rejection message description |
| REJECTION_CATEGORY | varchar(15) | String | false | Rejection message category |

*BTC_REJECTION_MSG table has indexes:*

| Index Name | Fields |
|---|---|
| PRIMARY | REJECT_MSG_ID |

**2. INTDSYSTEM_ERROR_CODES** table holds data of IntDS error codes. Data is described in *Appendix L*.

*INTDSYSTEM_ERROR_CODES table consists of below fields:*

| Field Title | DB Type | Java type | Not Null | Description |
|---|---|---|---|---|
| ERR_CODE_ID | int | int | true | Auto incremented, **Primary Key**. Error code identity number. |
| ERROR_CODE | varchar(45) | String | true | IntDS error code |
| ERROR_DESCR | varchar(500) | String | true | Error description |
| SUBSYSTEM_ABBR | varchar(15) | String | false | SubSystem Abbreviation (see start of document “Acronyms and Abbreviations of the Current Document”). Null in case IntDS common error. |

*INTDSYSTEM_ERROR_CODES table has unique indexes:*

| Index Name | Field |
|---|---|
| ERROR_CODE_UNIQUE | ERROR_CODE |

*INTDSYSTEM_ERROR_CODES table has indexes:*

| Index Name | Fields |
|---|---|
| PRIMARY | ERR_CODE_ID |

**3. INPUTSCR_FORMULA_PARTS** table holds data about parts of scriptSig formula for Inputs. (Appendix E)

*INPUTSCR_FORMULA_PARTS table consists of below fields:*

| Field Title | DB Type | Java type | Not Null | Description |
|---|---|---|---|---|
| INPSCR_PART_ID | int | int | true | Auto incremented, **Primary Key**. Identity number of Input script part. |
| SCRIPT_TYPE_ID | int | int | true | Script type identity, PK from SCRIPT_PAIRS_TYPES table, dependency with script pair type. The record is related to part of INPUT_SCRIPTSIG_FORMULA value. |
| ORDER_IN_FORMULA | int | int | true | Order number of the part in the script formula string. |
| OPCODE_ID | int | int | true | Opcode identity, PK from OPCODES table, dependency with opcode. Value is Zero if this part is not opcode. This part is script parameter in this case. |
| SCRIPT_PARAM_ID | int | int | true | Script parameter identity, PK from SCRIPT_PARAMS table, dependency with script parameter. Value is Zero if this part is not script parameter. This part is opcode in this case. |
| IS_OPCODE | bool | boolean | true | This field is true (1) if part is opcode, otherwise false (0). Default value is true (1). |

*INPUTSCR_FORMULA_PARTS table has foreign keys:*

| Foreign Key Name | Referenced Table | Field | Referenced Field |
|---|---|---|---|
| INP_FORMULA_SCRIPT_TYPE_ID_FK | **SCRIPT_PAIRS_TYPES** | SCRIPT_TYPE_ID | SCRIPT_TYPE_ID |
| INP_FORMULA_OPCODE_ID_FK | **OPCODES** | OPCODE_ID | OPCODE_ID |
| INP_FORMULA_SCRIPT_PARAMID_FK | **SCRIPT_PARAMS** | SCRIPT_PARAM_ID | SCRIPT_PARAM_ID |

*INPUTSCR_FORMULA_PARTS table has indexes:*

| Index Name | Fields |
|---|---|
| PRIMARY | INPSCR_PART_ID |
| INP_FORMULA_SCRIPT_TYPE_ID_FK_IDX | SCRIPT_TYPE_ID |

| | |
|---|---|
| INP_FORMULA_OPCODE_ID_FK_IDX | OPCODE_ID |
| INP_FORMULA_SCRIPT_PARAMID_FK_IDX | SCRIPT_PARAM_ID |

**4. OPCODES** table holds data about operation codes (see opcodes data in the Appendix D) which are used in the Inputs/Outputs scripts.

*OPCODES table consists of below fields:*

| Field Title | DB Type | Java type | Not Null | Description |
|---|---|---|---|---|
| OPCODE_ID | int | int | true | Auto incremented, **Primary Key**. Opcode identity number. |
| OPCODE_WORD | varchar(45) | String | true | Opcode as word representation. Unique. |
| OPCODE | varchar(25) | String | true | Opcode as numbers representation |
| OPCODE_HEX | varchar(45) | String | true | Opcode as hex representation |
| OPCODE_INPUT | varchar(255) | String | false | Opcode in the input script |
| OPCODE_OUTPUT | varchar(255) | String | false | Opcode in the output script |
| OPCODE_DESCR | varchar(1000) | String | true | Opcode description |
| IS_DISABLED | bool | boolean | true | This field is true (1) if opcode marked as disabled, otherwise false (0). Default value is false (0). If any opcode marked as disabled is present in a script - it must also abort and fail. |
| OPCODE_TYPE_ID | int | int | true | Opcode type identity, PK from OPCODE_TYPES table, dependency with opcode type. |

*OPCODES table has foreign keys:*

| Foreign Key Name | Referenced Table | Field | Referenced Field |
|---|---|---|---|
| OPCODE_TYPE_ID_OPCODE_TYPE_FK | **OPCODE_TYPES** | OPCODE_TYPE_ID | OPCODE_TYPE_ID |

*OPCODES table has unique indexes:*

| Index Name | Field |
|---|---|
| OPCODE_WORD_UNIQUE | OPCODE_WORD |

*OPCODES table has indexes:*

| Index Name | Fields |
|---|---|
| PRIMARY | OPCODE_ID |

| | |
|---|---|
| OPCODE_TYPE_ID_OPCODE_TYPE_IDX | OPCODE_TYPE_ID |

**5. OPCODE_TYPES** table holds data about opcode types (see opcode types in the Appendix C).

*OPCODE_TYPES table consists of below fields:*

| Field Title | DB Type | Java type | Not Null | Description |
|---|---|---|---|---|
| OPCODE_TYPE_ID | int | int | true | Auto incremented, **Primary Key**. Opcode type identity number. |
| OPCODE_TYPE | varchar(45) | String | true | Opcode type name |
| OPCODE_TYPE_DESCR | varchar(500) | String | false | Opcode type description |

OPCODE_*TYPES table has indexes:*

| Index Name | Fields |
|---|---|
| PRIMARY | OPCODE_TYPE_ID |

**6. OUTPUTSCR_FORMULA_PARTS** table holds data about parts of scriptPubKey formula for Outputs. (Appendix E)

*OUTPUTSCR_FORMULA_PARTS table consists of below fields:*

| Field Title | DB Type | Java type | Not Null | Description |
|---|---|---|---|---|
| OUTSCR_PART_ID | int | int | true | Auto incremented, **Primary Key**. Identity number of Output script part. |
| SCRIPT_TYPE_ID | int | int | true | Script type identity, PK from SCRIPT_PAIRS_TYPES table, dependency with script pair type. The record is related to part of OUTPUT_SCRIPTPUBKEY_FORMULA value. |
| ORDER_IN_FORMULA | int | int | true | Order number of the part in the script formula string. |
| OPCODE_ID | int | int | true | Opcode identity, PK from OPCODES table, dependency with opcode. Value is Zero if this part is not opcode. This part is script parameter in this case. |
| SCRIPT_PARAM_ID | int | int | true | Script parameter identity, PK from SCRIPT_PARAMS table, dependency with script parameter. Value is Zero |

| | | | | |
|---|---|---|---|---|
| | | | | if this part is not script parameter. This part is opcode in this case. |
| IS_OPCODE | bool | boolean | true | This field is true (1) if part is opcode, otherwise false (0). Default value is true (1). |

*OUTPUTSCR_FORMULA_PARTS table has foreign keys:*

| Foreign Key Name | Referenced Table | Field | Referenced Field |
|---|---|---|---|
| OUTP_FORMULA_SCRIPT_TYPE_ID_FK | **SCRIPT_PAIRS_TYPES** | SCRIPT_TYPE_ID | SCRIPT_TYPE_ID |
| OUTP_FORMULA_OPCODE_ID_FK | **OPCODES** | OPCODE_ID | OPCODE_ID |
| OUTP_FORMULA_SCRIPT_PARAMID_FK | **SCRIPT_PARAMS** | SCRIPT_PARAM_ID | SCRIPT_PARAM_ID |

*OUTPUTSCR_FORMULA_PARTS table has indexes:*

| Index Name | Fields |
|---|---|
| PRIMARY | OUTSCR_PART_ID |
| OUTP_FORMUL_SCRIPT_TYPE_ID_IDX | SCRIPT_TYPE_ID |
| OUTP_FORMULA_OPCODE_ID_FK_IDX | OPCODE_ID |
| OUTP_FORMULA_SCRIPT_PARAMID_FK_IDX | SCRIPT_PARAM_ID |

**7. SCRIPT_PARAMS** table holds data about script parameters (see parameters data in the Appendix F)

*SCRIPT_PARAMS table consists of below fields:*

| Field Title | DB Type | Java type | Not Null | Description |
|---|---|---|---|---|
| SCRIPT_PARAM_ID | int | int | true | Auto incremented, **Primary Key**. Script parameter identity number. |
| PARAM_NAME | varchar(255) | String | true | Parameter name |
| PARAM_DESCR | varchar(255) | String | false | Parameter description |

*SCRIPT_PARAMS table has unique indexes:*

| Index Name | Field |
|---|---|
| PARAM_NAME_UNIQUE | PARAM_NAME |

*SCRIPT_PARAMS table has indexes:*

| Index Name | Fields |
|---|---|
| PRIMARY | SCRIPT_PARAM_ID |

**8. SCRIPT_PAIRS_TYPES** table holds data about types of script pairs in the Output and Input parts of transaction (see types data in the Appendix E)

*SCRIPT_PAIRS_TYPES table consists of below fields:*

| Field Title | DB Type | Java type | Not Null | Description |
|---|---|---|---|---|
| SCRIPT_TYPE_ID | int | int | true | Auto incremented, **Primary Key**. Identity number of types of script pairs. |
| SCRIPT_TYPE | varchar(45) | String | true | Script type. Unique |
| SCRIPT_TYPE_TITLE | varchar(255) | String | true | Script type title. Unique |
| SCRIPT_TYPE_DESCR | varchar(500) | String | false | Script type description |
| OUTPUT_SCRIPT_FORMULA | varchar(255) | String | true | Output's Script formula. Unique |
| INPUT_SCRIPT_FORMULA | varchar(255) | String | false | Input's Script formula |
| OUTPUT_SCRIPT_TITLE | varchar(45) | String | false | Output's Script title. Example: scriptPubKey |
| INPUT_SCRIPT_TITLE | varchar(45) | String | false | Output's Script title. Example: scriptSig |

*SCRIPT_PAIRS_TYPES table has unique indexes:*

| Index Name | Field |
|---|---|
| SCRIPT_TYPE_UNIQUE | SCRIPT_TYPE |
| SCRIPT_TYPE_TITLE_UNIQUE | SCRIPT_TYPE_TITLE |
| OUTPUT_SEND_FORMULA_UNIQUE | OUTPUT_SCRIPTPUBKEY_FORMULA |

*SCRIPT_PAIRS_TYPES table has indexes:*

| Index Name | Fields |
|---|---|
| PRIMARY | SCRIPT_TYPE_ID |

### 3.8.2 Monitoring System DB Diagram

The Monitoring System has **“trx_monitoring”** DB. This DB consists of Block Chain data and sub-systems data (as new transactions, btc addresses, etc.) which should be monitored. The name prefix of sub-system tables should correspond to sub system name:

- two tables with “**STRXMSS_**” name prefix stores data from “*Single-sig Trx Management SubSystem*”

- tables with “**ATRXMSS_**” name prefix stores data from “*Accounting Trx Management SubSystem*”

- tables with “**BTRXMSS_**” name prefix stores data from “*Bank Trx Management SubSystem*”

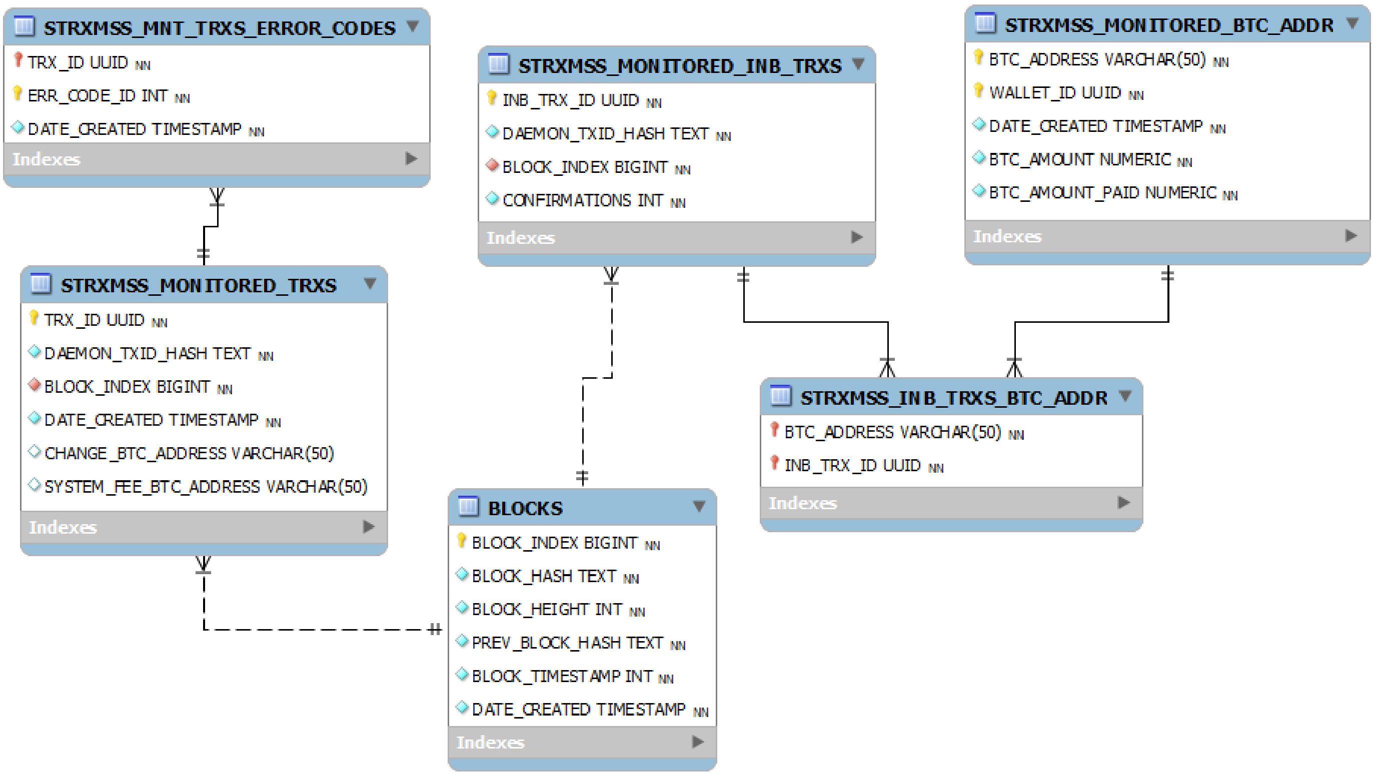
- tables with “**EATRXMSS_**” name prefix stores data from “*Exchange Trx Management SubSystem*”

- tables with “**MTRXMSS_**” name prefix stores data from “*Message Trx Management SubSystem*”

- tables with “**CONTRMSS_**” name prefix stores data from “*Contracts Management SubSystem*”

***Notes:** There are only STrxMSS tables at this moment. “trx_monitoring” DB structure should be updated in future versions of this document depending on planning stage of each sub-system.*

The diagram below (Pic. 3.8.1) shows the Monitoring System DB structure.




Pic. 3.8.1

### 3.8.3 Monitoring System DB Description

**BLOCKS** table holds data about blocks. The table is used during the monitoring iteration to determine the latest block that has been changed since last iteration. Blocks data is received and captured periodically from Block Chain.

*BLOCKS table consists of below fields:*

| Field Title | DB Type | Java type | Not Null | Description |
|---|---|---|---|---|
| BLOCK_INDEX | int | int | true | Auto incremented, **Primary Key**. Block identity number. |
| BLOCK_HASH | text | String | true | Block Identifier is a hash of block. Maximum size - 2,000,000 chars. Unique |
| BLOCK_HEIGHT | int | int | true | The block height or index. The first block is Genesis. Genesis height is zero. |
| PREV_BLOCK_HASH | text | String | true | The hash of the previous block |
| BLOCK_TIMESTAMP | int | int | true | The Date-time when the block was created |
| DATE_CREATED | timestamp(6) | String | true | Date-time of record creation |

*BLOCKS table has unique indexes:*

| Index Name | Field |
|---|---|
| BLOCK_HASH_UNIQUE | BLOCK_HASH |

*BLOCKS table has indexes:*

| Index Name | Fields |
|---|---|
| PRIMARY | BLOCK_INDEX |

**Single-sig Transaction Monitoring SubSystem tables:**

**1. STRXMSS_MNT_TRXS_ERROR_CODES** table is join table between STRXMSS_MONITORED_TRXS table and INTDSYSTEM_ERROR_CODES table from "shared_data" DB. Dependency with IntDS error from "shared_data" DB.

*TRANSACTIONS_ERROR_CODES table consists of below fields:*

| Field Title | DB Type | Java type | Not Null | Description |
|---|---|---|---|---|

| | | | | |
|---|---|---|---|---|
| TRX_ID | UUID | java.util.UUID -> setObject/getObject in JDBC | true | Transaction identifier, PK from STRXMSS_MONITORED_TRXS table |
| ERR_CODE_ID | int | int | true | Error code identity number, PK from INTDSYSTEM_ERROR_CODES table, dependency with IntDS error from “shared_data” DB. Error with descriptions can be found in the Appendix L |
| DATE_CREATED | timestamp(6) | String | true | Date-time of record creation |

*STRXMSS_MNT_TRXS_ERROR_CODES table has foreign keys:*

| Foreign Key Name | Referenced Table | Field | Referenced Field |
|---|---|---|---|
| TRX_ID_ERROR_CODES_FK | **STRXMSS_MONITORED_TRXS** | TRX_ID | TRX_ID |

*STRXMSS_MNT_TRXS_ERROR_CODES table has indexes:*

| Index Name | Fields |
|---|---|
| PRIMARY | ERR_CODE_ID, TRX_ID |

**2. STRXMSS_MONITORED_BTC_ADDR** table holds data about Btc addresses which should be monitored for inbound transactions.

*STRXMSS_MONITORED_BTC_ADDR table consists of below fields:*

| Field Title | DB Type | Java type | Not Null | Description |
|---|---|---|---|---|
| BTC_ADDRESS | varchar(50) | String | true | **Primary Key,** the Bitcoin address identifier. |
| WALLET_ID | UUID | java.util.UUID -> setObject/getObject in JDBC | true | **Primary Key,** Wallet identifier. PK from WALLETS table, “trx_management” DB. Dependency with Wallet record in “trx_management” DB |
| DATE_CREATED | timestamp(6) | String | true | Date-time of record creation |
| BTC_AMOUNT | numeric | BigDecimal | true | Bitcoins amount which should be paid. |
| BTC_AMOUNT_PAID | numeric | BigDecimal | true | Bitcoins amount which already was paid. |

*STRXMSS_MONITORED_BTC_ADDR table has indexes:*

| Index Name | Fields |
|---|---|
| PRIMARY | BTC_ADDRESS, WALLET_ID |

**3. STRXMSS_MONITORED_INB_TRXS** table holds data about Inbound Btc transactions which should be monitored for confirmations.

*STRXMSS_MONITORED_INB_TRXS table consists of below fields:*

| Field Title | DB Type | Java type | Not Null | Description |
|---|---|---|---|---|
| INB_TRX_ID | UUID | java.util.UUID -> setObject/getObject in JDBC | true | **Primary Key,** the transaction identifier. |
| DAEMON_TXID_HASH | text | String | true | Transaction Identifier is a hash of completed transaction which allows other transactions to spend its outputs. Transaction Identifier is received from Block Chain via the DmnCS. Maximum size - 1,000,000 chars. Unique |
| BLOCK_INDEX | int | int | true | Block identifier, PK from BLOCKS table, dependency with block in which transaction was included. |
| CONFIRMATIONS | int | int | true | Number of new blocks in Block Chain after the transaction has been included in the block and block was published to the network. Confirmations is received from Block Chain via the DmnCS. The transaction should be considered as **confirmed** if it is a six number of blocks deep. Zero by default. |

*STRXMSS_MONITORED_INB_TRXS table has indexes:*

| Index Name | Fields |
|---|---|
| PRIMARY | DAEMON_TXID_HASH |
| BLOCK_INDEX_FK_IDX | BLOCK_INDEX |

*TRANSACTIONS table has foreign keys:*

| Foreign Key Name | Referenced Table | Field | Referenced Field |
|---|---|---|---|
| BLOCK_INDEX_FK | **BLOCKS** | BLOCK_INDEX | BLOCK_INDEX |

**4. STRXMSS_INB_TRXS_BTC_ADDR** table is join table between STRXMSS_MONITORED_BTC_ADDR table and STRXMSS_MONITORED_INB_TRXS table.

| Field Title | DB Type | Java type | Not Null | Description |
|---|---|---|---|---|
| BTC_ADDRESS | varchar(50) | String | true | Btc address identity, PK from STRXMSS_MONITORED_BTC_ADDR table |
| INB_TRX_ID | UUID | java.util.UUID -> setObject/getObject in JDBC | true | Inbound Trx identifier, PK from STRXMSS_MONITORED_INB_TRXS table |

*STRXMSS_INB_TRXS_BTC_ADDR table has indexes:*

| Index Name | Fields |
|---|---|
| PRIMARY | BTC_ADDRESS, INB_TRX_ID |
| TXID_HASH_INBTRX_BTCADDR_FK_IDX | INB_TRX_ID |

*STRXMSS_INB_TRXS_BTC_ADDR table has foreign keys:*

| Foreign Key Name | Referenced Table | Field | Referenced Field |
|---|---|---|---|
| BTC_ADDR_INBTRX_BTCADDR_FK | **STRXMSS_MONITORED_BTC_ADDR** | BTC_ADDRESS | BTC_ADDRESS |
| INB_TRXID_INBTRX_BTCADDR_FK | **STRXMSS_MONITORED_INB_TRXS** | INB_TRX_ID | INB_TRX_ID |

**5. STRXMSS_MONITORED_TRXS** table holds data about Outbound Btc transactions which should be monitored for confirmations.

*STRXMSS_MONITORED_TRXS table consists of below fields:*

| Field Title | DB Type | Java type | Not Null | Description |
|---|---|---|---|---|
| TRX_ID | UUID | java.util.UUID -> setObject/getObject in JDBC | true | **Primary Key,** the transaction identifier. |
| DAEMON_TXID_HASH | text | String | true | Transaction Identifier is a hash of completed transaction which allows other transactions to spend its outputs. Transaction Identifier is received from Block |

| | | | | |
|---|---|---|---|---|
| | | | | Chain via the DmnCS. Zero by default. Maximum size - 1,000,000 chars. Unique |
| BLOCK_INDEX | int | int | true | Block identifier, PK from BLOCKS table, dependency with block in which transaction was included |
| DATE_CREATED | timestamp(6) | String | true | Date-time of record creation |
| CHANGE_BTC_ADDRESS | varchar(50) | String | false | User’s Btc address for receiving change. This field is not null if this Outbound trx has a change which should be returned to User, otherwise null. Null by default. |
| SYSTEM_FEE_BTC_ADDRESS | varchar(50) | String | false | Company Btc address for receiving trx fee as IntDS profit. This field is not null if this Outbound trx has IntDS fee which should be returned to the Company Wallet, otherwise null. Null by default. |

*STRXMSS_MONITORED_TRXS table has foreign keys:*

| Foreign Key Name | Referenced Table | Field | Referenced Field |
|---|---|---|---|
| BLOCK_INDEX_BLOCKS_FK | **BLOCKS** | BLOCK_INDEX | BLOCK_INDEX |

*STRXMSS_MONITORED_TRXS table has unique indexes:*

| Index Name | Field |
|---|---|
| DAEMON_TXID_ HASH_UNIQUE | DAEMON_TXID_ HASH |

*STRXMSS_MONITORED_TRXS table has indexes:*

| Index Name | Fields |
|---|---|
| PRIMARY | TRX_ID |
| BLOCK_INDEX_BLOCKS_FK_IDX | BLOCK_INDEX |

## 3.9 Functions and Stored Procedures Specifications

By default, PostgreSQL supports 3 procedural languages: SQL, PL/pgSQL, and C. PL/pgSQL [1.20] is SQL Procedural Language.

The advantages of using PostgreSQL stored procedures are [1.21]:

1. Reduce the number of round trips between application and database servers. All SQL statements are wrapped inside a function stored in the PostgreSQL database server so the application only has to issue a function call to get the result back instead of sending multiple SQL statements and wait for the result between each call.
2. Increase application performance because user-defined functions pre-compiled and stored in the PostgreSQL database server.
3. Be able to reuse in many applications. Once you develop a function, you can reuse it in any applications.

Note: Disadvantages as manage versions, conversion from PostgreSQL to other DB types, etc. are not considered in the scope of this project

### 3.9.1 STrxMSS Functions and Stored Procedures

#### Add Input data from UTXO

| PSQL SP Name | Description | Example to call | Result example |
|---|---|---|---|
| add_input_from_utxo | Procedure add Input data record from given UTXO for new Outbound transaction.<br>1) INSERT INTO INPUTS (…) VALUES (…)<br>Fields-Values:<br>INPUT_ID<br>TRX_ID=0<br>TEMP_TRX_ID=TEMP_OUTB_TRXS.TEMP_TRX_ID<br>INPUT_INDEX=<ind><br>PREV_DAEMON_TXID_HASH=<TRANSACTIONS.DAEMON_TXID_HASH of current UTXO><br>VOUT_INDEX=<OUTPUTS.OUTPUT_INDEX of current UTXO><br>VOUT_BTC_VALUE=<OUTPUTS.BTC_VALUE of current UTXO><br>SCRIPT_SIG_HASH=0<br>SCRIPT_TYPE_ID=1<br><br>2) Add dependency Input -> previous Trx Output<br>UPDATE OUTPUTS SET SPENT_BY_INPUT_ID=INPUTS,INPUT_ID WHERE OUTPUT_ID=<OUTPUTS.OUTPUT_ID of current UTXO> | | |

**Input Parameters:**

| Parameter title | PSQL Type | Java Type | Value Example | Description |
|---|---|---|---|---|
| | | | | |

**Output Parameters:**

| Parameter title | PSQL Type | Java Type | Value Example | Description |
|---|---|---|---|---|
| | | | | |

**Exceptions:**

| PSLQ Error code | PSQL Condition Name | Error Description | Error Returned Value | Error Returned Value Java Type |
|---|---|---|---|---|
| | | | | |

### Add Inbound Trx data from message

| PSQL SP Name | Description | Example to call | Result example |
|---|---|---|---|
| add_inb_trx_from_mnts | Procedure creates record in the TRANSACTIONS table for Inbound Trx.<br>TrxMC DB is “trx_management”.<br>1) INSERT INTO TRANSACTIONS (…) VALUES (…)<br>Fields-Values criteria:<br>DAEMON_TXID_HASH<br>TRX_RAW_SGN_DATA<br>TRX_RAW_SGN_NOSERIAL<br>DATE_CREATED=<now><br>DATE_UPDATED=DATE_CREATED<br>CONFIRMATIONS=6<br>BLOCK_HASH<br>IS_OUTBOUND=false<br>LOCK_TIME=0<br>IS_EVERY_OUTPUT_SPENT=false<br>TRX_STATUS_ID=1<br>HAS_USER_CHANGE=false<br>HAS_SYSTEM_FEE=false<br>IS_REJECTED=false<br>REJECT_MSG_ID=null<br>MINER_FEE<br><br>2) INSERT INTO WALLETS_INB_TRXS (…) VALUES (…)<br>TRX_ID=<new TRANSACTIONS.TRX_ID><br>WALLET_ID=<given Wallet Id><br><br>3) INSERT INTO OUTPUTS (…) VALUES (…)<br>Fields-Values criteria:<br>TRX_ID=<new TRANSACTIONS.TRX_ID><br>OUTPUT_INDEX<br>BTC_VALUE<br>BTC_ADDRESS<br>IS_SPENT=false | | |

| | | | |
|---|---|---|---|
| | IS_SYSTEM_FEE=false<br>IS_CHANGE=false<br>SCRIPT_PUB_KEY_HASH<br>PUBK_SCRIPT_TYPE_ID=1<br><br>6) INSERT INTO INPUTS (…) VALUES (…)<br>Fields-Values criteria:<br>TRX_ID=<new TRANSACTIONS.TRX_ID><br>INPUT_INDEX<br>PREV_DAEMON_TXID_HASH<br>VOUT_INDEX<br>VOUT_BTC_VALUE<br>SCRIPT_SIG_HASH<br>SCRIPT_TYPE_ID=1 | | |

**Input Parameters:**

| Parameter title | PSQL Type | Java Type | Value Example | Description |
|---|---|---|---|---|
| | | | | |

**Output Parameters:**

| Parameter title | PSQL Type | Java Type | Value Example | Description |
|---|---|---|---|---|
| | | | | |

**Exceptions:**

| PSLQ Error code | PSQL Condition Name | Error Description | Error Returned Value | Error Returned Value Java Type |
|---|---|---|---|---|
| | | | | |

## Add Outbound Trx data from temporary Trx data

| PSQL SP Name | Description | Example to call | Result example |
|---|---|---|---|
| add_outb_trx_from_tmp | Procedure creates record in the TRANSACTIONS table for Outbound Trx.<br>TrxMC DB is “trx_management”.<br>1) INSERT INTO TRANSACTIONS (…) VALUES (…)<br>Fields-Values:<br>DAEMON_TXID_HASH=0<br>TRX_RAW_SGN_DATA=TEMP_OUTB_TRXS.TRX_RAW_SGN_DATA<br>TRX_RAW_SGN_NOSERIAL=<br>TEMP_OUTB_TRXS.TRX_RAW_SGN_NOSERIAL<br>DATE_CREATED=<now><br>DATE_UPDATED=DATE_CREATED<br>CONFIRMATIONS=0<br>BLOCK_HASH=null<br>IS_OUTBOUND=true<br>LOCK_TIME=0<br>IS_EVERY_OUTPUT_SPENT=false<br>TRX_STATUS_ID=3<br>HAS_USER_CHANGE=TEMP_OUTB_TRXS.HAS_USER_CHANGE | | |

| | | | |
|---|---|---|---|
| | HAS_SYSTEM_FEE=<true if TEMP_OUTB_TRXS.INTDS_FEE >0 otherwise false><br>IS_REJECTED=false<br>REJECT_MSG_ID=null<br>MINER_FEE= TEMP_OUTB_TRXS.PRIORITY_FEE<br><br>2) INSERT INTO WALLETS_OUTB_TRXS (…) VALUES (…)<br>TRX_ID=<new TRANSACTIONS.TRX_ID><br>WALLET_ID=TEMP_OUTB_TRXS.FROM_WALLET_ID<br><br>3)If TRANSACTIONS.HAS_USER_CHANGE=true INSERT INTO WALLETS_INB_TRXS (…) VALUES (…)<br>TRX_ID=<new TRANSACTIONS.TRX_ID><br>WALLET_ID=TEMP_OUTB_TRXS.FROM_WALLET_ID<br><br>4) If TRANSACTIONS.HAS_SYSTEM_FEE=true INSERT INTO WALLETS_INB_TRXS (…) VALUES (…)<br>TRX_ID=<new TRANSACTIONS.TRX_ID><br>WALLET_ID=<WALLETS.WALLET_ID<br>wich can be received according to criteria:<br>OUTPUTS.TRX_ID= new TRANSACTIONS.TRX_ID<br>OUTPUTS.BTC_ADDRESS=SYSTEM_BTC_ADDRESS.BTC_ADDRESS<br>SYSTEM_BTC_ADDRESS.WALLET_ID= WALLETS.WALLET_ID<br>WALLETS.IS_SYSTEM_WALLET=true><br><br>5) UPDATE OUTPUTS SET TRX_ID=<new TRANSACTIONS.TRX_ID> WHERE TEMP_TRX_ID=TEMP_OUTB_TRXS.TEMP_TRX_ID<br>6) UPDATE INPUTS SET TRX_ID=<new TRANSACTIONS.TRX_ID> WHERE TEMP_TRX_ID=TEMP_OUTB_TRXS.TEMP_TRX_ID | | |

**Input Parameters:**

| Parameter title | PSQL Type | Java Type | Value Example | Description |
|---|---|---|---|---|
| | | | | |

**Output Parameters:**

| Parameter title | PSQL Type | Java Type | Value Example | Description |
|---|---|---|---|---|
| | | | | |

**Exceptions:**

| PSLQ Error code | PSQL Condition Name | Error Description | Error Returned Value | Error Returned Value Java Type |
|---|---|---|---|---|
| | | | | |

### Add Wallet Stored Procedure

| PSQL Func Name | Description | Example to call | Result example |
|---|---|---|---|
| add_wallet | Procedure adds new wallet.<br>Add record to WALLETS table:<br>WALLET_ID = SELECT <postgresql uuid generation function><br>DATE_CREATED=<now><br>DATE_UPDATED=<now><br>BTC_AVAILABLE _FUNDS=0<br>BTC_BALANCE=0<br>IS_SYSTEM_WALLET=<isSystemWallet><br>IS_LOCKED=false<br>DATE_TO_UNLOCK=NULL | | |

**Input Parameters:**

| Parameter title | PSQL Type | Java Type | Value Example | Description |
|---|---|---|---|---|
| isSystemWallet | | | | |
| | | | | |

**Output Parameters:**

| Parameter title | PSQL Type | Java Type | Value Example | Description |
|---|---|---|---|---|
| Wallet_ID | | | | Wallet ID of the newly created wallet. |
| isSystemWallet | | | | |

Exceptions:

| PSLQ Error code | PSQL Condition Name | Error Description | Error Returned Value | Error Returned Value Java Type |
|---|---|---|---|---|
| | | | | |

### Add Mnemonic seed parts Function

| PSQL Func Name | Description | Example to call | Result example |
|---|---|---|---|
| add_mnm_parts | Procedure adds mnemonic seed parts for a wallet.<br>Add record to MNM_SEED_PARTS table:<br><br>1) Values:<br>WALLET_ID = <wallet id><br>MNM_SEED_ID = SELECT <postgresql uuid generation function><br>MODULUS = <modulus> | | |

| | | | |
|---|---|---|---|
| | PARTS_NUMBER = <n><br>RESTORE_PARTS_NUMBER = <k><br>IS_SYSTEM_SEED = <isSystemWallet><br><br>2) Parse JSON array partsArray and assign value to MNM_SEED_PARTS field. Refer section 3.1.4 for format of the string.<br>If isSystemWallet = true:<br>Form string (mnm_parts) with all k parts of the partsArray.<br><br>Else<br>Form string (mnm_parts) with k-1 parts starting at index 2.<br>Return arrValue of index 1.<br><br>MNM_SEED_PARTS = mnm_parts | | |

**Input Parameters:**

| Parameter title | PSQL Type | Java Type | Value Example | Description |
|---|---|---|---|---|
| Wallet_Id | | | | Wallet_Id returned by add_wallet SP |
| isSystemWallet | | | | isSystemWallet as returned by add_wallet SP |
| modulus | | | | Modulus returned by 4S API |
| partsArray | | | | String with mnemonic seed parts. Refer section 3.1.4 for format. |
| n | | | 3 | N as returned by 4S API |
| k | | | 2 | K as returned by 4S API |

**Output Parameters:**

| Parameter title | PSQL Type | Java Type | Value Example | Description |
|---|---|---|---|---|
| mnmUserPart | (Optional) | | | User part of the mnemonic seed. |

Exceptions:

| PSLQ Error code | PSQL Condition Name | Error Description | Error Returned Value | Error Returned Value Java Type |
|---|---|---|---|---|
| | | | | |

### Calculate total utxo balance for a wallet Stored Procedure

| PSQL SP Name | Description | Example to call | Result example |
|---|---|---|---|
| utxos_balance_for_wallet | Procedure calculates current UTXOs balance for wallet.<br>1) Get a list of unspent outputs (UTXOs): Call "utxos_for_wallet(…)"<br><br>2) Sum of UTXOs Outputs Btc values is the required balance:<br>Sum of OUTPUTS.BTC_VALUE | | |

**Input Parameters:**

| Parameter title | PSQL Type | Java Type | Value Example | Description |
|---|---|---|---|---|
| | | | | |

**Output Parameters:**

| Parameter title | PSQL Type | Java Type | Value Example | Description |
|---|---|---|---|---|
| | | | | |

Exceptions:

| PSLQ Error code | PSQL Condition Name | Error Description | Error Returned Value | Error Returned Value Java Type |
|---|---|---|---|---|
| | | | | |

### Calculate total amount spent by confirmed transactions in a wallet Stored Procedure

| PSQL SP Name | Description | Example to call | Result example |
|---|---|---|---|
| spent_confirmed_from_wallet | Procedure gets the total amount spent from wallet.<br><br>1) Get all outbound transaction ids for given wallet id.<br>TRX ID list: WALLETS_OUTB_TRXS.TRX_ID WALLETS_OUTB_TRXS.WALLET_ID=<Given wallet Id> | | |

| | | | |
|---|---|---|---|
| | 2) Get transactions that are confirmed and not rejected.<br>Per each TRANSACTIONS.TRX_ID:<br>TRANSACTIONS.CONFIRMATIONS>=6,<br>TRANSACTIONS.IS_REJECTED=false<br><br>3) Per every TRX ID, find list of inputs<br>Per each TRX ID:<br>INPUTS.TRX_ID= TRANSACTIONS.TRX_ID<br><br>4) Sum of Btc values is the total amount spent by confirmed transactions:<br>Sum of INPUTS.BTC_VALUE | | |

**Input Parameters:**

| Parameter title | PSQL Type | Java Type | Value Example | Description |
|---|---|---|---|---|
| | | | | |

**Output Parameters:**

| Parameter title | PSQL Type | Java Type | Value Example | Description |
|---|---|---|---|---|
| | | | | |

Exceptions:

| PSLQ Error code | PSQL Condition Name | Error Description | Error Returned Value | Error Returned Value Java Type |
|---|---|---|---|---|
| | | | | |

### Calculate total amount spent by pending transactions in a wallet Stored Procedure

| PSQL SP Name | Description | Example to call | Result example |
|---|---|---|---|
| spent_pending_from_wallet | Procedure gets the total amount spent from wallet but still pending (transactions not part of main blockchain).<br><br>1) Get all outbound transaction ids for given wallet id.<br>TRX ID list: WALLETS_OUTB_TRXS.TRX_ID<br>WALLETS_OUTB_TRXS.WALLET_ID=<Given wallet Id><br><br>2) Get transactions that are pending and not rejected. | | |

| | | | |
|---|---|---|---|
| | Per each TRANSACTIONS.TRX_ID:<br>TRANSACTIONS.CONFIRMATIONS < 6,<br>TRANSACTIONS.IS_REJECTED=false<br><br>3) Per every TRX ID, find list of inputs<br>Per each TRX ID:<br>INPUTS.TRX_ID= TRANSACTIONS.TRX_ID<br><br>4) Sum of Btc values is the total amount spent but still pending:<br>Sum of INPUTS.BTC_VALUE | | |

**Input Parameters:**

| Parameter title | PSQL Type | Java Type | Value Example | Description |
|---|---|---|---|---|
| | | | | |

**Output Parameters:**

| Parameter title | PSQL Type | Java Type | Value Example | Description |
|---|---|---|---|---|
| | | | | |

Exceptions:

| PSLQ Error code | PSQL Condition Name | Error Description | Error Returned Value | Error Returned Value Java Type |
|---|---|---|---|---|
| | | | | |

### Delete temporary Trx data Stored Procedure

| PSQL SP Name | Description | Example to call | Result example |
|---|---|---|---|
| delete_temp_trx | Procedure deletes temporary transaction record and corresponding Inputs/Outputs records. TrxMC DB is “trx_management”.<br>DELETE FROM OUTPUTS WHERE TEMP_TRX_ID= …<br>DELETE FROM INPUTS WHERE TEMP_TRX_ID …<br>DELETE FROM TEMP_OUTB_TRXS WHERE TEMP_TRX_ID= … AND EXTERNAL_TRX_ID= … | | |

**Input Parameters:**

| Parameter title | PSQL Type | Java Type | Value Example | Description |
|---|---|---|---|---|

| | | | | |
|---|---|---|---|---|
| | | | | |

**Output Parameters:**

| Parameter title | PSQL Type | Java Type | Value Example | Description |
|---|---|---|---|---|
| | | | | |

**Exceptions:**

| PSLQ Error code | PSQL Condition Name | Error Description | Error Returned Value | Error Returned Value Java Type |
|---|---|---|---|---|
| | | | | |

### Get all UTXOs balance for given Wallet

| PSQL SP Name | Description | Example to call | Result example |
|---|---|---|---|
| utxos_balance_for_wallet | Procedure calculates current UTXOs balance for wallet.<br>1) Get a list of unspent outputs (UTXOs): Call “utxos_for_wallet(…)”<br>2) Sum of UTXOs Outputs Btc values is the required balance:<br>Sum of OUTPUTS.BTC_VALUE | | |

**Input Parameters:**

| Parameter title | PSQL Type | Java Type | Value Example | Description |
|---|---|---|---|---|
| | | | | |

**Output Parameters:**

| Parameter title | PSQL Type | Java Type | Value Example | Description |
|---|---|---|---|---|
| | | | | |

**Exceptions:**

| PSLQ Error code | PSQL Condition Name | Error Description | Error Returned Value | Error Returned Value Java Type |
|---|---|---|---|---|
| | | | | |

### Get Wallet Balance data for wallet Function

| PSQL Func Name | Description | Example to call | Result example |
|---|---|---|---|
| get_wallet_balance | Function gets the balance data (Available Balance, Current Balance) for a wallet id or a set of wallet ids. | | |

| | | | |
|---|---|---|---|
| | 1) Call following stored procedures for wallet id(s):<br>unspent_amnt = SELECT **utxos_balance_for_wallet(…)**<br><br>confirmed_spent = SELECT **spent_confirmed_from_wallet(…)**<br><br>pending_spent = SELECT **spent_pending_from_wallet(…)**<br><br>2) Calculate available balance<br>avlblBalance = unspent_amnt – (confirmed_spent + pending_spent)<br><br>3) Calculate current balance<br>currBalance = unspent_amnt – confirmed_spent<br><br>4) Update record in WALLETS table:<br>BTC_AVAILABLE_FUNDS = <avlblBalance><br>BTC_BALANCE = <currBalance><br>DATE_UPDATED = <now><br><br>5) Get required fields from WALLETS table:<br>SELECT BTC_AVAILABLE_FUNDS,<br>BTC_BALANCE,<br>IS_LOCKED<br>where WALLET_ID = <Given wallet id> | | |

**Input Parameters:**

| Parameter title | PSQL Type | Java Type | Value Example | Description |
|---|---|---|---|---|
| Wallet ids | | | | |
| array of wallet ids | | | | |

**Output Parameters:**

| Parameter title | PSQL Type | Java Type | Value Example | Description |
|---|---|---|---|---|
| | | | | |

Exceptions:

| PSLQ Error code | PSQL Condition Name | Error Description | Error Returned Value | Error Returned Value Java Type |
|---|---|---|---|---|
| | | | | |

### Get Wallet data for wallet Stored Procedure

| PSQL Func Name | Description | Example to call | Result example |
|---|---|---|---|
| get_wallet_data | Procedure gets the wallet data for wallet.<br>1) Call Function get_wallet_balance() for given wallet to get available balance, current balance and IS_LOCKED for given wallet id:<br>SELECT **get_wallet_balance(…)**<br><br>2) From table WALLETS, get value of field IS_SYSTEM_WALLET for given wallet id:<br>SELECT IS_SYSTEM_WALLET from WALLETS where WALLET_ID = <Given wallet Id><br><br>3) Get no. of input transactions:<br>SELECT COUNT(TRX_ID) from WALLETS_INB_TRXS where WALLET_ID = <Given wallet id><br><br>4) Get no. of output transactions<br>SELECT COUNT(TRX_ID) from WALLETS_OUTB_TRXS where WALLET_ID = <Given wallet id> | | |

**Input Parameters:**

| Parameter title | PSQL Type | Java Type | Value Example | Description |
|---|---|---|---|---|
| | | | | |
| | | | | |

**Output Parameters:**

| Parameter title | PSQL Type | Java Type | Value Example | Description |
|---|---|---|---|---|
| | | | | |

Exceptions:

| PSLQ Error code | PSQL Condition Name | Error Description | Error Returned Value | Error Returned Value Java Type |
|---|---|---|---|---|
| | | | | |

### Get outbound transaction data Stored Procedure

| PSQL Func Name | Description | Example to call | Result example |
|---|---|---|---|

| | | | |
|---|---|---|---|
| get_outb_trx_data | Procedure gets transaction data for outbound transaction.<br>1) From table TRANSACTIONS, get **daemonTrxId, isRejected** for given transaction id:<br>SELECT TRX_ID, DAEMON_TXID_HASH, IS_REJECTED from TRANSACTIONS where DAEMON_TXID_HASH = <Given transaction Id><br><br>2) From table TRX_STATUSES, get **daemonTrxStatus** for given transaction id:<br>SELECT STATUS from TRX_STATUSES where TRX_STATUS_ID = TRANSACTIONS.TRX_STATUS_ID and TRANSACTIONS. DAEMON_TXID_HASH = <Given transaction Id><br><br>3) Assign a bool value to **isConfirmed** depending on number of confirmations:<br>confirmations = SELECT CONFIRMATIONS from TRANSACTIONS where DAEMON_TXID_HASH = <Given transaction Id><br>If confirmations >=6, isConfirmed = true<br>Else isConfirmed = false<br><br>4) Calculate miner fee.<br>sumOfInputs =<br>SELECT SUM(VOUT_BTC_VALUE) from INPUTS WHERE TRX_ID = <TRX_ID from step 1><br><br>sumOfOutputs =<br>SELECT SUM(BTC_VALUE) from OUTPUTS WHERE TRX_ID = <TRX_ID from step 1><br><br>**minerFee** = sumOfInputs – sumOfOutputs<br>Return minerFee only if minerFee > 0<br><br>5) Get system fee.<br>sysFeePresent = SELECT HAS_SYSTEM_FEE from TRANSACTIONS where TRANSACTION_ID = <Given transaction Id><br>If sysFeePresent = true:<br>**systemFee** = SELECT BTC_VALUE from OUTPUTS where TRX_ID = <Given transaction Id> AND IS_SYSTEM_FEE = true<br><br>Return daemonTrxId, isRejected, daemonTrxStatus, isConfirmed, minerFee, systemFee | | |

**Input Parameters:**

| Parameter title | PSQL Type | Java Type | Value Example | Description |
|---|---|---|---|---|
| daemonTrxId | | | | |
| | | | | |

**Output Parameters:**

| Parameter title | PSQL Type | Java Type | Value Example | Description |
|---|---|---|---|---|
| daemonTrxId | | | | |
| isRejected | | | | |
| daemonTrxStatus | | | | |
| isConfirmed | | | | |
| minerFee | | | | |
| systemFee | | | | |

Exceptions:

| PSLQ Error code | PSQL Condition Name | Error Description | Error Returned Value | Error Returned Value Java Type |
|---|---|---|---|---|
| | | | | |

## Get Inbound transactions for a bitcoin address Stored Procedure

| PSQL SP Name | Description | Example to call | Result example |
|---|---|---|---|
| get_inb_trx_data | Procedure gets all inbound transactions for given bitcoin address.<br><br>1) Get all inbound transaction ids for given wallet id.<br>TRX ID list: WALLETS_INB_TRXS.TRX_ID<br>WALLETS_INB_TRXS.WALLET_ID=<Given wallet Id><br><br>2) Get daemon trx hash and date created for each transaction.<br>Per each TRANSACTIONS.TRX_ID:<br>DAEMON_TXID_HASH,<br>DATE_CREATED<br><br>3) Per every TRX ID, find outputs that have destination address same as given btc address.<br>Per each TRX ID:<br>OUTPUTS.TRX_ID=<br>TRANSACTIONS.TRX_ID | | |

| | | | |
|---|---|---|---|
| | OUTPUTS.BTC_ADDRESS = <Given btc address><br><br>4) Sum of Btc values is the total output amount for given btc address per transaction:<br>Sum of OUTPUTS.BTC_VALUE | | |

**Input Parameters:**

| Parameter title | PSQL Type | Java Type | Value Example | Description |
|---|---|---|---|---|
| btcAddress | | | | |
| daemonWalletId | | | | |

**Output Parameters:**

| Parameter title | PSQL Type | Java Type | Value Example | Description |
|---|---|---|---|---|
| daemonTrxid | | | | |
| btcAmount | | | | |
| dateCreated | | | | |
| | | | | |

Exceptions:

| PSLQ Error code | PSQL Condition Name | Error Description | Error Returned Value | Error Returned Value Java Type |
|---|---|---|---|---|
| | | | | |

### Send Raw Transaction error

| PSQL SP Name | Description | Example to call | Result example |
|---|---|---|---|
| send_raw_trx_error | Procedure creates record in the TRANSACTIONS_ERROR_CEDES table and update Trx status<br>2) INSERT INTO TRANSACTIONS_ERRORO_CODES (…) VALUES (…)<br>TRX_ID=<TRANSACTIONS.TRX_ID><br>ERROR_CODE_ID=7<br>DATE_CREATED=<now><br><br>2) UPDATE TRANSACTIONS SET TRX_STATUS_ID=4, DATE_UPDATED=<now> WHERE TRX_ID=<transaction Id> | | |

**Input Parameters:**

| Parameter title | PSQL Type | Java Type | Value Example | Description |
|---|---|---|---|---|
| | | | | |

**Output Parameters:**

| Parameter title | PSQL Type | Java Type | Value Example | Description |
|---|---|---|---|---|
| | | | | |

**Exceptions:**

| PSLQ Error code | PSQL Condition Name | Error Description | Error Returned Value | Error Returned Value Java Type |
|---|---|---|---|---|
| | | | | |

### Select all UTXOs for given Wallet

| PSQL SP Name | Description | Example to call | Result example |
|---|---|---|---|
| utxos_for_wallet | Procedure selects all current UTXOs for wallet. Criteria:<br>1) Get all inbound transaction ids for given wallet id.<br>WALLETS_INB_TRXS.TRX_ID<br>WALLETS_INB_TRXS.WALLET_ID=<Given wallet Id><br>2) Get transactions that are confirmed.<br>Per each TRANSACTIONS.TRX_ID:<br>TRANSACTIONS.CONFIRMATIONS>=6,<br>TRANSACTIONS.IS_EVERY_OUTPUT_SPENT=false<br>3) Per every TRX ID, find list of unspent outputs (UTXOs)<br>Per each TRX ID:<br>OUTPUTS.TRX_ID=<br>TRANSACTIONS.TRX_ID<br>OUTPUTS.IS_SPENT=false<br>4) Only consider those outputs whose btc addresses belong to the wallet.<br>Per each Output:<br>SYSTEM_BTC_ADDRESSES.BTC_ADDRESS=OUTPUTS.BTC_ADDRESS<br>SYSTEM_BTC_ADDRESSES.WALLET_ID=<Given wallet Id> | | |

**Input Parameters:**

| Parameter title | PSQL Type | Java Type | Value Example | Description |
|---|---|---|---|---|

| | | | | |
|---|---|---|---|---|
| | | | | |

**Output Parameters:**

| Parameter title | PSQL Type | Java Type | Value Example | Description |
|---|---|---|---|---|
| | | | | |

**Exceptions:**

| PSLQ Error code | PSQL Condition Name | Error Description | Error Returned Value | Error Returned Value Java Type |
|---|---|---|---|---|
| | | | | |

### Update Outbound Trx Data after issuing into Block Chain

| PSQL SP Name | Description | Example to call | Result example |
|---|---|---|---|
| data_after_send_raw_trx | Procedure updates Trx status to “Pending” …<br>1) UPDATE TRANSACTIONS SET TRX_STATUS_ID=2, DATE_UPDATED=<now>, DAEMON_TXID_HASH=<Trx hash returned by Daemon after sending> WHERE TRX_ID=<transaction Id><br>2) Every Outputs of previous Inbound Trxs according to UTXOs data should be updated<br><br>UPDATE OUTPUTS SET IS_SPENT=true, DATE_SPENT=<now> WHERE SPENT_BY_INPUT_ID=<Input Id of current transaction><br><br>3) Check previous Inbound Trxs according to UTXOs if every Outputs were spent.<br>UPDATE TRANSACTIONS SET IS_EVERY_OUTPUT_SPENT=true WHERE TRX_ID=<UTXOs Trx Id><br><br>4) If there is a change: UPDATE SYSTEM_BTC_ADDRESSES SET IS_USED=true WHERE BTC_ADDRESS=<change Btc address> AND | | |

| | | | |
|---|---|---|---|
| | WALLET_ID=<Wallet Id related to this Trx><br>5) If there is a company fee:<br>UPDATE SYSTEM_BTC_ADDRESSES SET IS_USED=true WHERE BTC_ADDRESS=<company fee Btc address> | | |

**Input Parameters:**

| Parameter title | PSQL Type | Java Type | Value Example | Description |
|---|---|---|---|---|
| | | | | |

**Output Parameters:**

| Parameter title | PSQL Type | Java Type | Value Example | Description |
|---|---|---|---|---|
| | | | | |

**Exceptions:**

| PSLQ Error code | PSQL Condition Name | Error Description | Error Returned Value | Error Returned Value Java Type |
|---|---|---|---|---|
| | | | | |

## 3.9.2 Monitoring System Functions and Stored Procedures

### 3.9.2.1 move_inc_to_archive

Parameters:

array of btc addresses identified as incoming (list_inc_addr)

Return:

error from SQl server or Success

1. Copy respective records from STRXMSS_MONITORED_BTC_ADDR to Archive table

INSERT INTO Archive

SELECT * from STRXMSS_MONITORED_BTC_ADDR WHERE BTC_ADDR = list_inc_addr[n];

2. Delete corresponding records from STRXMSS_INB_TRXS_BTC_ADDR, STRXMSS_MONITORED_INB_TRXS

# 4. Intelligent Daemon System Workflow Diagrams

The following notation is used:

- Y – Yes
- N – No
- OK – Positive Result
- Err – System Error
- P[a-z][0-9] – Process/Sub-process [group alfa] [process number]
- C[a-z][0-9] – Connector [group alfa] [connector number]
- (…) - Expression
- […] – Variable
- <…> - Value of variable

***Note**: Log[DEBUG] lines were not included in workflow diagram consideration. Debug lines should be included in an each component source code in the development stage if necessary.*

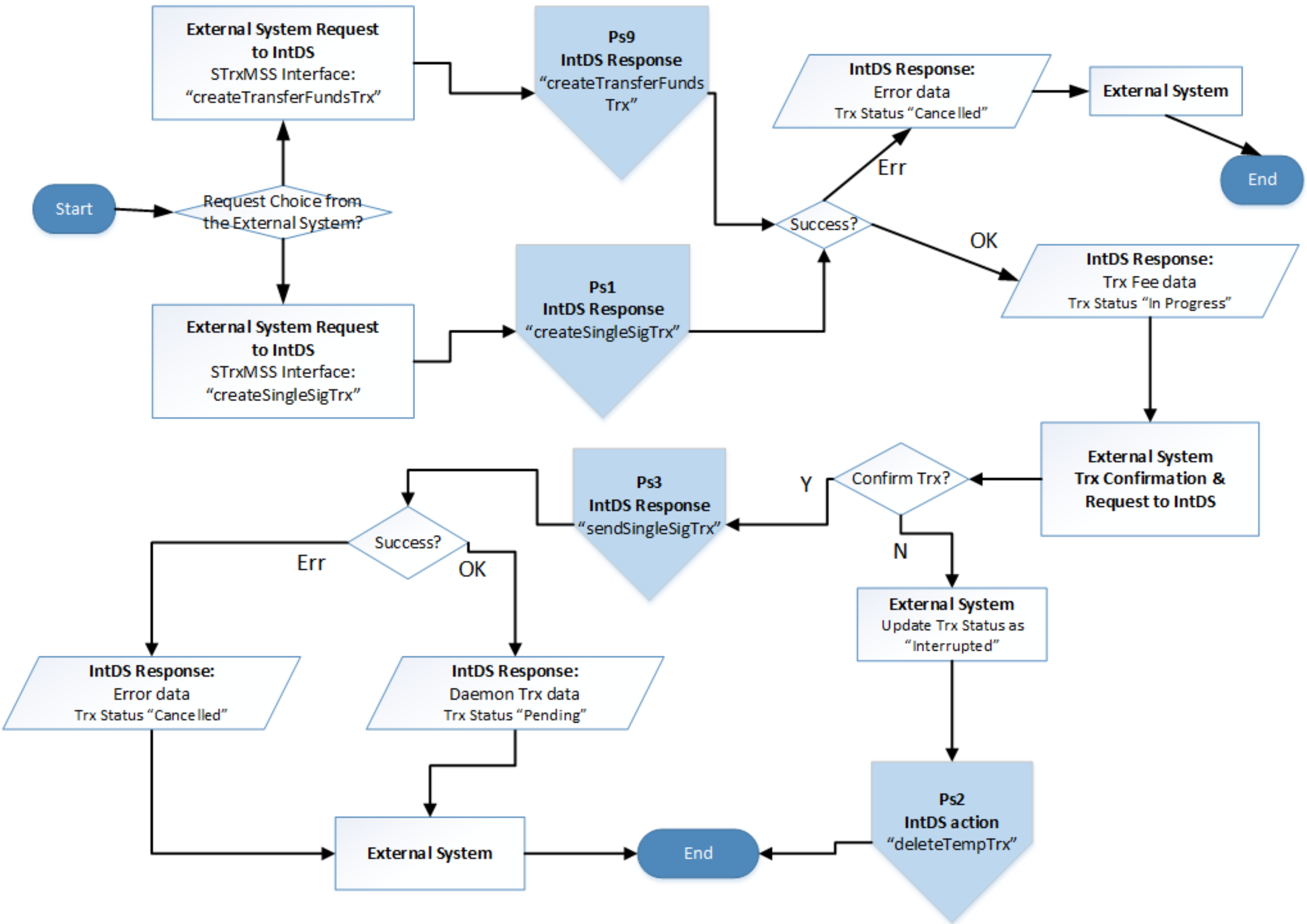
## 4.1 Single-sig Transaction Management SubSystem Workflows

STrxMSS interface description can be found in the paragraph “5.1. Single-sig Transaction Management SubSystem Interface”

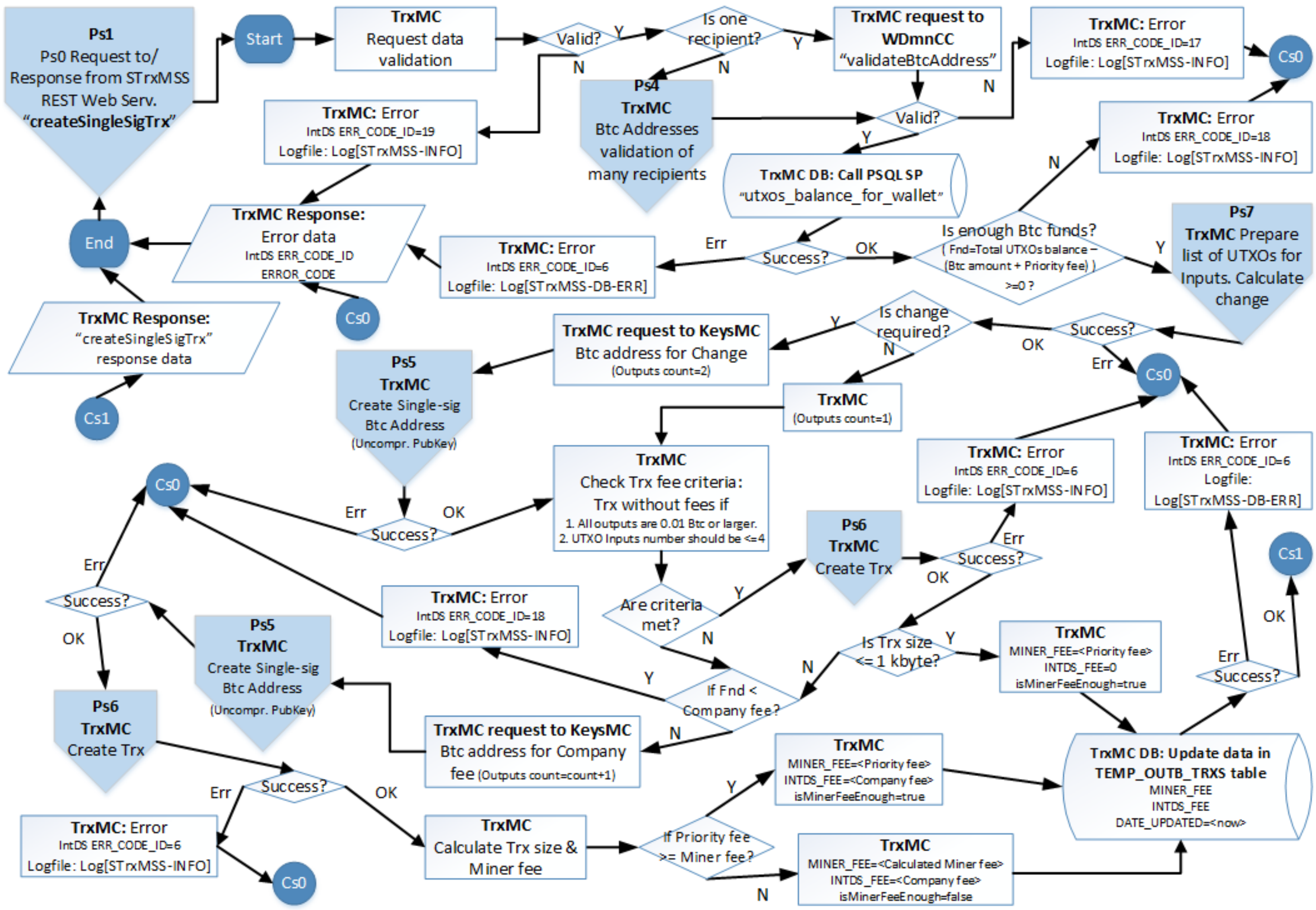
### 4.1.1 Outbound Transaction Workflows

**Diagram Ps0. High Level Diagram. Create Outbound Single-sig Trx:**

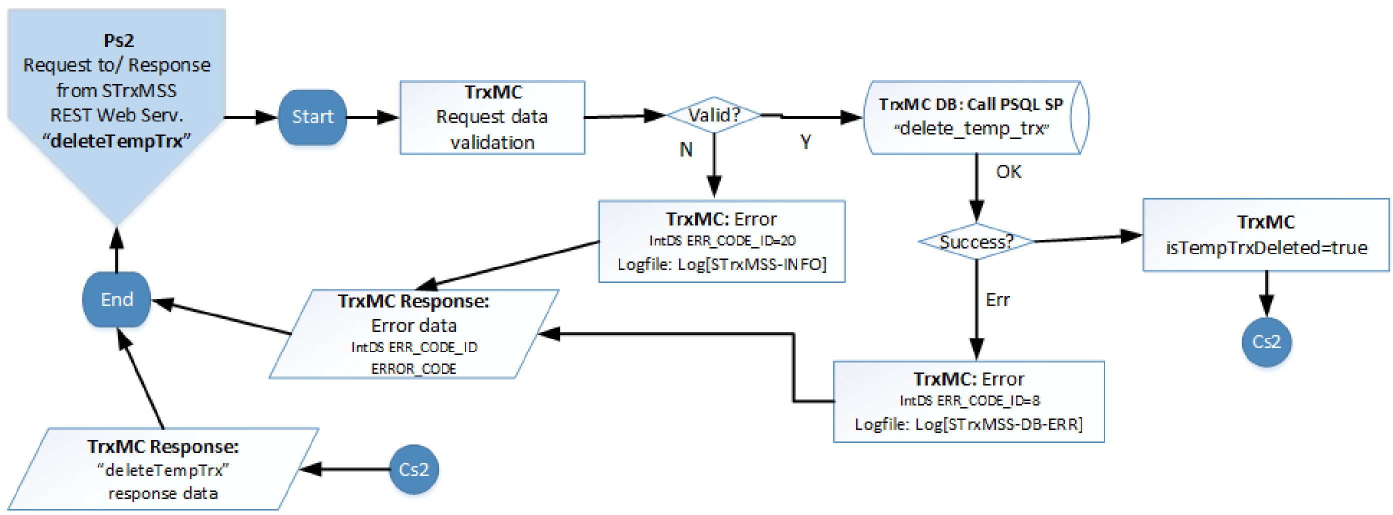
All Inputs and Outputs of this transaction are correspond to P2PKH type only.



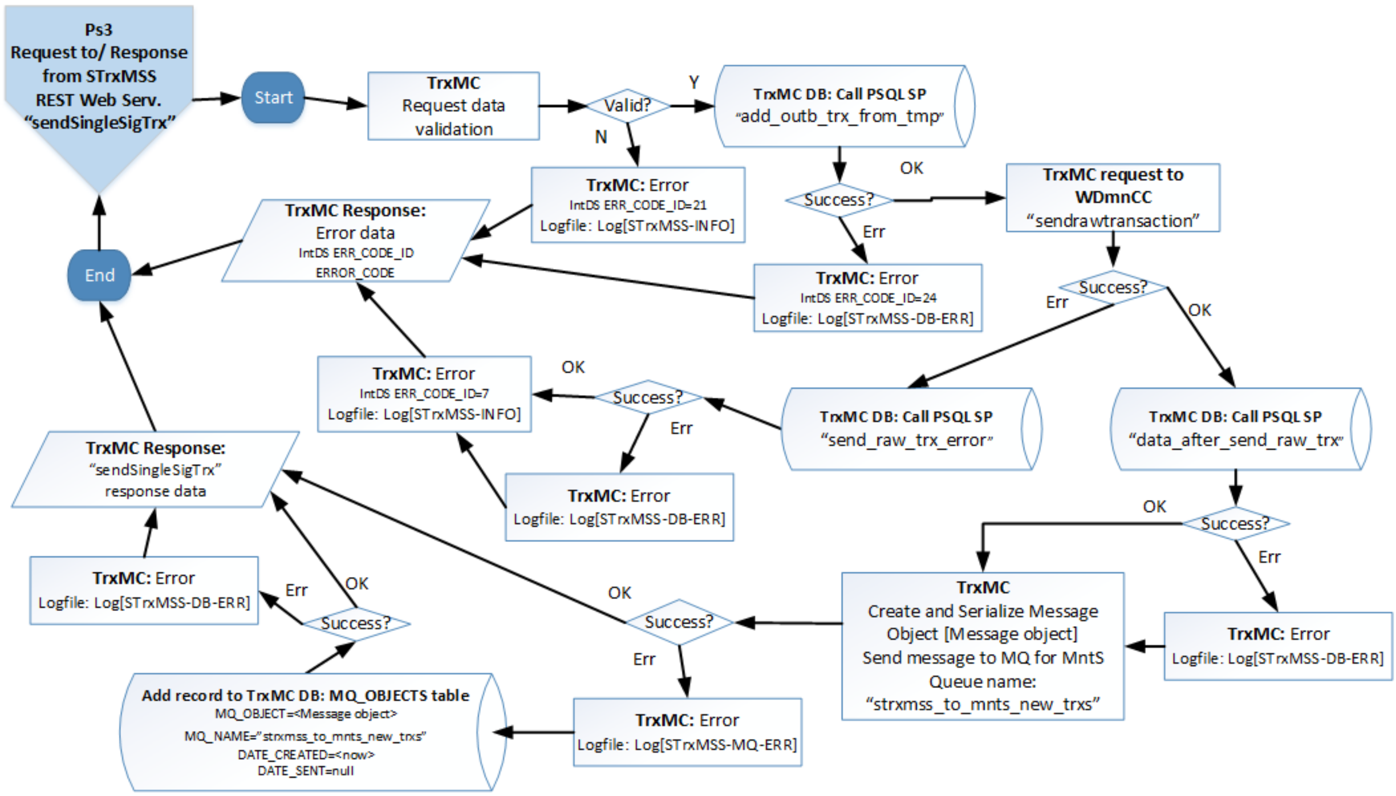
## Diagram Ps1. Create Temporary Single-sig Trx: “createSingleSigTrx” function



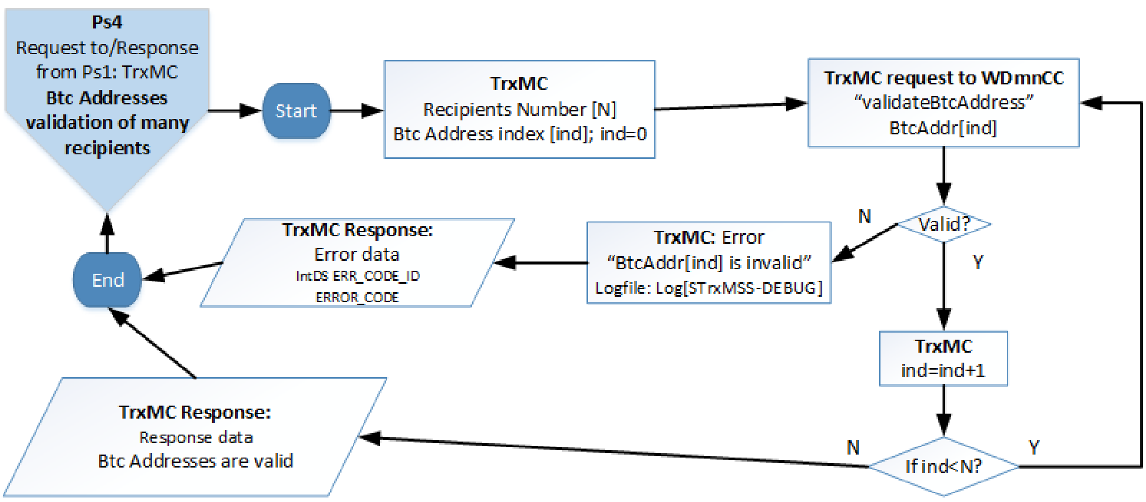

## Diagram Ps2. Delete Temporary Single-sig Trx: “deleteTempTrx” function



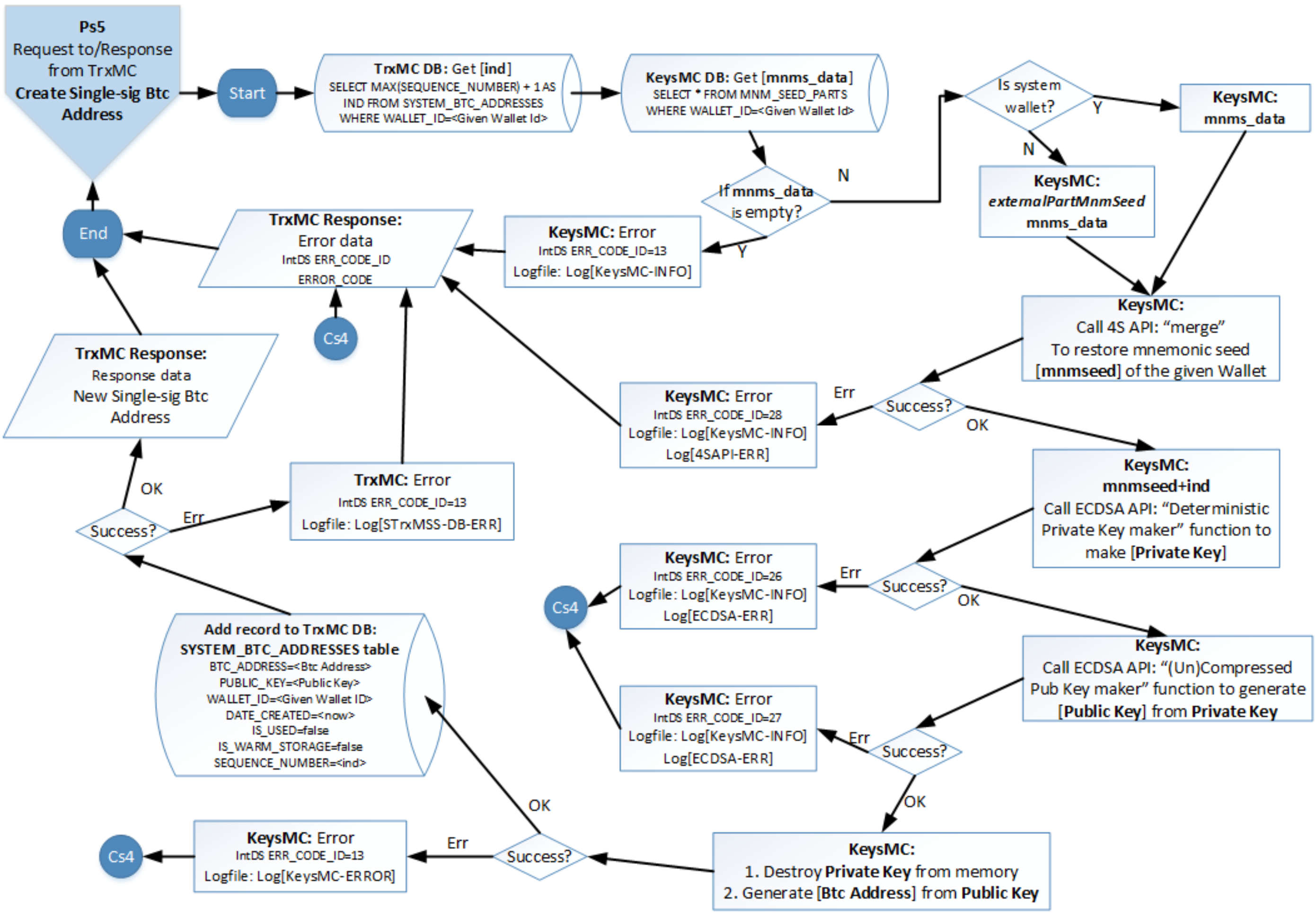
## Diagram Ps3. Send Single-sig Trx to blockchain: "sendSingleSigTrx" function



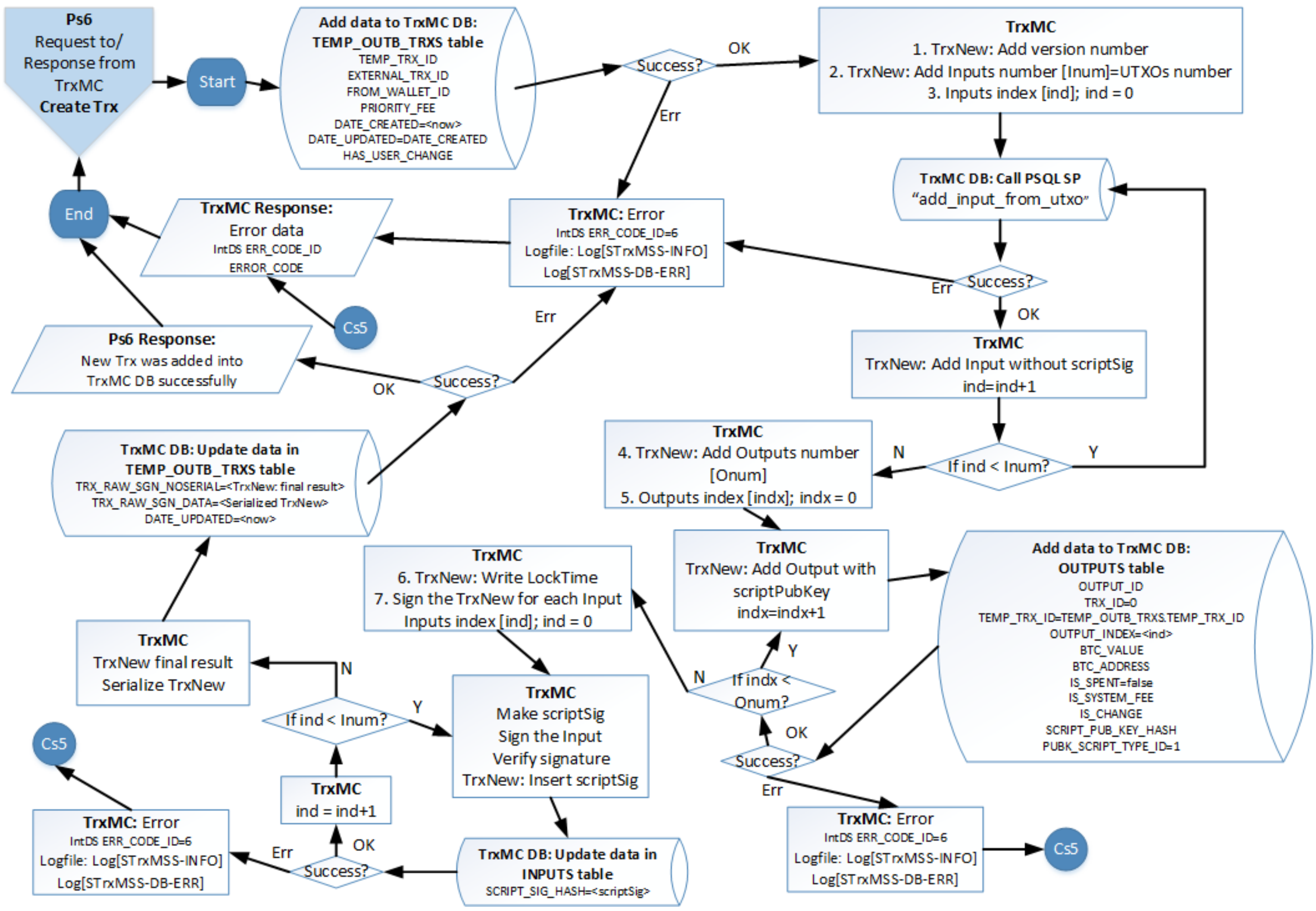

## Diagram Ps4. Btc Addresses Validation of many recipients:



## Diagram Ps5. Create Single-sig Btc Address:



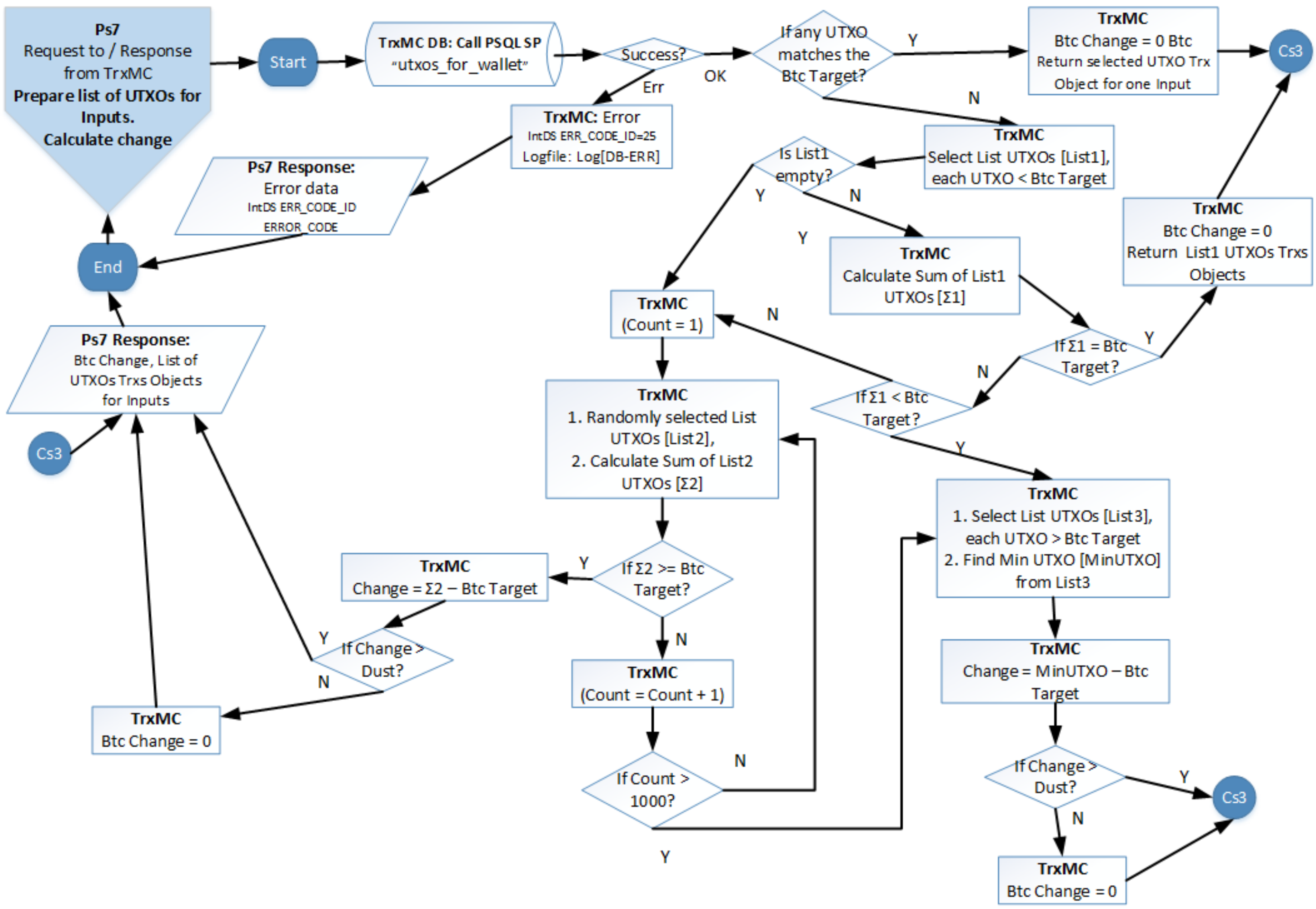
## Diagram Ps6. Create Single-sig Transaction (P2PKH):



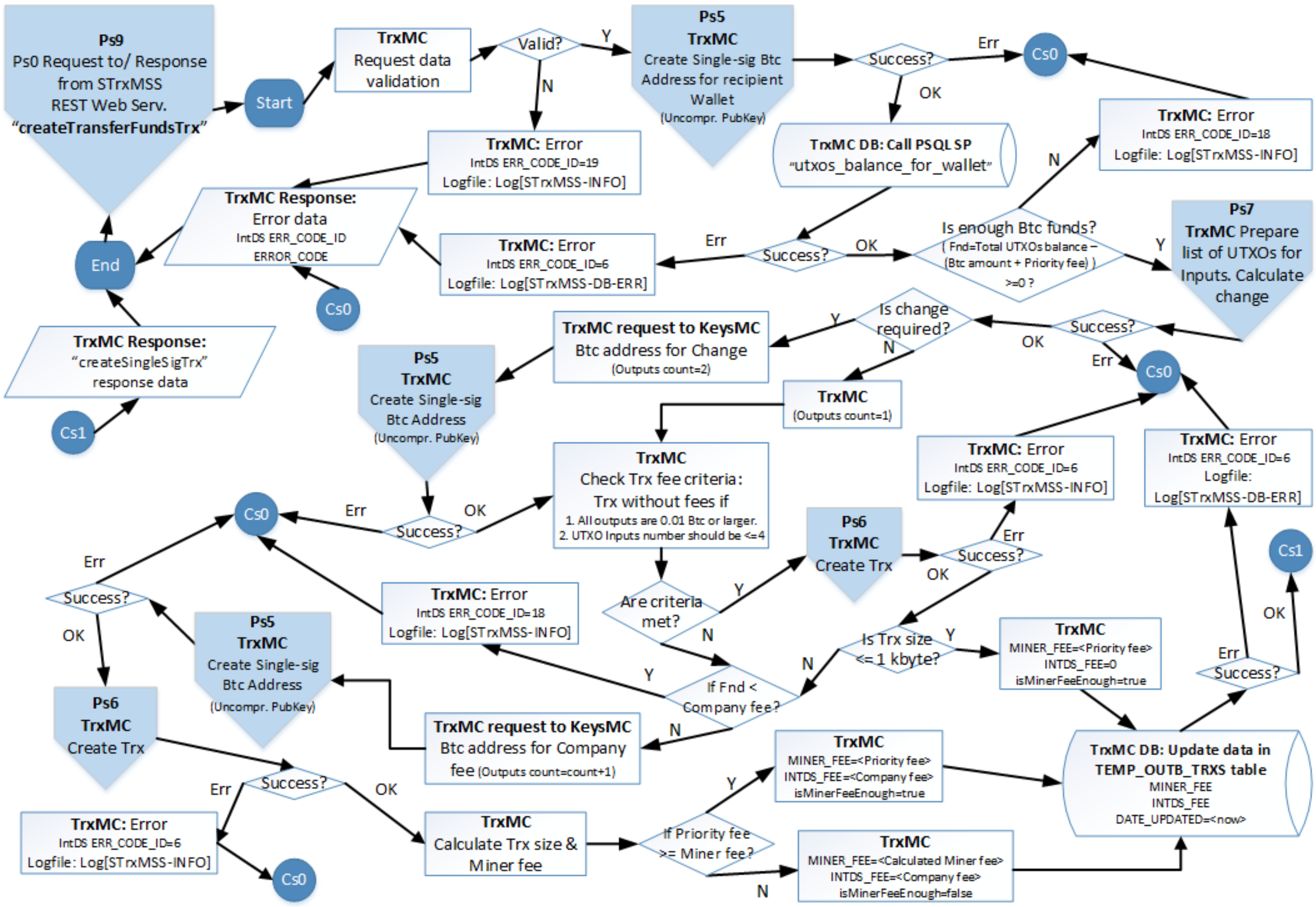
## Diagram Ps7. Prepare List of UTXOs:

Diagram criteria are:

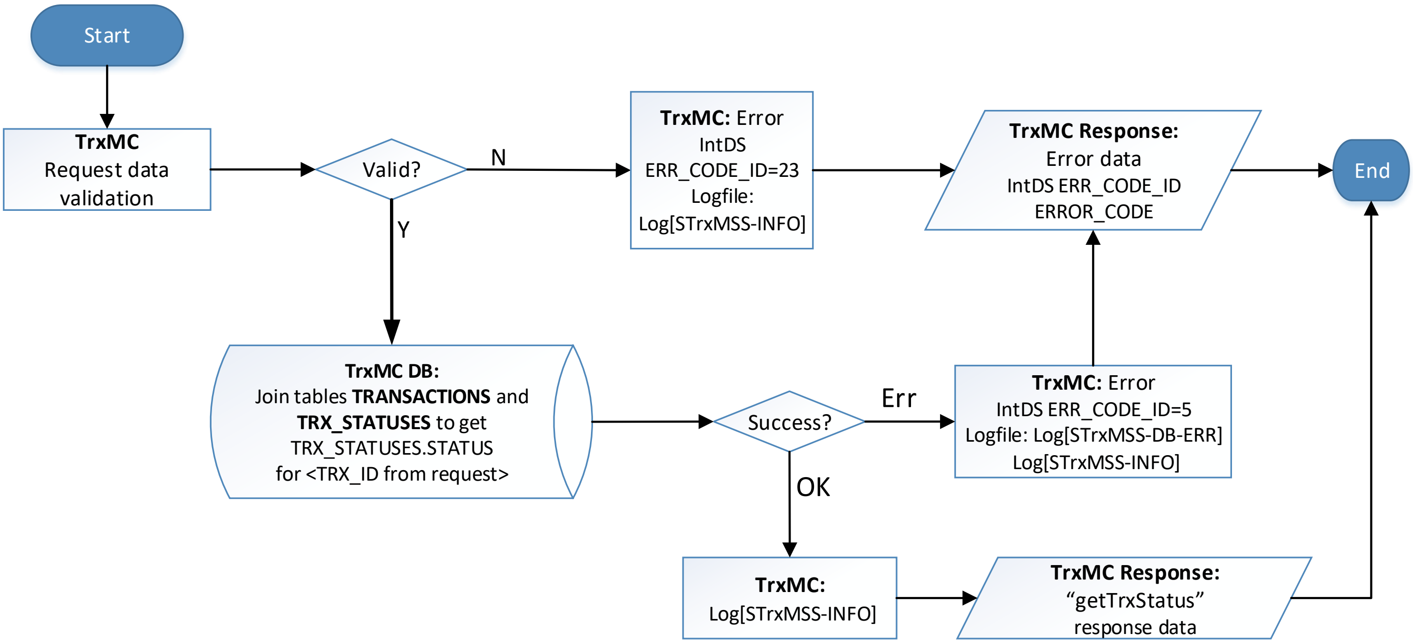
1. "Btc Target" is used here for the Btc total amount to be spent plus Priority fee if applied. Number of Outputs does not matter. *Btc Target = Btc amount + Priority fee*
2. Btc funds is enough. *Total UTXOs balance - Btc Target >= 0*



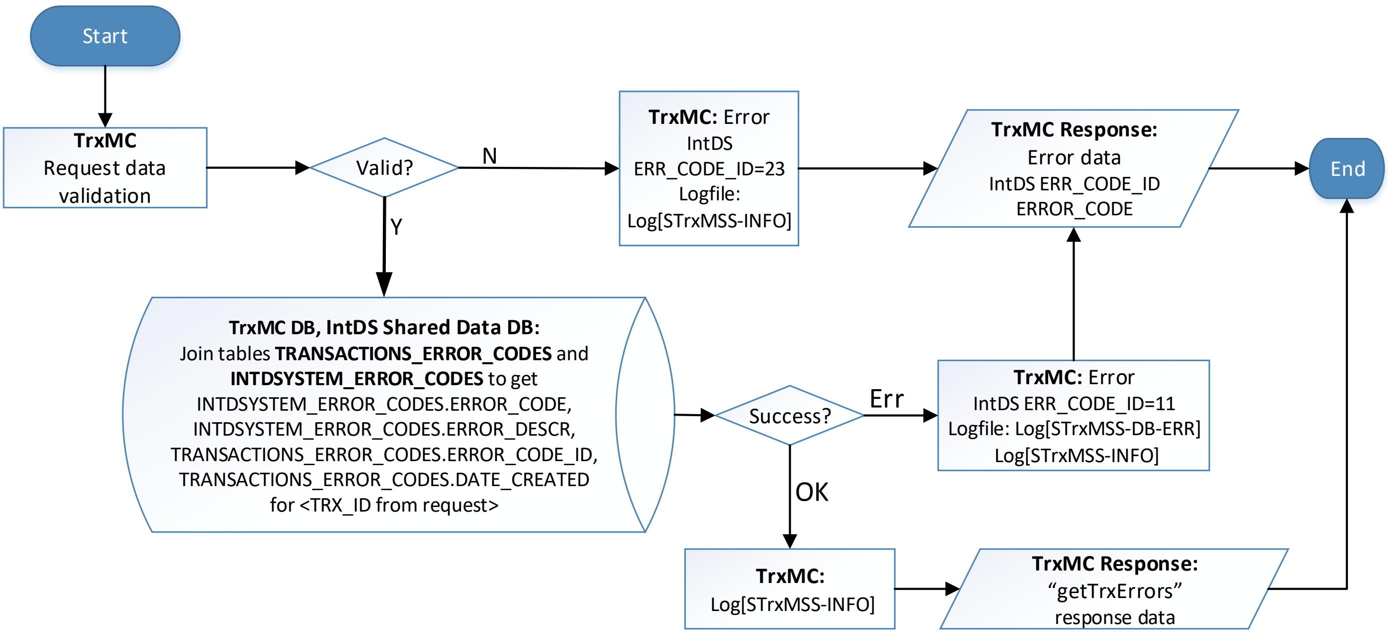
## Diagram Ps9. Create Temporary Single-sig Trx: “createTransferFundsTrx” function



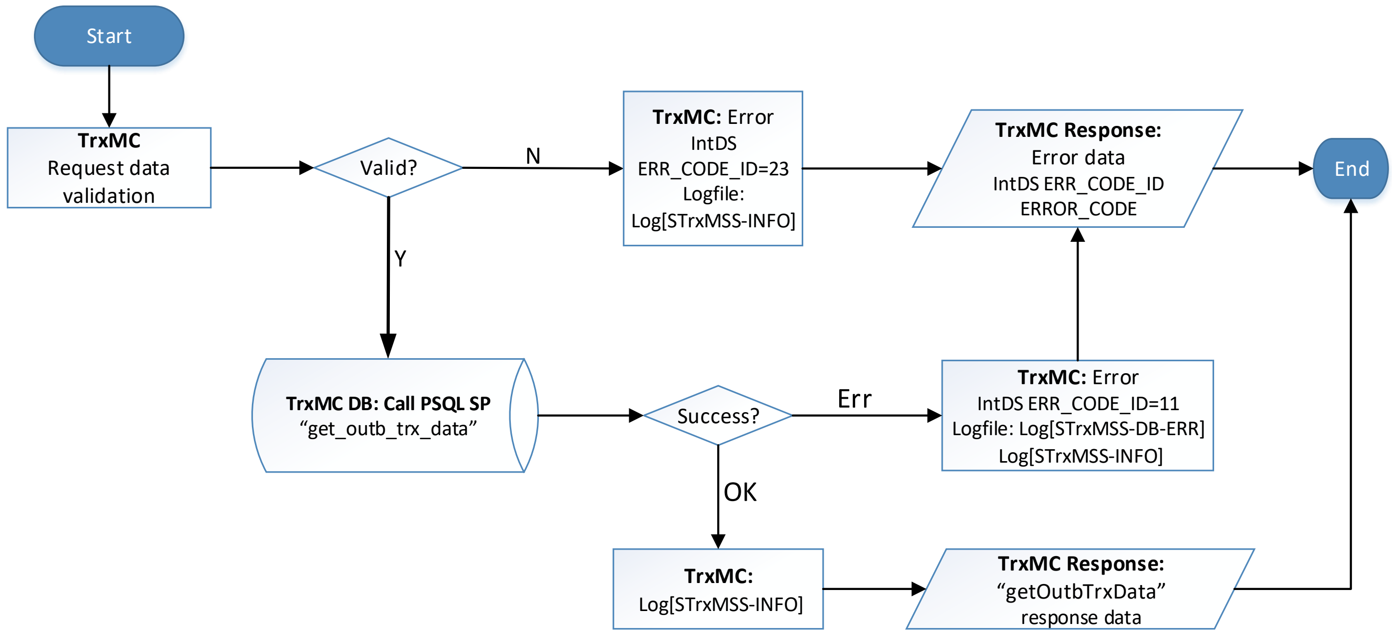
### **Get Transaction Status:** "getTrxStatus" function




### **Get Transaction Errors:** "getTrxErrors" function

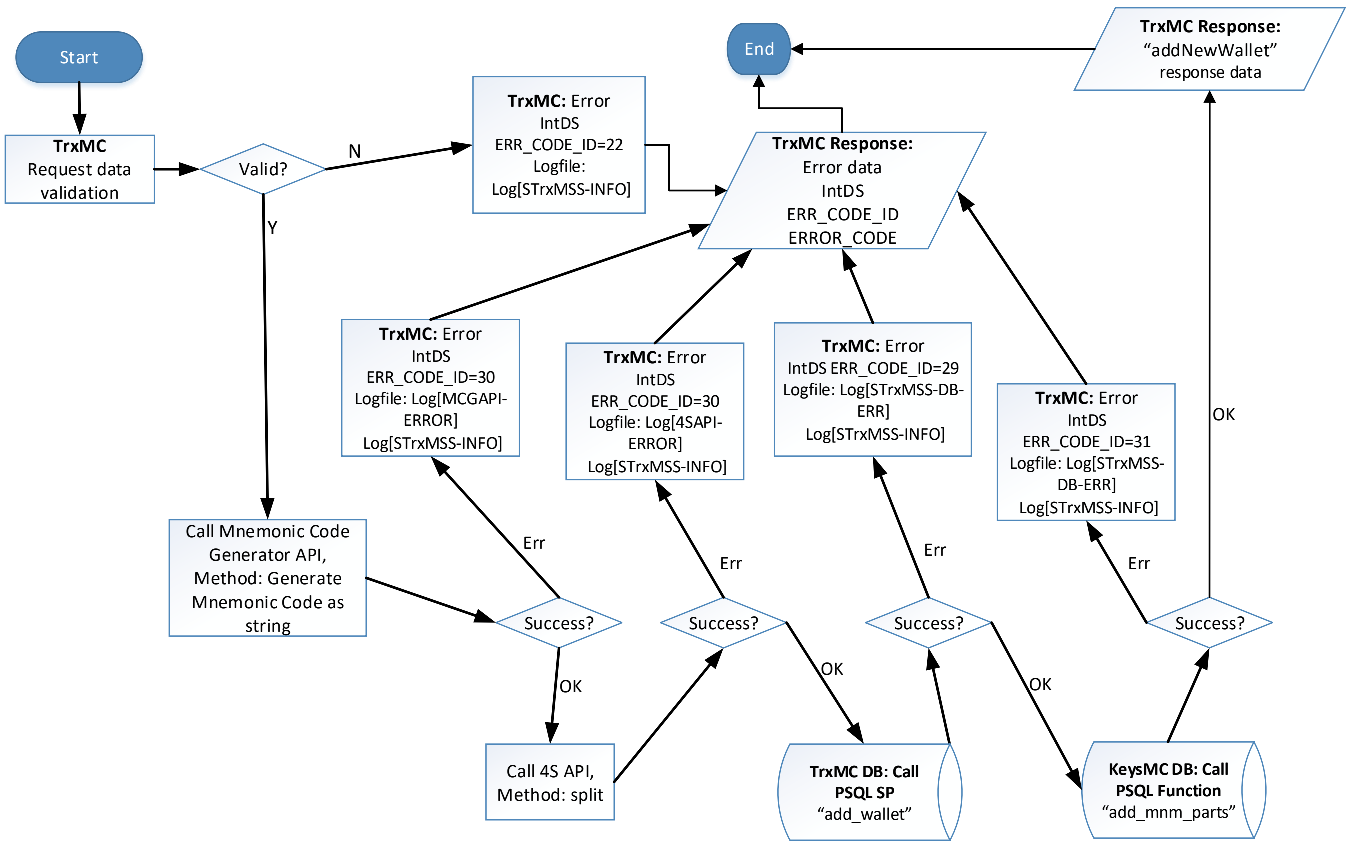


### **Get Data for Given Outbound Transaction:** "getOutbTrxData" function

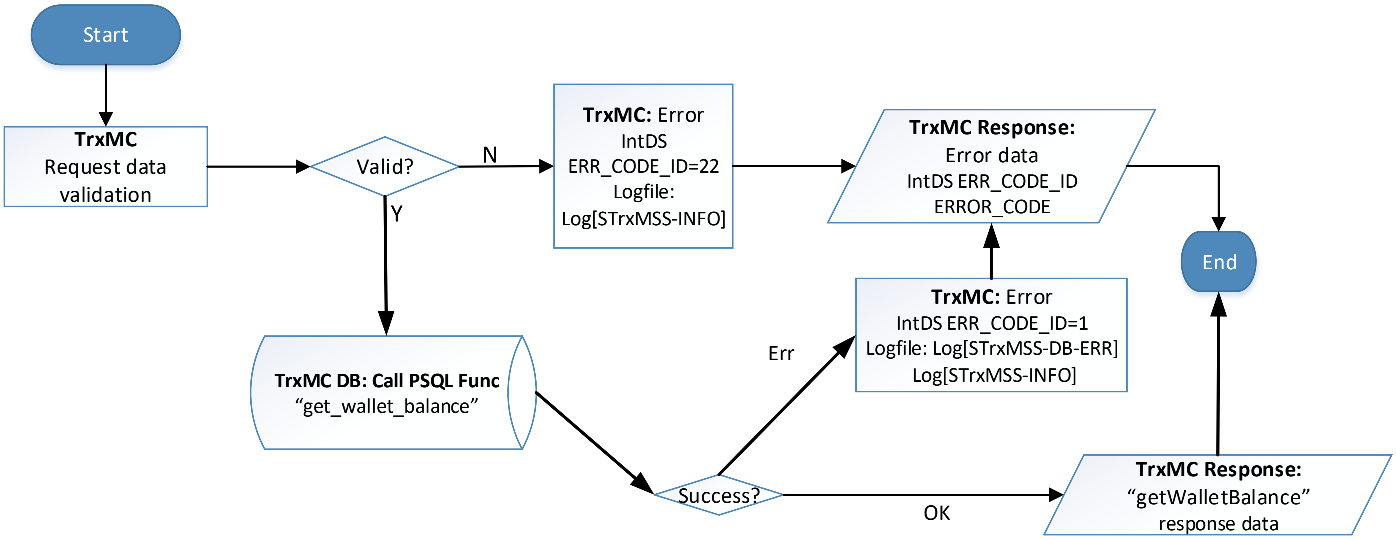



## 4.1.2 Wallet Functions Workflows

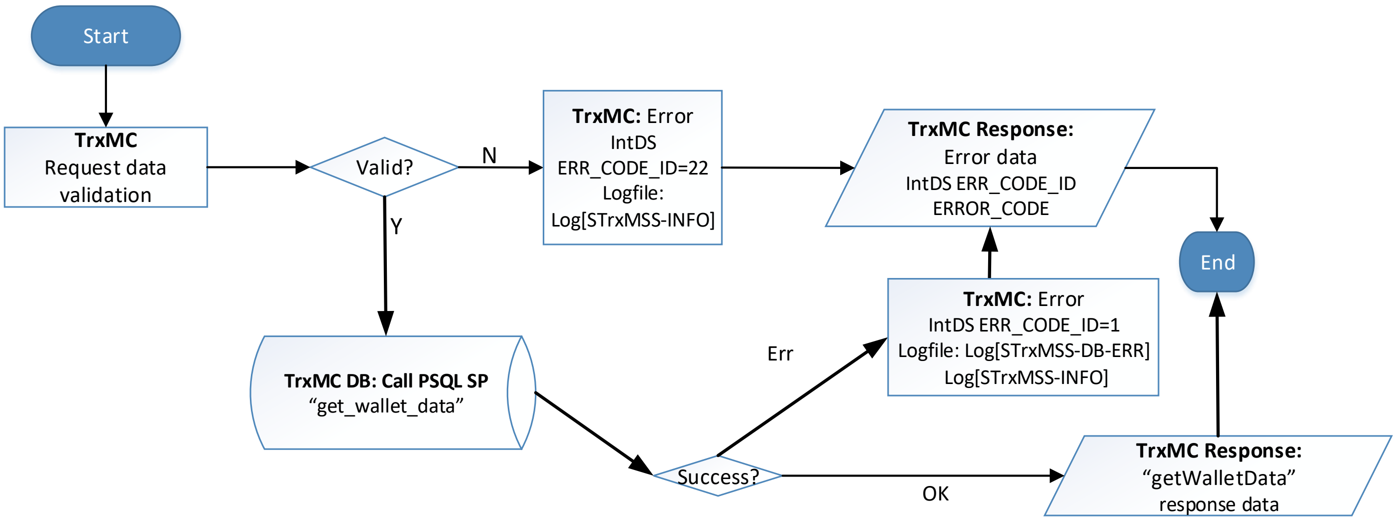
### **Add New Wallet:** "addNewWallet" function



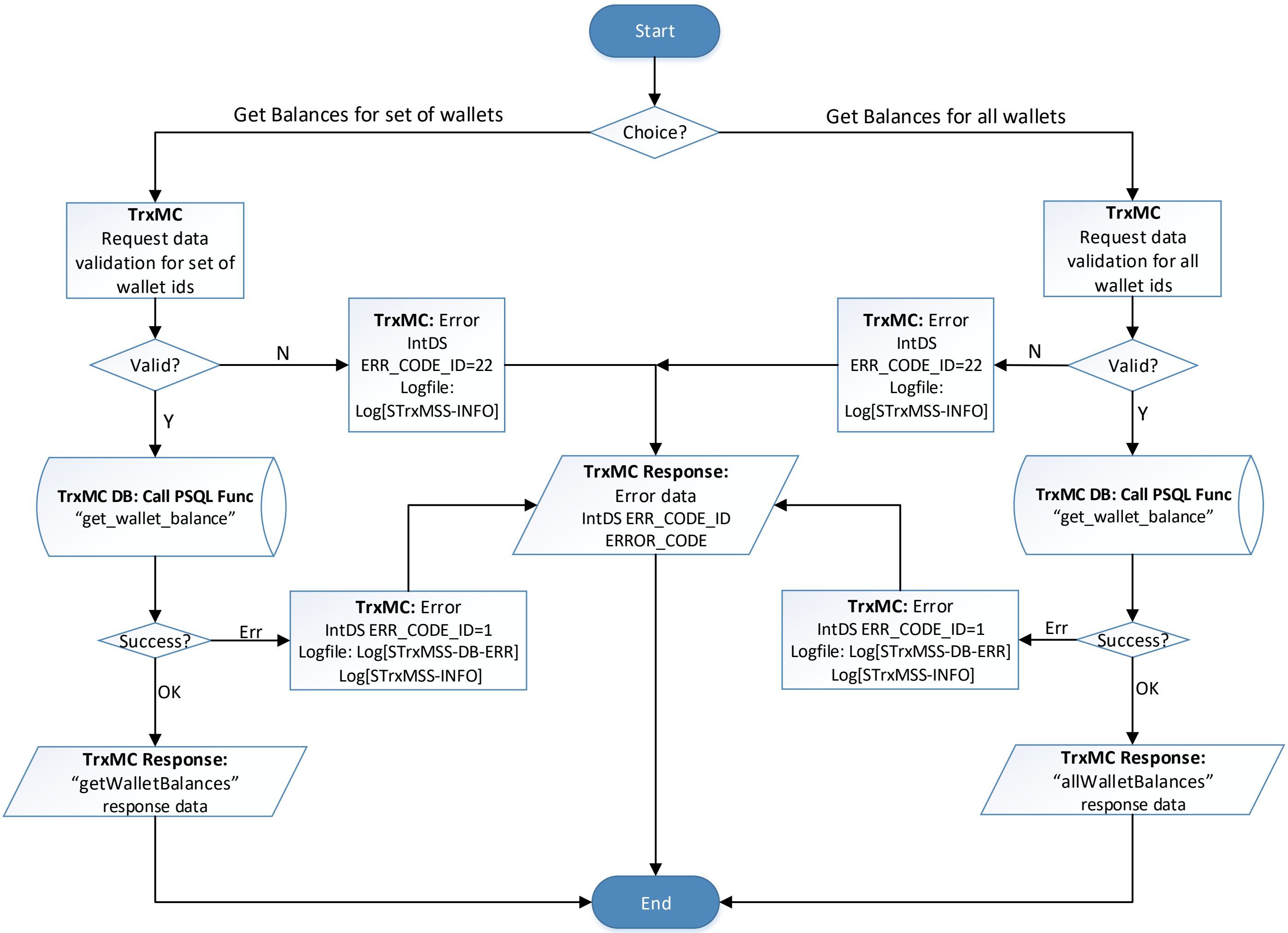
## **Get Balance for Given Wallet:** “getWalletBalance” function




## **Get Data Associated with Given Wallet:** “getWalletData” function



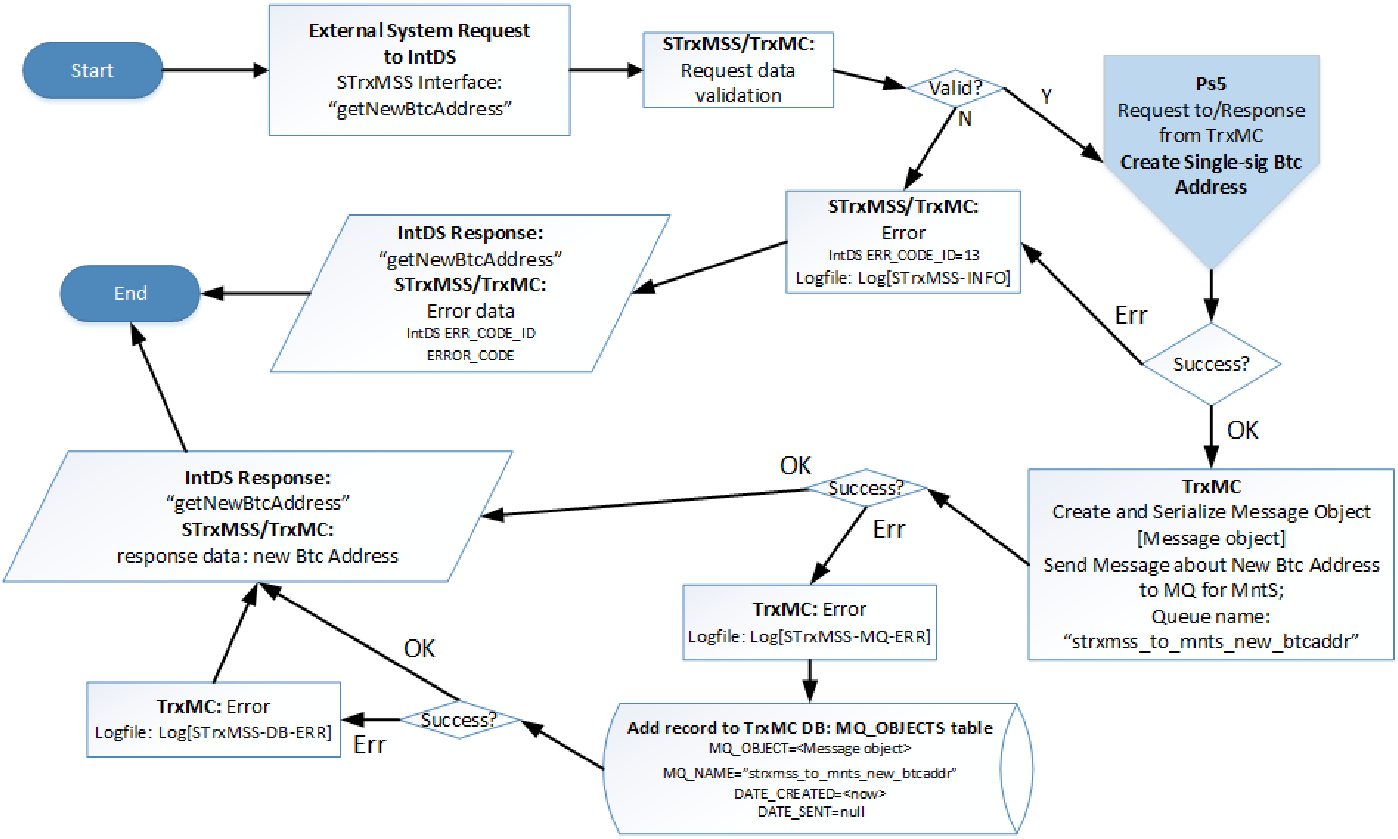
## **Get Wallets Balances:** “getWalletBalances” and “allWalletBalances” function



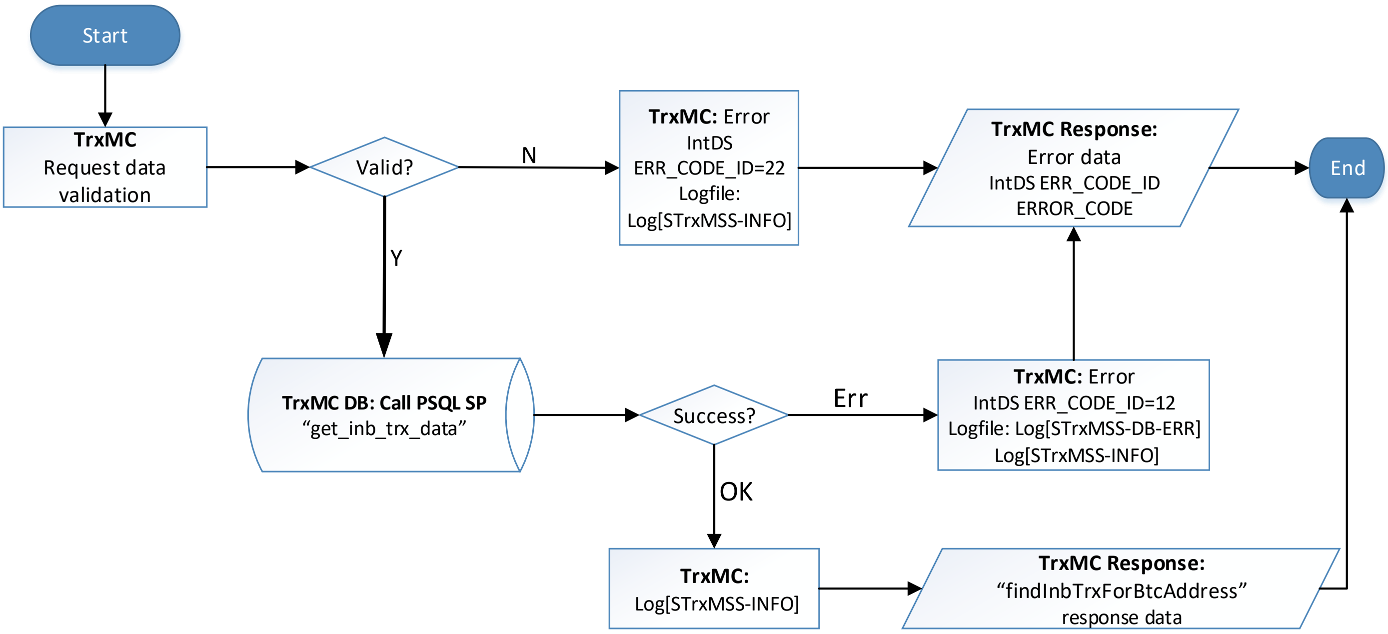
### 4.1.3 Inbound Transactions Functions Workflows

**Diagram Ps8. Create new Btc Address**: "getNewBtcAddress" function

Diagram Ps5 is involved in this process from "Outbound Transaction Workflow" point.

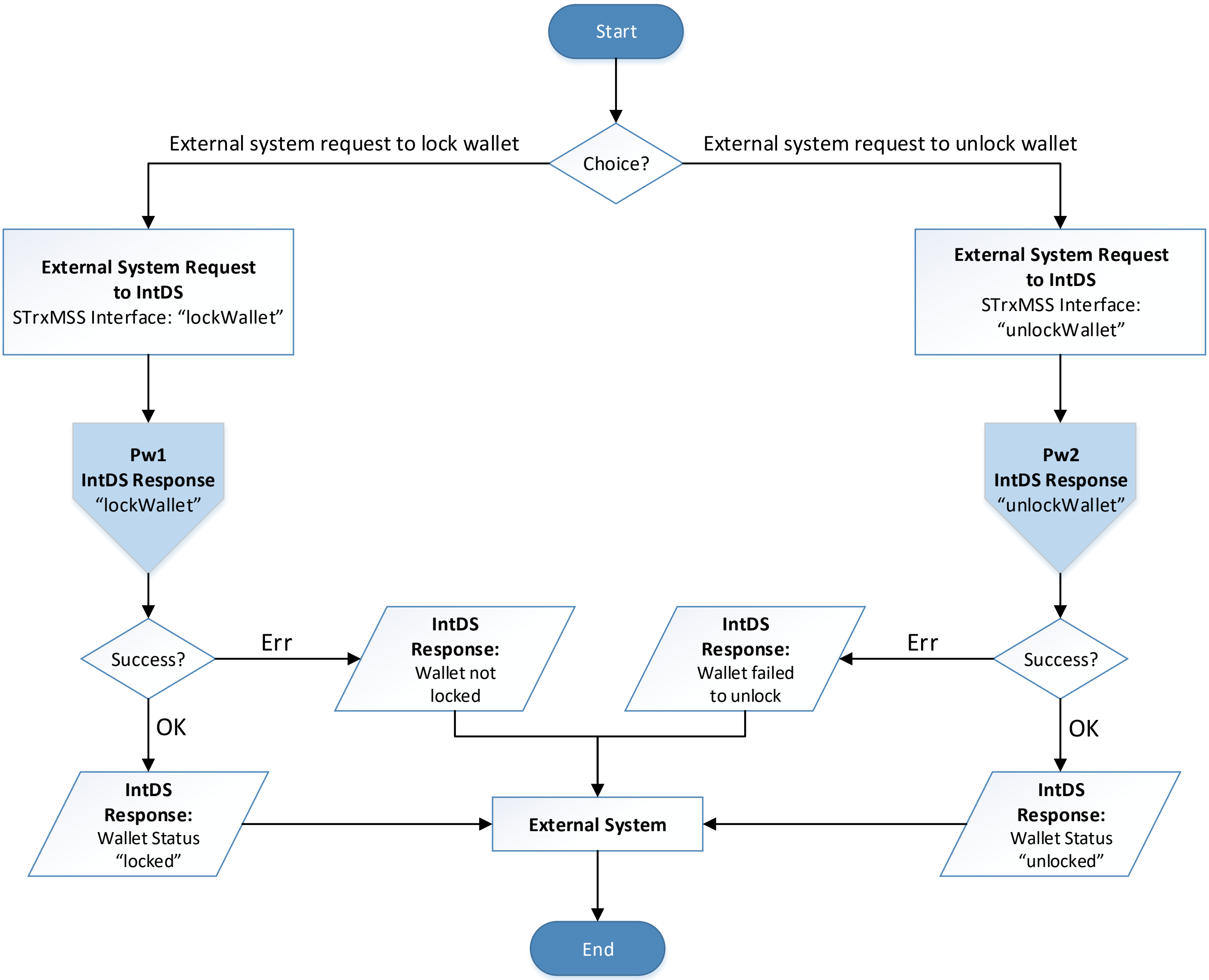


### **Find All Inbound Transactions for Given Bitcoin Address:** “findInbTrxForBtcAddress” function

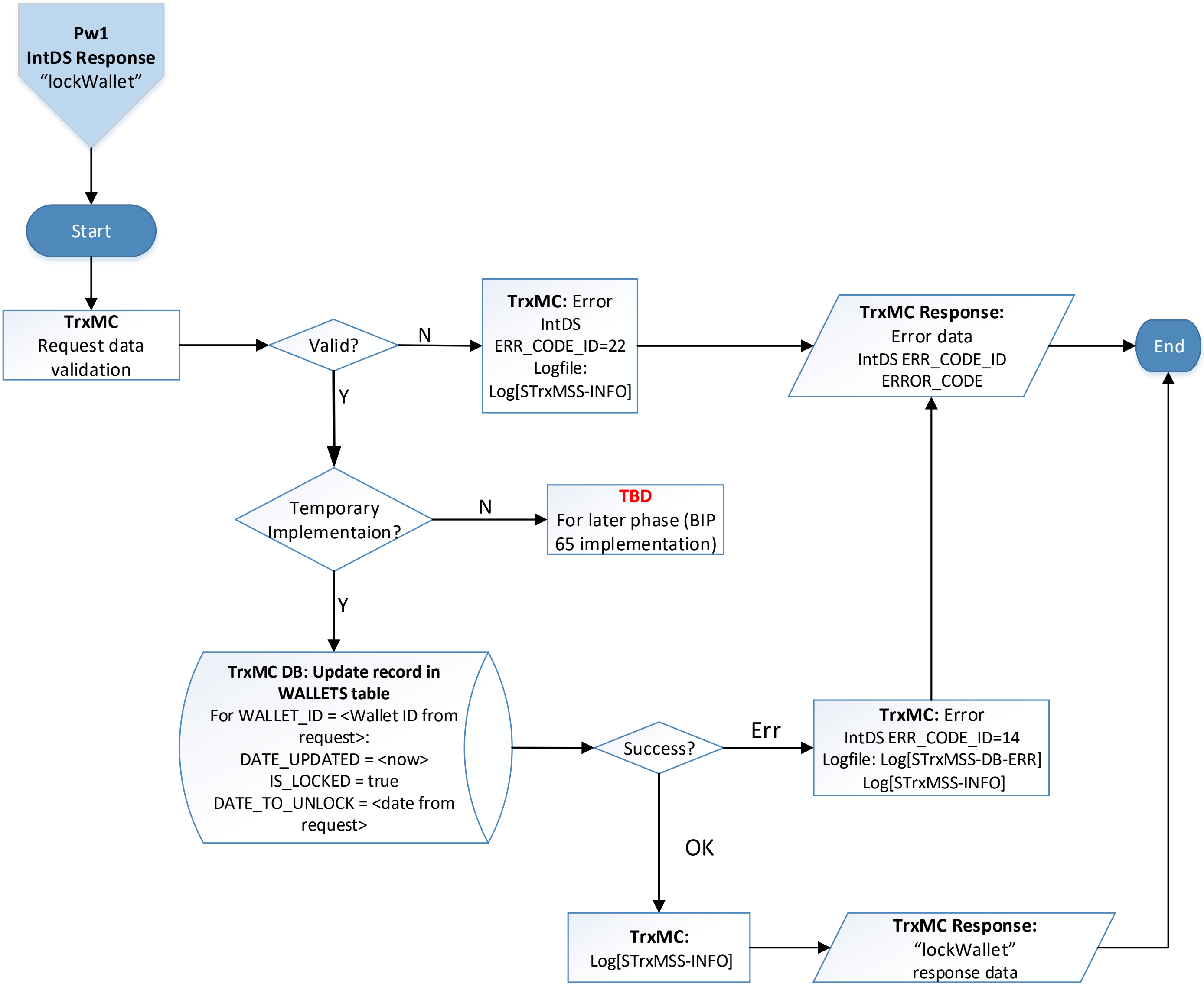


### 4.1.4 Warm Storage Functions Workflows

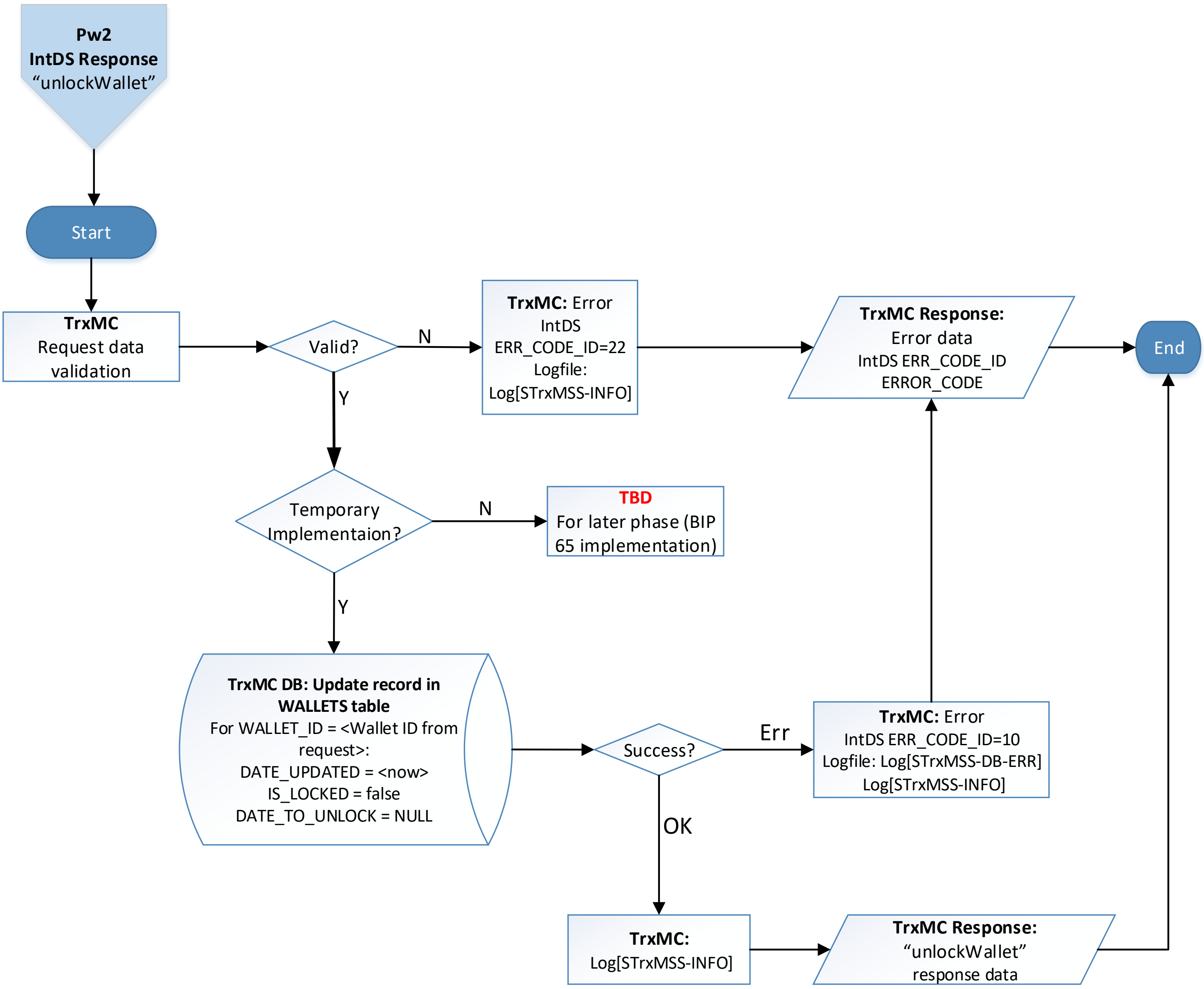
**Diagram Pw0. High Level Diagram. Lock and Unlock Wallet Processes:**



## **Diagram Pw1. Lock Wallet:** “lockWallet” function

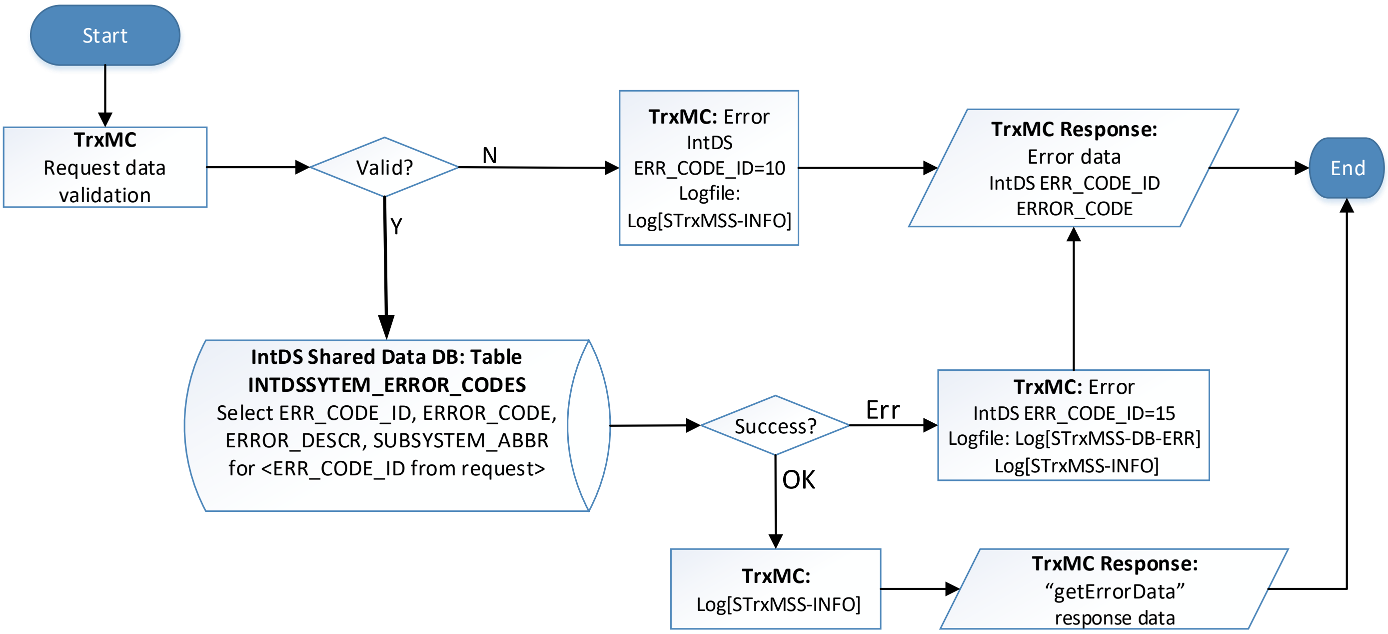


### **Diagram Pw2. Unlock Wallet:** "unlockWallet" function

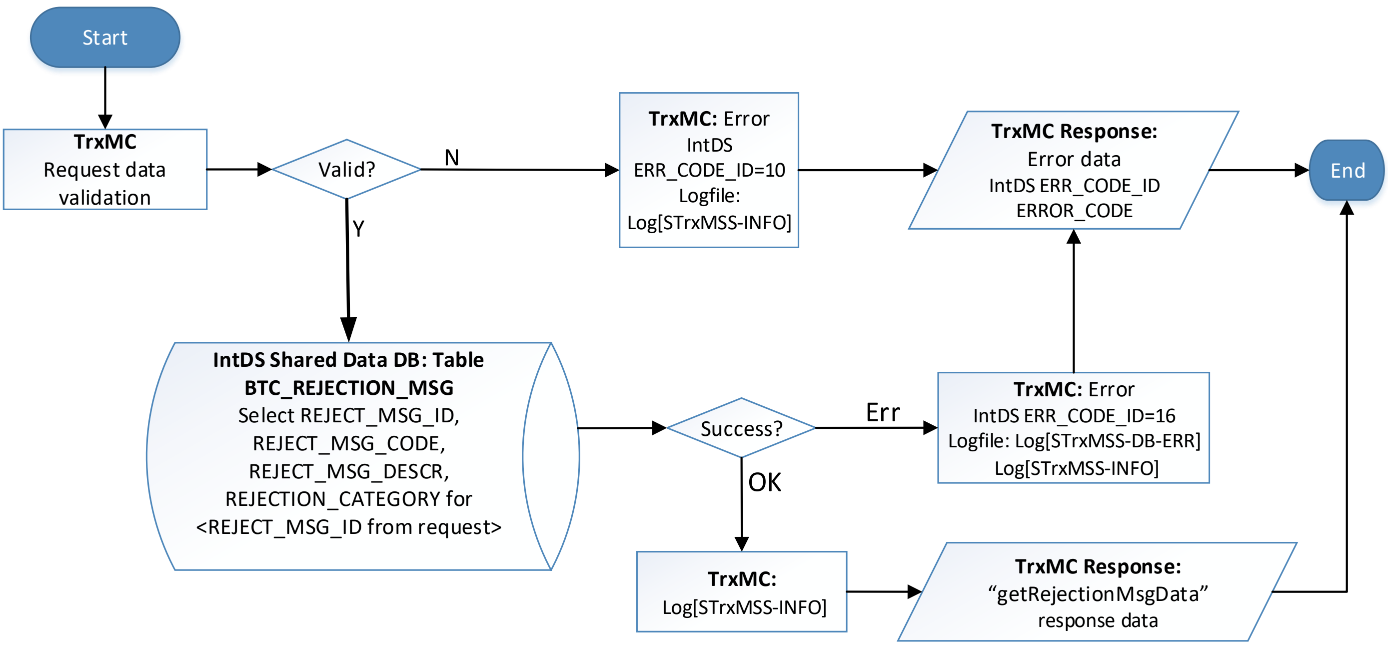


### 4.1.5 Other Functions Workflows

#### **Get data associated with particular error id:** "getErrorData" function




#### **Get data associated with particular rejection message:** "getRejectionMsgData" function



### 4.1.6 STrxMSS MQ Consumers and Producers Workflows

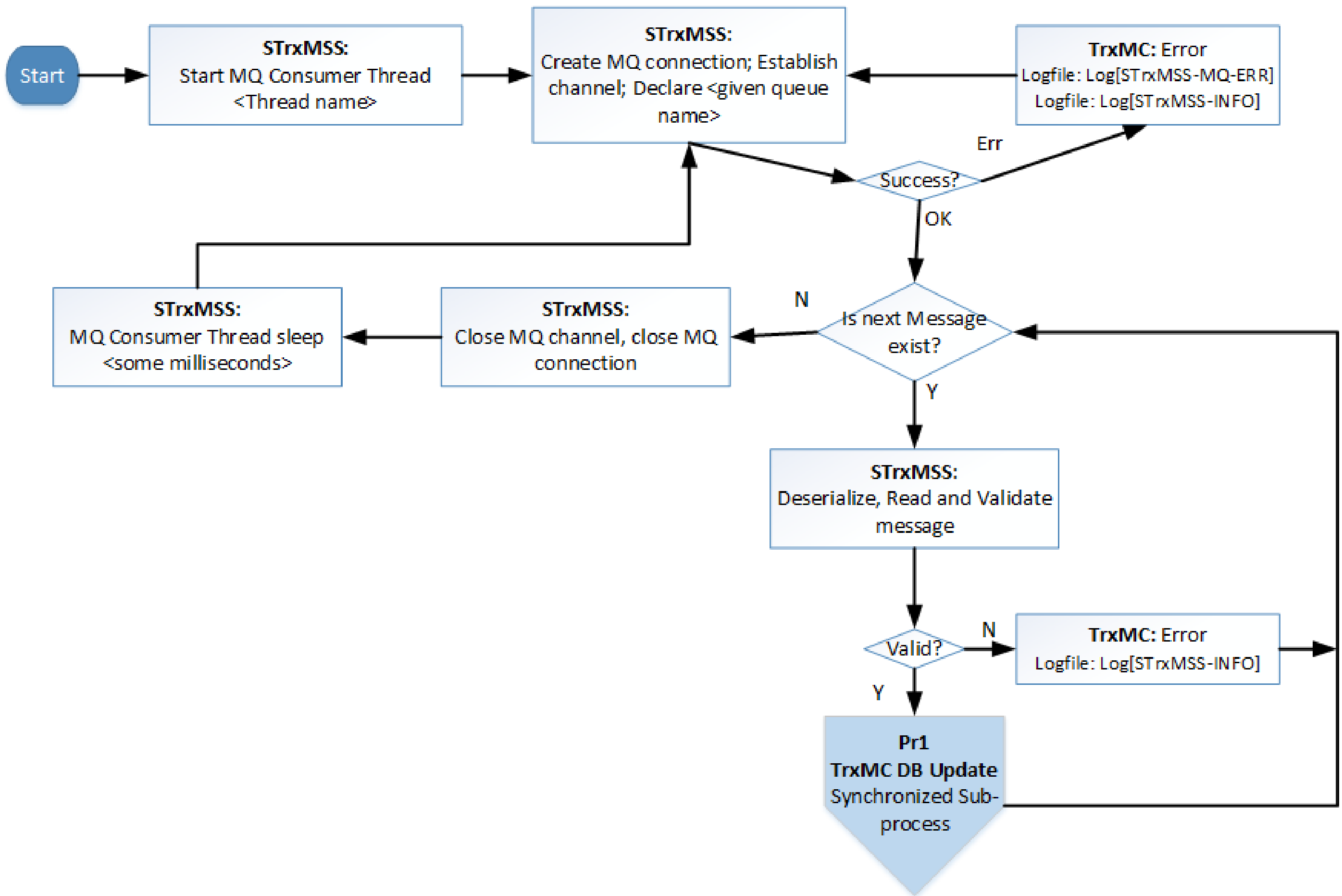
1. StrxMSS has three Consumers Threads:

- "InbTrxsDataThread" is consumer of messages from "mnts_to_strxmss_inb_trxs" queue
- "OutbTrxsDataThread" is consumer of messages from "mnts_to_strxmss_outb_trxs" queue
- "RejectMsgDataThread" is consumer of messages from "mnts_to_strxmss_reject_msg" queue

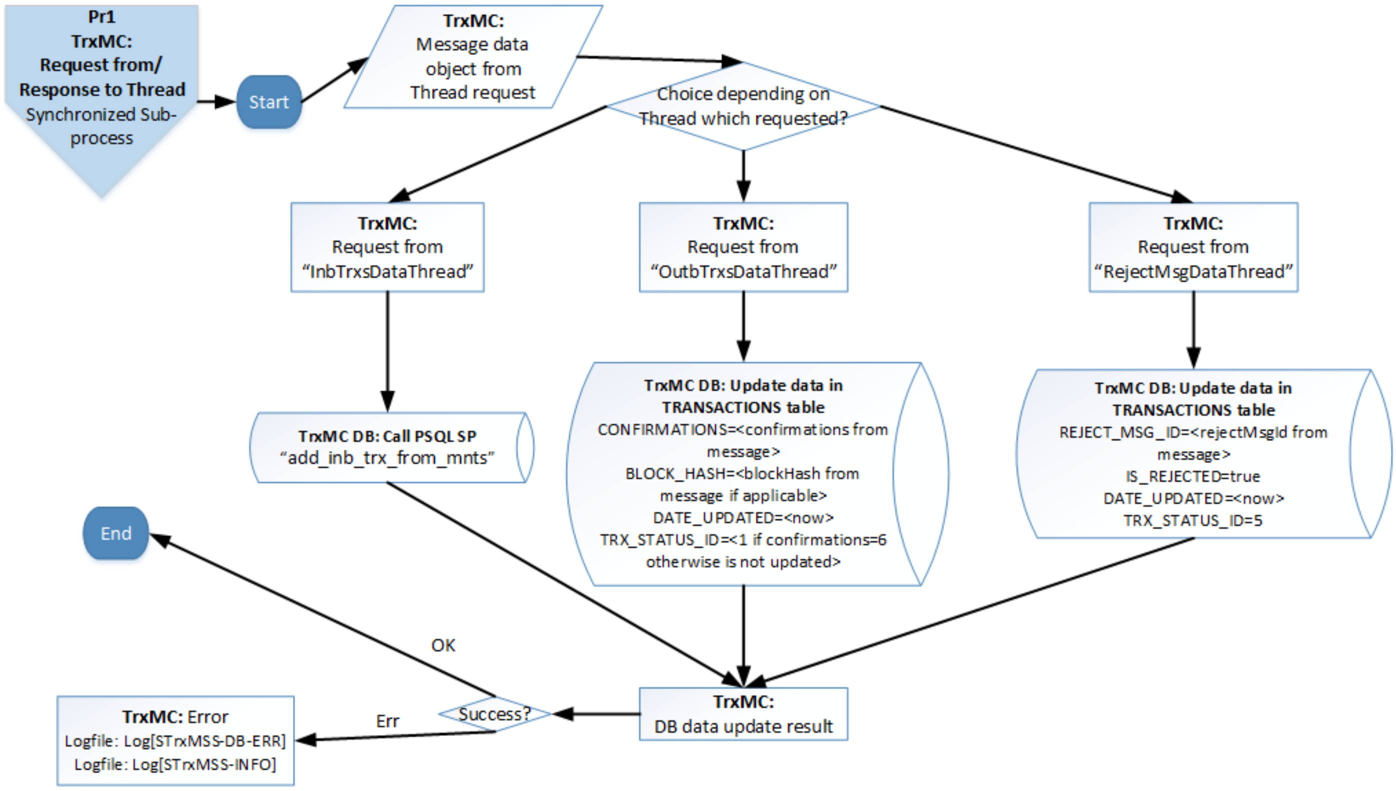
All threads are running in parallel independently according to the same logic. Diagram Pr0 shows general logic of Consumer thread. There is Sub-process Pr1 which should be synchronized, because each thread is updating the same DB table. See Diagram Pr1.

2. StrxMSS has three Producers. There are two simple Producers (see Ps3 and Ps8 diagrams) and one Producer Thread "DelayedMsgThread". There are some Delayed Messages which are stored into STrxMSS DB if simple Producer cannot send them. Producer Thread receives binary objects of messages from DB and sends them to MQ Exchange. Exchange routes and distributes messages between Queues. Diagram Pm0 shows "DelayedMsgThread" logic.

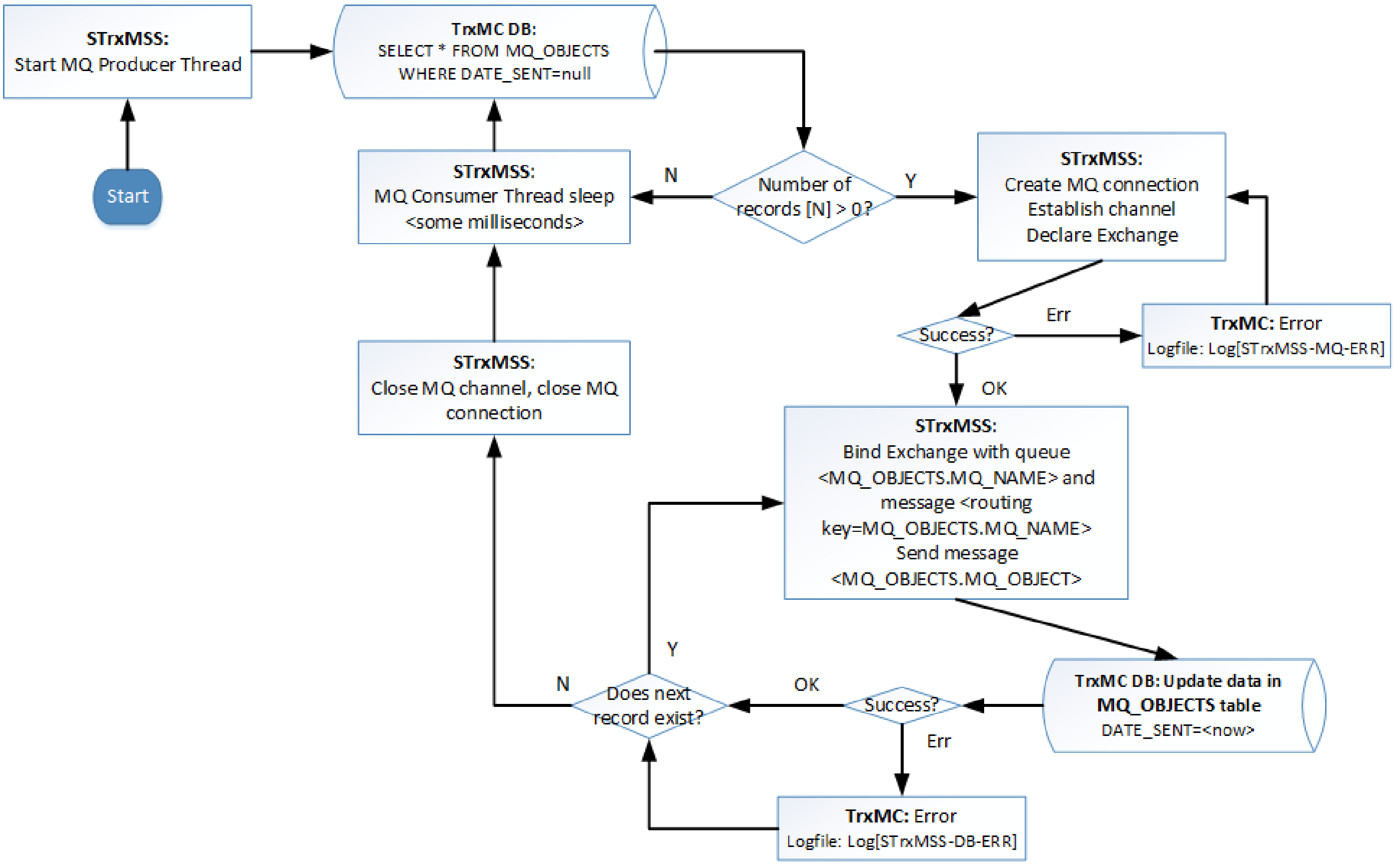
**Diagram Pr0. Consumer thread workflow:**



## Diagram Pr1. Save data from message in the STrxMSS DB



### Diagram Pm0. “DelayedMsgThread” Producer Thread:




## 4.2 Accounting Transaction Management SubSystem Workflows

This point can be done in the scope of future development. Will need some researching activity.

## 4.3 Bank Transaction Management SubSystem Workflows

This point can be done in the scope of future development. Will need some researching activity.

## 4.4 Exchange Transaction Management SubSystem Workflows

This point can be done in the scope of future development. Will need some researching activity.

## 4.5 Message Transaction Management SubSystem Workflows

This point can be done in the scope of future development. Will need some researching activity.

## 4.6 Contracts Management SubSystem Workflows

This point can be done in the scope of future development. Will need some researching activity.

## 4.7 Monitoring System Workflows

Monitoring system (MntS) is in charge of monitoring following items and sending appropriate messages to the MQ:

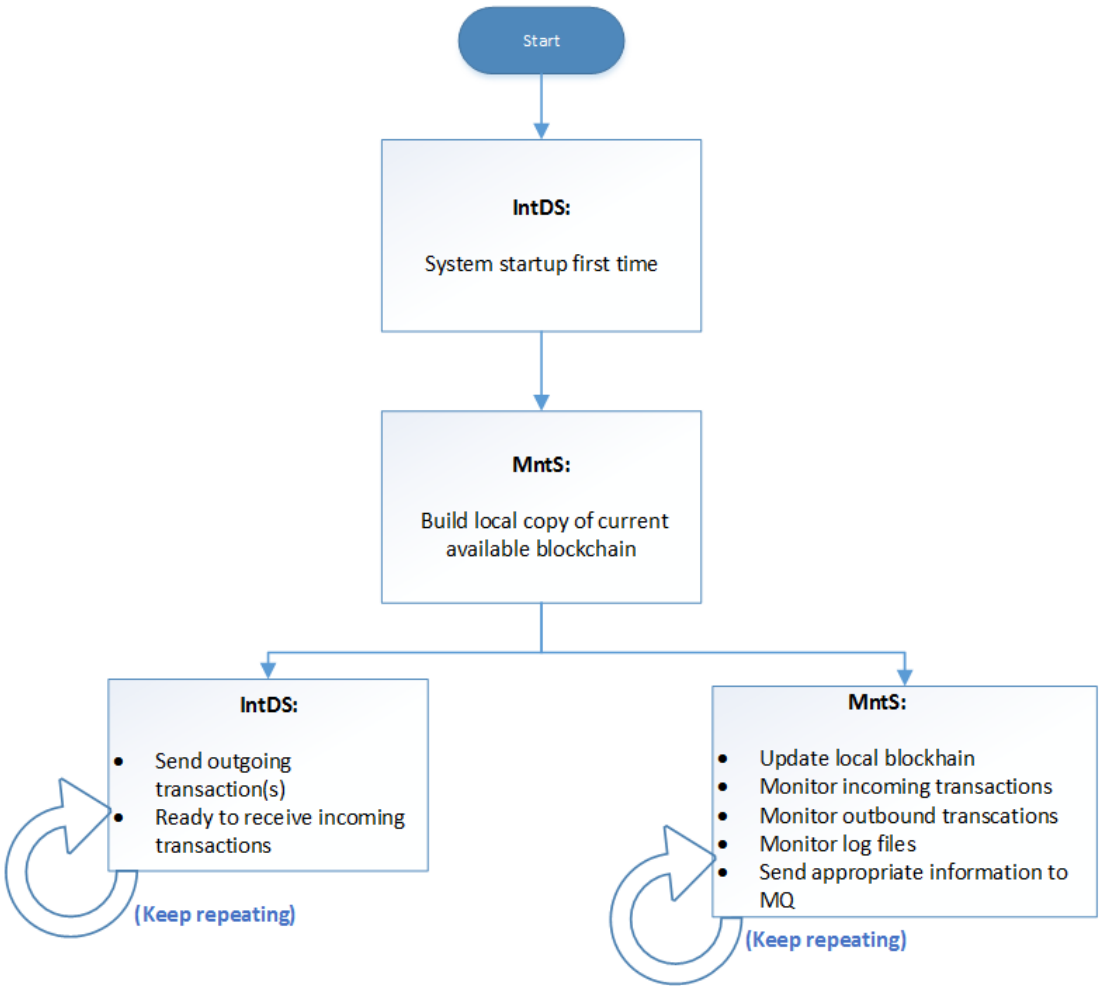
- the blockchain
- incoming transactions
- outgoing transactions from the system.

Refer section 3.8 for details of the database used by MntS.

MntS tasks can be summarized as follows:

- Download blockchain data into local files
- Store block data in MntS database
- Periodically update the database to simulate the blockchain i.e. Build local copy of the blockchain
- Keep monitoring for incoming and outgoing transactions
- Keep monitoring log files.

Following flow diagram summarizes these various monitoring tasks of MntS:




Following subsections describe each of these monitoring tasks in detail. Each task is accompanied with respective work flow diagram(s) and corresponding description.

MntS flow diagrams are all interconnected. Hence off-page connectors ( are used to connect all these diagrams together.

*Note: The flow diagrams cover the overall flow of each task. Detailed error handling, logging and generic conditions are not covered. Developers are expected to implement these by default.*

### 4.7.1 Build local blockchain (system start up for first time)

IntDS needs to store block information from the main blockchain locally. In order to do this, MntS needs to store required information in MntS database when system starts for the first time. This is a one-time process and will be implemented only when the system starts for the first time. This task will be undertaken by the MntS thread '***Sys. Start Thread***'.

For all times after that, only the latest block(s) will be downloaded and analysed.

Details of each step:

**Download blockchain into local files**

When the system starts for the first time, entire blockchain will be downloaded into local files. To get a copy of the blockchain locally, it is necessary to run the official bitcoin blockchain application called ‘Bitcoin-QT’. Bitcoin-QT stores the blockchain information in a series of .dat files.

The raw blockchain data files are stored in the following locations on the hard drive:

Linux: ~/.bitcoin/blocks

MacOS: ~/Library/Application Support/Bitcoin/blocks

Windows: %APPDATA%Bitcoin\blocks

WinXP: C:\Documents and Settings\YourUserName\Application data\Bitcoin\blocks

Win7/Win8/Vista: C:\Users\YourUserName\AppData\Roaming\Bitcoin\blocks

They will appear as a series of 128mb files blk00000.dat through blk00???.dat.

Each blk00*.dat file is a collection of several raw blocks. Refer section 9.1 for format of a block.

**Store downloaded block data in MntS database**

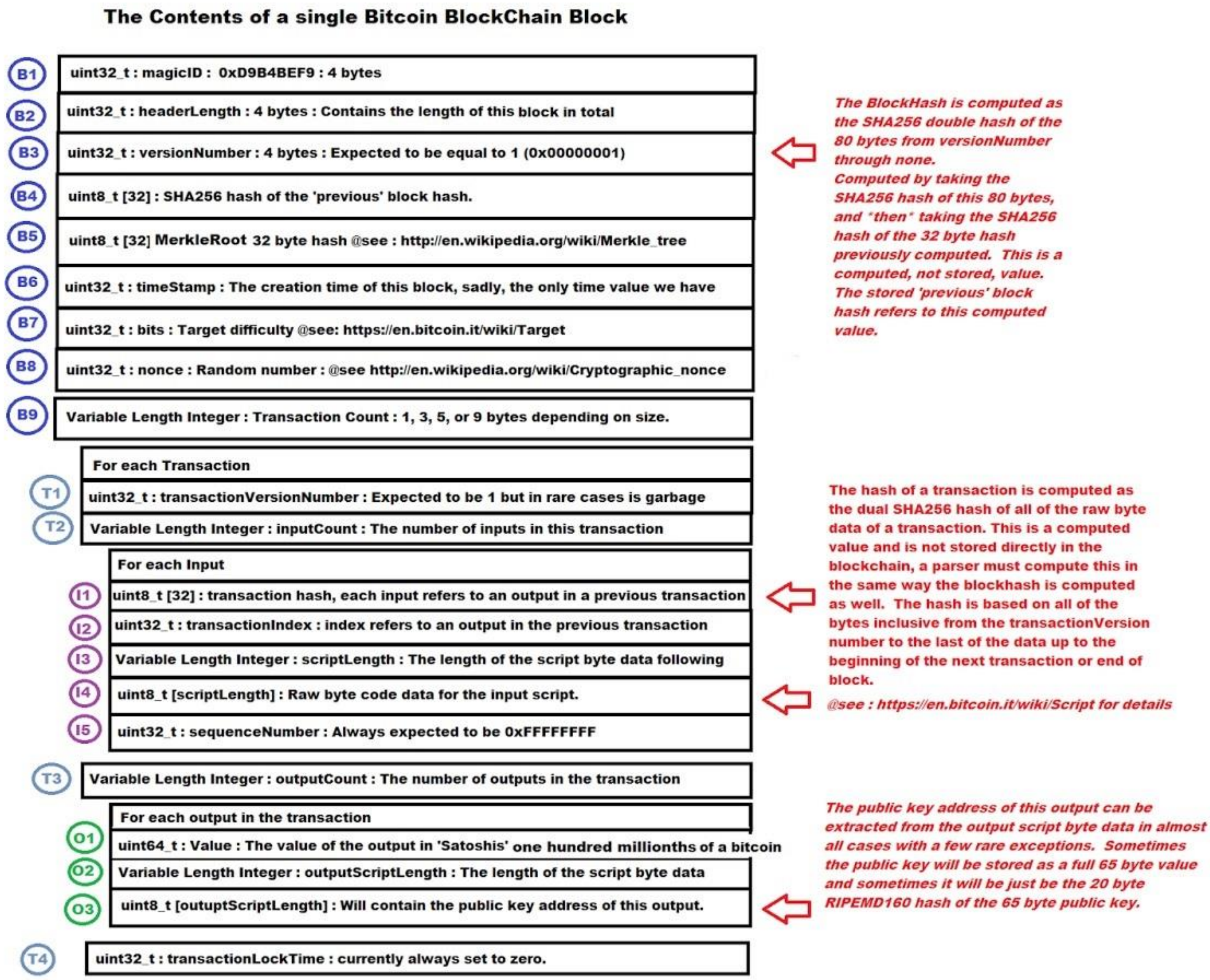
Once .dat files have been downloaded, MntS can start parsing the block data and updating the BLOCKS table from the monitoring system database.

The BLOCKS table has following columns:

BLOCK_INDEX, BLOCK_HASH, BLOCK_HEIGHT, PREV_BLOCK_HASH, BLOCK_TIMESTAMP, DATE_CREATED_TIMESTAMP

Hence, we need to extract all the relevant information from each .dat file and store in respective fields in BLOCKS table.

Block structure can be summarized as follows:




(Reference: [2.26])

1. Sys. Start Thread will scan through each downloaded .dat file and process block data.
2. The blocks are separated by a block separator (known as ‘magic id’).
3. Once a block has been identified (based on occurrence of magic id), get the bytes that form blockheader.
4. For every block, identify the blockheader and compute corresponding blockhash. Blockheader consists of the 80 bytes from version number to nonce. BlockHash is computed as the SHA256 double hash of the blockheader.

Following is the breakdown of Blockheader at byte level. Refer section 9.1 for a detailed explanation.

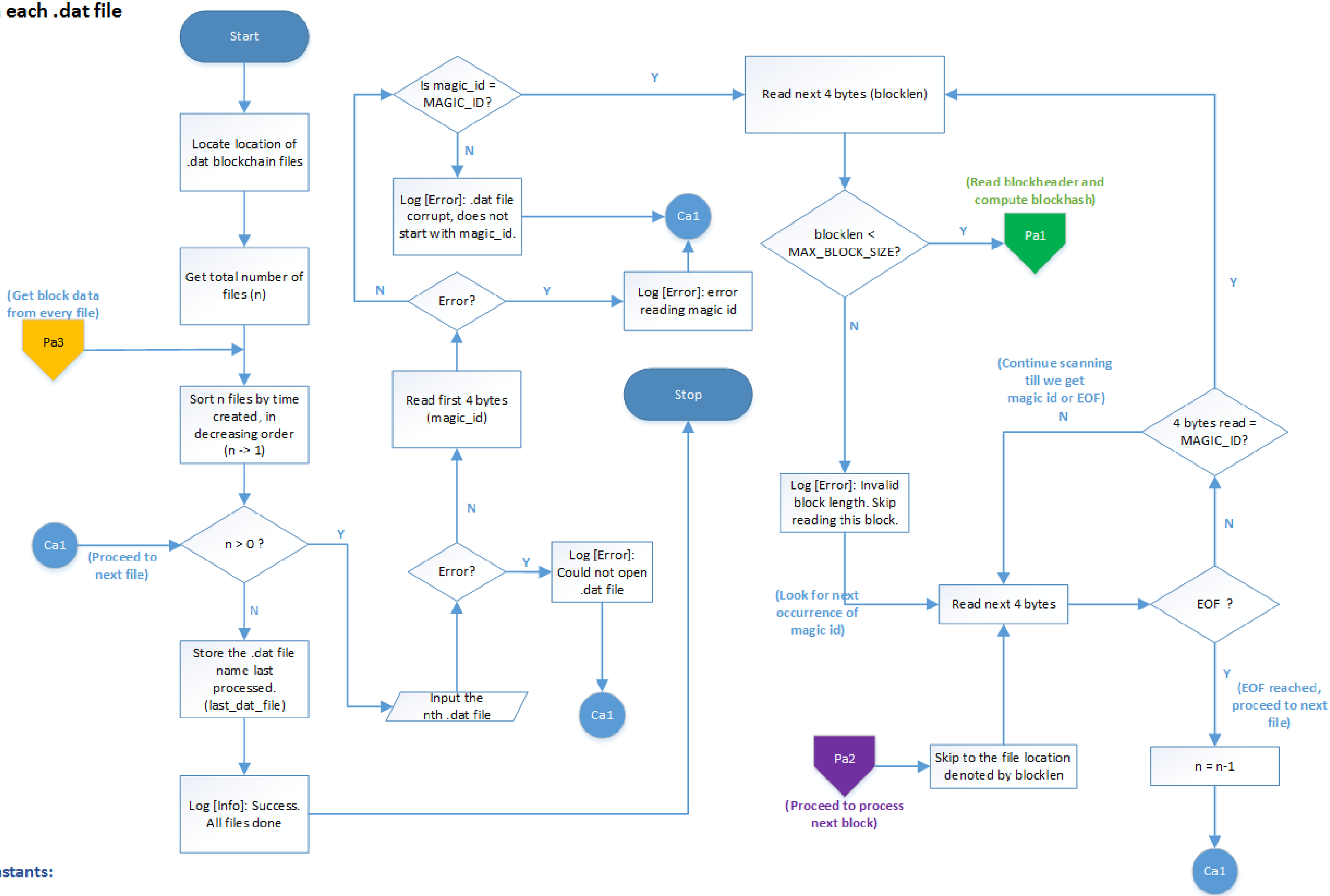
| **Blockheader part** | **Length in Bytes** | **Byte location in the Blockheader (starting at 0)** |
|---|---|---|
| Version number | 4 | 0-3 |
| Previous Block Hash | 32 | 4-35 |
| Merkle Root Hash | 32 | 36-67 |
| Timestamp | 4 | 68-71 |
| Target Difficulty | 4 | 72-75 |
| Nonce | 4 | 76-79 |

5. Store blockhash, previous blockhash and block timestamp in BLOCKS table.
6. Constants used during this processing:

MAGIC_ID = 0xD9B4BEF9

MAX_BLOCK_SIZE = 1 MB

**Scan each .dat file**



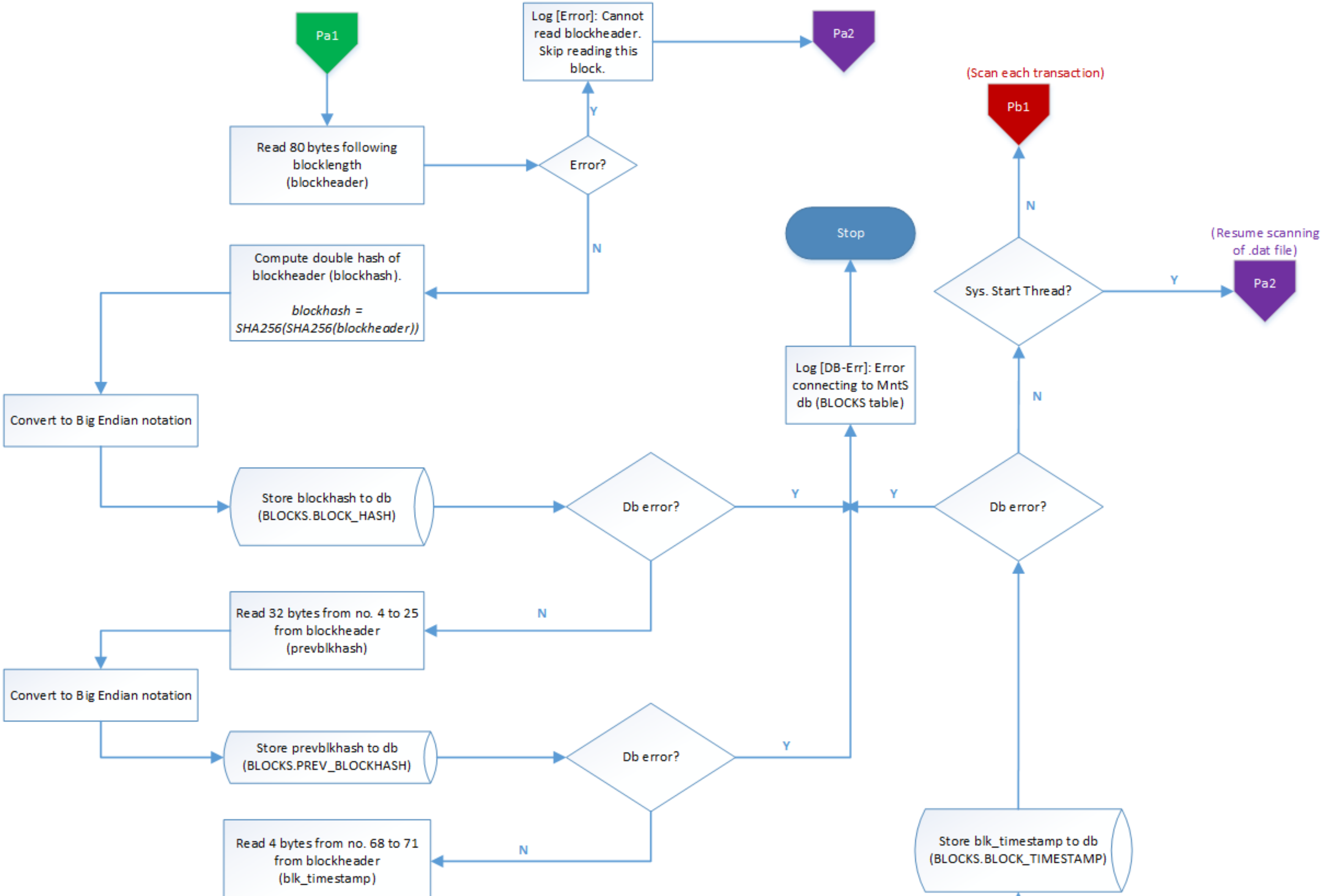

**Constants:**

MAGIC_ID = 0xD9B4BEF9

MAX_BLOCK_SIZE = 1 MB

7. Once blockheader is read, skip to the file location denoted by blocklen value.
8. This is because we are storing only the block data at this stage. Remaining bytes consist of transaction data for this particular block. Hence we can skip through this portion of the block and proceed to next block once blockheader is processed.
9. Scan for next magic id occurrence and repeat the processing of blockheader for each block. If EOF is reached, open next file for processing.

**Read Blockheader and compute the blockhash for each block**



**Build local copy of blockchain**

After the above steps are done, the BLOCKS table will contain block data from all the downloaded dat files.

Next, the BLOCKS table will be sorted such that it resembles the blockchain. Starting with the most recent block in the blockchain, genesis block will be the bottom-most block in the BLOCKS table.

1. Find genesis block in the table (Genesis Block hash = 000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f) and assign corresponding BLOCK_HEIGHT field as 0.
2. Find next block such that block hash of previous block found matches with value in PREV_BLOCK_HASH field of this block. Increment corresponding BLOCK_HEIGHT value by 1.
3. Repeat this process till the last block downloaded (i.e. there will be no block with PREV_BLOCK_HASH field same as this block’s hash).

Note: There can be more than 1 block with same height and same parent (PREV_BLOCK_HASH field). Only one of these blocks will be part of the main blockchain. The other block will become “orphan block”.

Everytime MntS downloads latest block(s), they will be added to the top of the table with PREV_BLOCK_HASH field matching with previous hash of block.

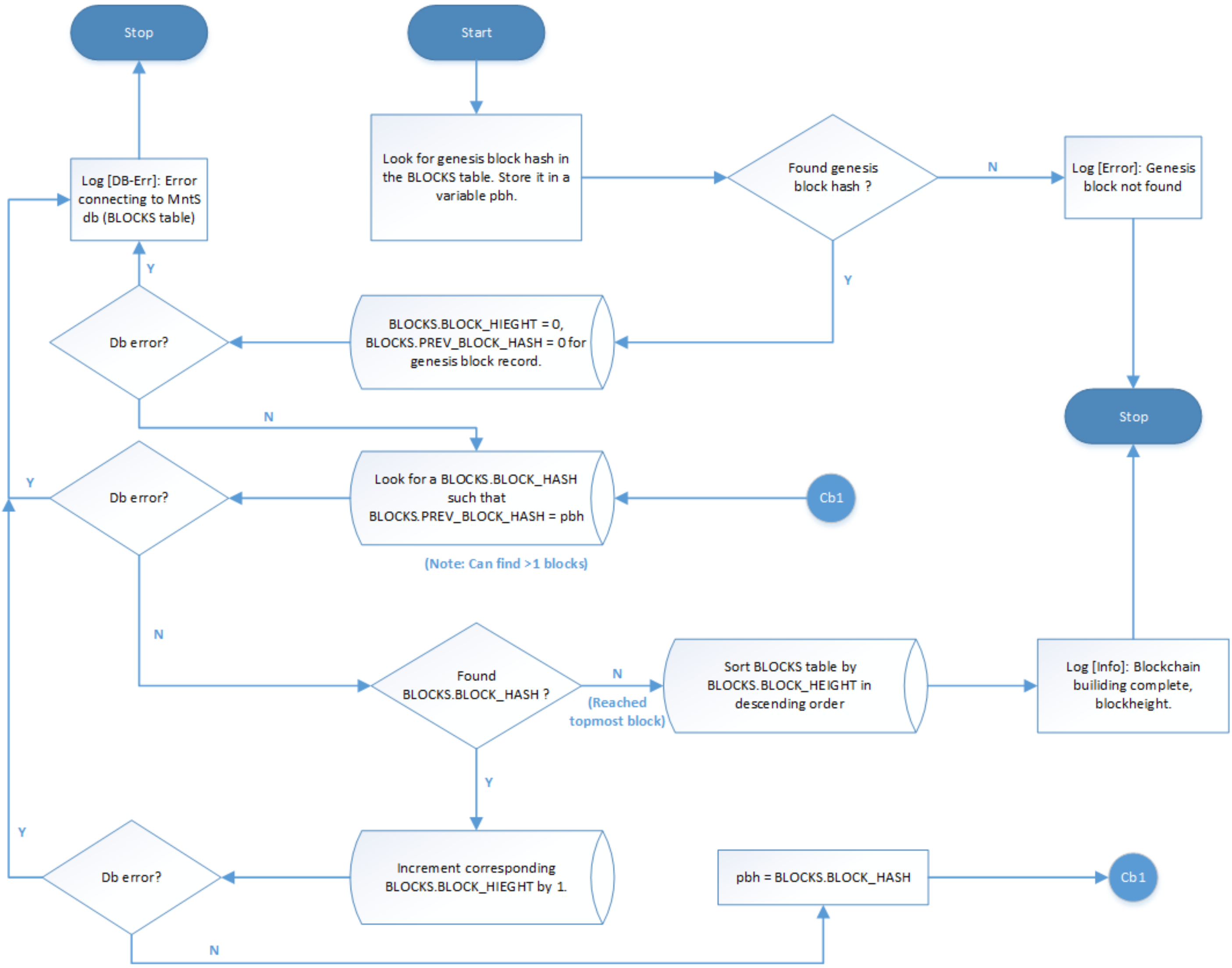
Constants:

Genesis Block hash = 000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f

**Update BLOCKS table to simulate the blockchain**

**Constant:**

Genesis Block hash = 000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f

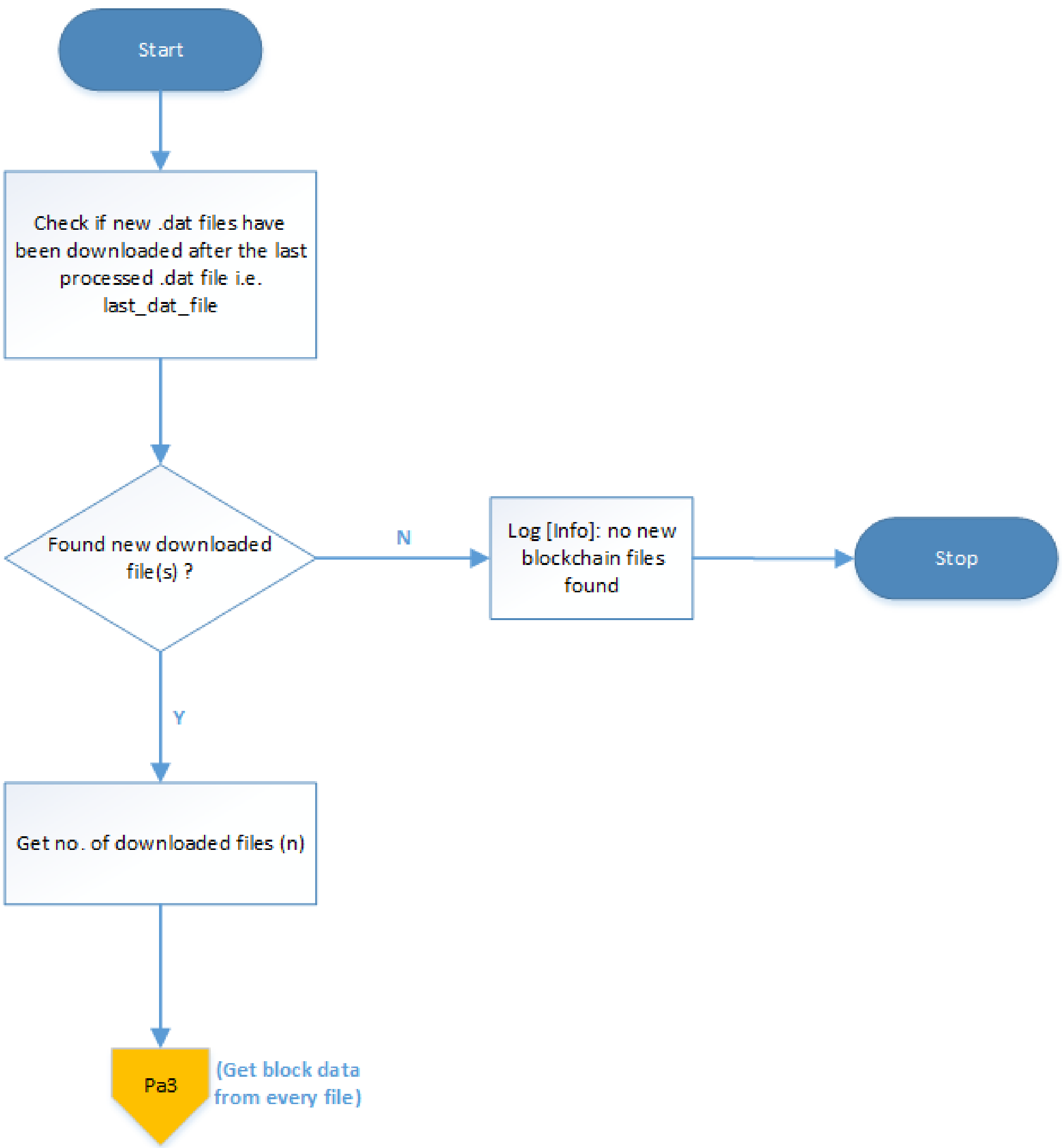


### 4.7.2 Update local blockchain and scan transaction data

The MntS thread (**Sync. Thread**) will sync with the main blockchain periodically (every 3 minutes). The BLOCKS table will be updated if new block(s) is downloaded.

MntS will store last .dat file it had processed. Sync. Thread will pick up .dat(s) file created after the last processed file. Blocks from the new .dat file(s) will be scanned and relevant information will be updated in the BLOCKS table.

**Update BLOCKS table with newly downloaded blocks.**



Unlike building blockchain at system startup, we will need to analyse transaction data of each block while updating the blockchain. This is because we need to check for incoming and outgoing transactions in every block. Hence, after processing and storing blockheader information, transaction data will be scanned for every downloaded block.

Each transaction will be checked if it is an incoming transaction or not. The Inc. Thread will be invoked ONLY if transaction is found to be incoming.

However, Out. Thread will be invoked for every transaction.

Each transaction will be checked in the following sequence:

- Check if it is an incoming transaction.

To identify incoming transactions, MntS will check if value of destination address in any of the outputs is same as any of the btc addresses generated by IntDS. Btc addresses generated by IntDS will be stored in STRXMSS_MONITORED_BTC_ADDR table.

- Calculate transaction hash.
- Invoke Inc. Thread & Out. Thread.
    - Inc. Thread will be launched to monitor this transaction only if the transaction is found to be incoming.

Incoming transactions will be monitored for:

- confirmations (till it reaches a value equal to greater than 6)
- BTC amount received (till it reaches a value greater than or equal to expected BTC amount)

    - Invoke Out. Thread.

Out. Thread will check transaction hash of every transaction that is included in the block and compare it with the transaction hashes that were generated by IntDS (stored in table STRXMSS_MONITORED_TRXS). If match is found, it means that a transaction sent by IntDS was included in this block. If match is not found, Out. Thread will check if the transaction has been sitting in the mempool for a week or more.

Outbound transactions will be monitored for:

- transaction stuck in mempool for long (not included in the blockchain for a week or more after transaction creation)
- confirmations (till it reaches a value equal to greater than 6)

**Incoming transactions** to the system are the transactions which have destination address that matches with the BTC_ADDRESS field in the STRXMSS_MONITORED_BTC_ADDR table. MntS will fill all fields in STRXMSS_MONITORED_BTC_ADDR table (except BTC_AMOUNT_PAID) with appropriate values received from MQ.

In order to check if the transaction is incoming, MntS will check if btc address in the scriptPubKey of every output matches with any value of BTC_ADDRESS field in the STRXMSS_MONITORED_BTC_ADDR table. Detailed steps of this process are as follows:

For every transaction in newly downloaded block(s):

1. Compare destination address in every output of the transaction with all values in the BTC_ADDRESS field from STRXMSS_MONITORED_BTC_ADDR table.

    a. To get to the outputs of a transaction, we need to go through all the inputs first, since the output count is located after the last input.

    b. Once we get to the output count, go through each output and decode the ScriptPubKey to get the destination address. Compare this address with every value in the BTC_ADDRESS field from STRXMSS_MONITORED_BTC_ADDR table. (Refer: Subproc Decode op_script).

    c. If match found: check if the value in this output is equal to the expected amount.

    - If yes, move this transaction record from STRXMSS_MONITORED_TRXS to ARCHIVE.

    - If value is less than expected amount, update the BTC_AMOUNT_PAID field with actual value received. Do not move the record to ARCHIVE, so that MntS will continue monitoring this address.

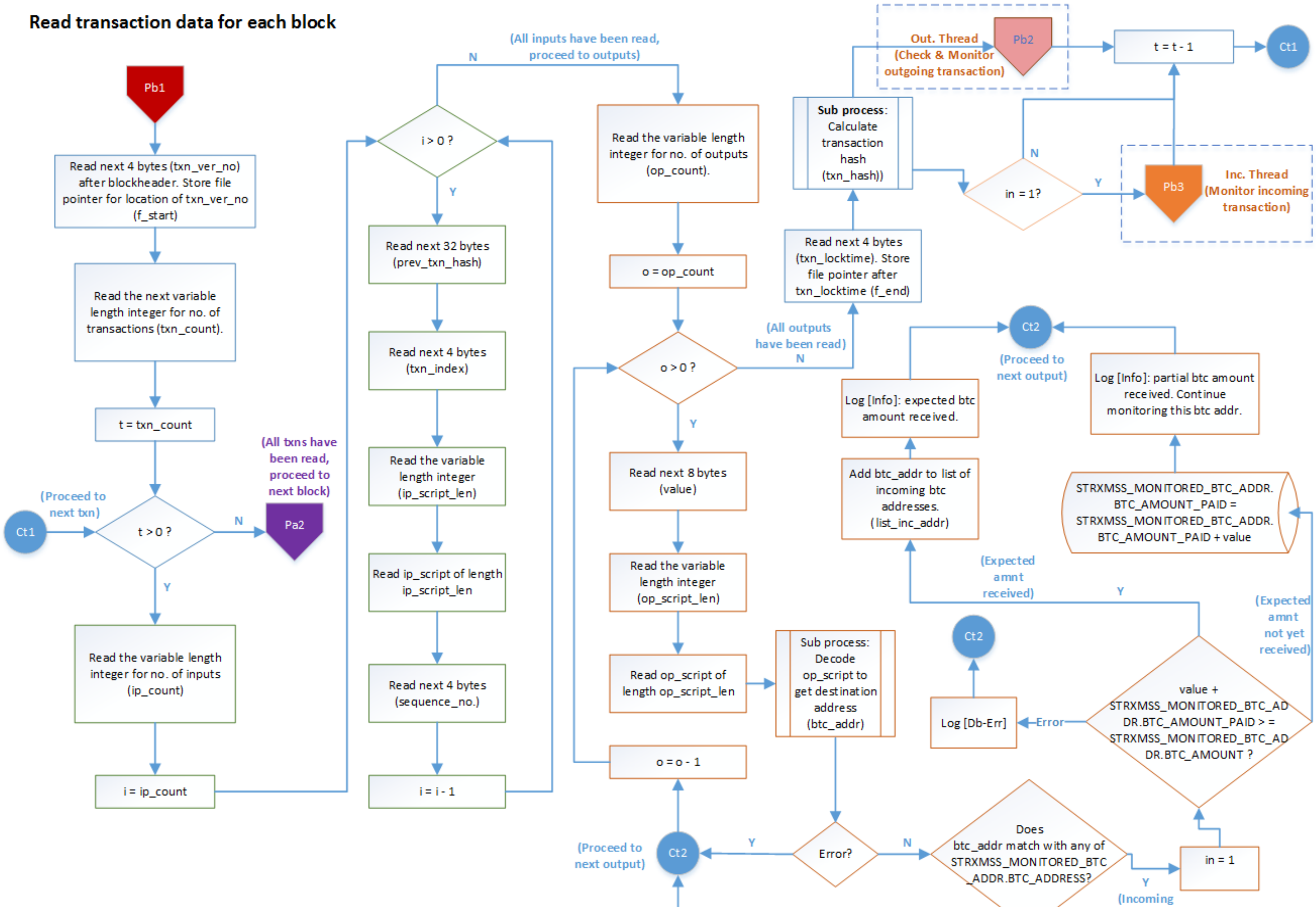
    - Identify this transaction as an Incoming transaction.

    - Store the btc address identified as the destination address alongwith value 126ubscrip (in an appropriate data structure).

2. Once all outputs of a transaction have been scanned for destination address, calculate the transaction hash of this transaction. At this point, following threads will be invoked:

    - Inc. Thread will monitor confirmations if the transaction is identified as “Incoming’ in step 1c.

    - Out. Thread will check if the transaction is outbound and monitor confirmations if it is.

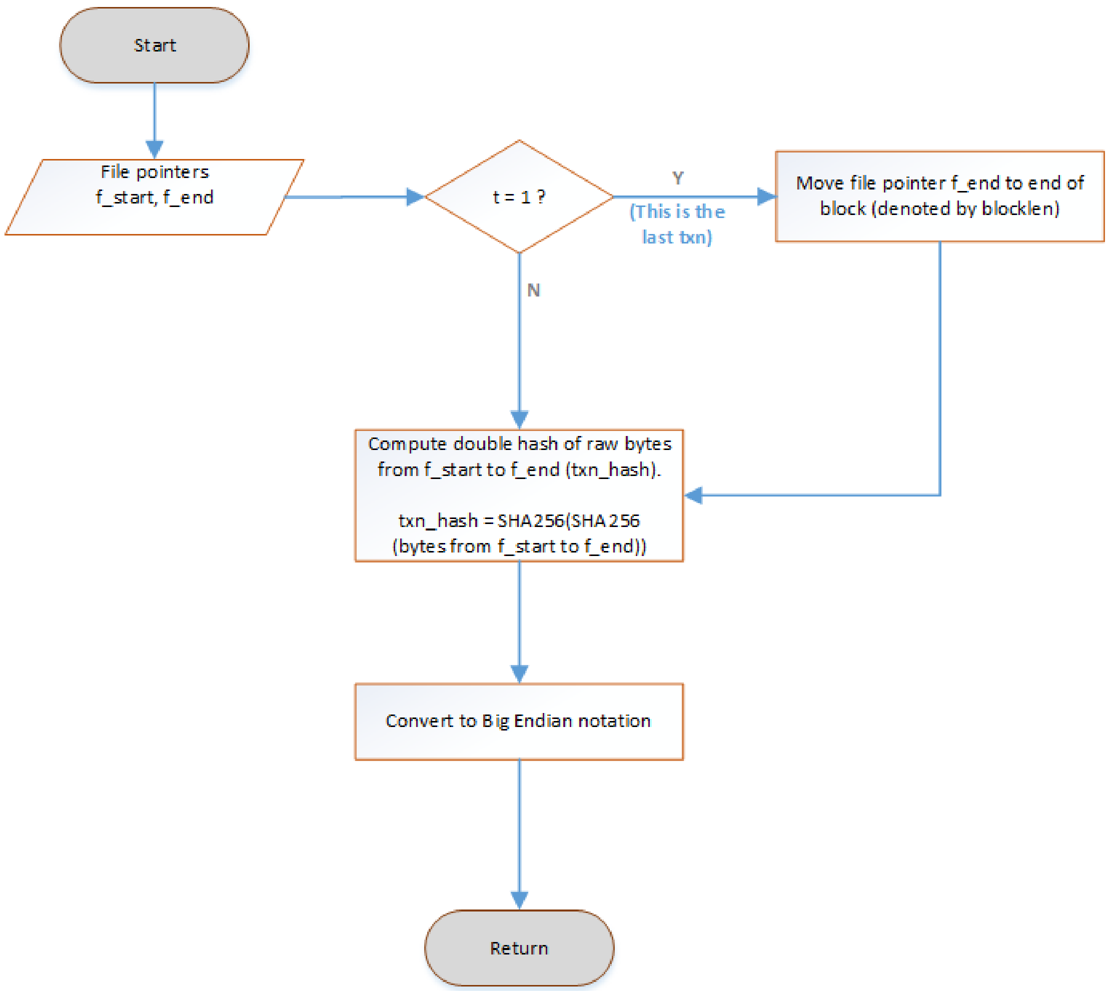

Read transaction data for each block
Pb1
Read next 4 bytes (txn_ver_no) after blockheader. Store file pointer for location of txn_ver_no (f_start)
Read the next variable length integer for no. of transactions (txn_count).
t = txn_count
(Proceed to next txn)
Ct1
t > 0 ?
N
(All txns have been read, proceed to next block)
Pa2
Y
Read the variable length integer for no. of inputs (ip_count)
i = ip_count
i > 0 ?
Y
Read next 32 bytes (prev_txn_hash)
Read next 4 bytes (txn_index)
Read the variable length integer (ip_script_len)
Read ip_script of length ip_script_len
Read next 4 bytes (sequence_no.)
i = i - 1
N
(All inputs have been read, proceed to outputs)
Read the variable length integer for no. of outputs (op_count).
o = op_count
o > 0 ?
Y
Read next 8 bytes (value)
Read the variable length integer (op_script_len)
Read op_script of length op_script_len
o = o - 1
(Proceed to next output)
Ct2
Sub process: Decode op_script to get destination address (btc_addr)
Error?
Y
N
Does btc_addr match with any of STRXMSS_MONITORED_BTC_ADDR.BTC_ADDRESS?
Y
(Incoming
in = 1
(All outputs have been read)
N
Read next 4 bytes (txn_locktime). Store file pointer after txn_locktime (f_end)
Sub process: Calculate transaction hash (txn_hash))
Out. Thread (Check & Monitor outgoing transaction)
Pb2
t = t - 1
Ct1
N
in = 1?
Y
Pb3
Inc. Thread (Monitor incoming transaction)
Ct2
(Proceed to next output)
Log [Info]: expected btc amount received.
Log [Info]: partial btc amount received. Continue monitoring this btc addr.
Add btc_addr to list of incoming btc addresses. (list_inc_addr)
STRXMSS_MONITORED_BTC_ADDR.BTC_AMOUNT_PAID = STRXMSS_MONITORED_BTC_ADDR.BTC_AMOUNT_PAID + value
(Expected amnt received)
Y
(Expected amnt not yet received)
N
value + STRXMSS_MONITORED_BTC_ADDR.BTC_AMOUNT_PAID >= STRXMSS_MONITORED_BTC_ADDR.BTC_AMOUNT ?
Error
Log [Db-Err]
Ct2

**Sub processes involved in above flow:**

**1. Subproc: Calculate transaction hash**

- Transaction hash is calculated by taking the double hash of all raw bytes from transaction version number of current transaction to the beginning of the next transaction or end of block.
- Since we have already read the transaction version in the main flow above, we have its file pointer location (f_start).
- We also have the file pointer after reading transaction locktime (f_end).
- Check if the transaction is the last transaction in the block.
  - If it is, go to the end of the block. Let the file pointer location be f_end.
  - If not, continue.
- Compute the double hash of all bytes from f_start to f_end. This is the transaction hash.
- Convert it to big endian form.

**Sub process: Calculate transaction hash (txn_hash)**




**Subproc: Decode op_script to get destination address**

- Since we deal with P2PKH addresses, the length of op_script should always be **25 bytes**.

Following is the breakdown of a P2PKH scriptPubKey at byte level. Refer section 8.1.2 for a detailed explanation.

| **Script part** | **Length in Bytes** | **Byte location in the scriptPubKey (starting at 0)** |
|---|---|---|
| OP_DUP (0X76) | 1 | 0 |
| OP_HASH (0XA9) | 1 | 1 |
| Length of PubKeyHash | 1 | 2 |

| | | |
|---|---|---|
| PubkeyHash | 20 | 3-22 |
| OP_EQUALVERIFY (0X88) | 1 | 23 |
| OP_CHECKSIG (0XAC) | 1 | 24 |

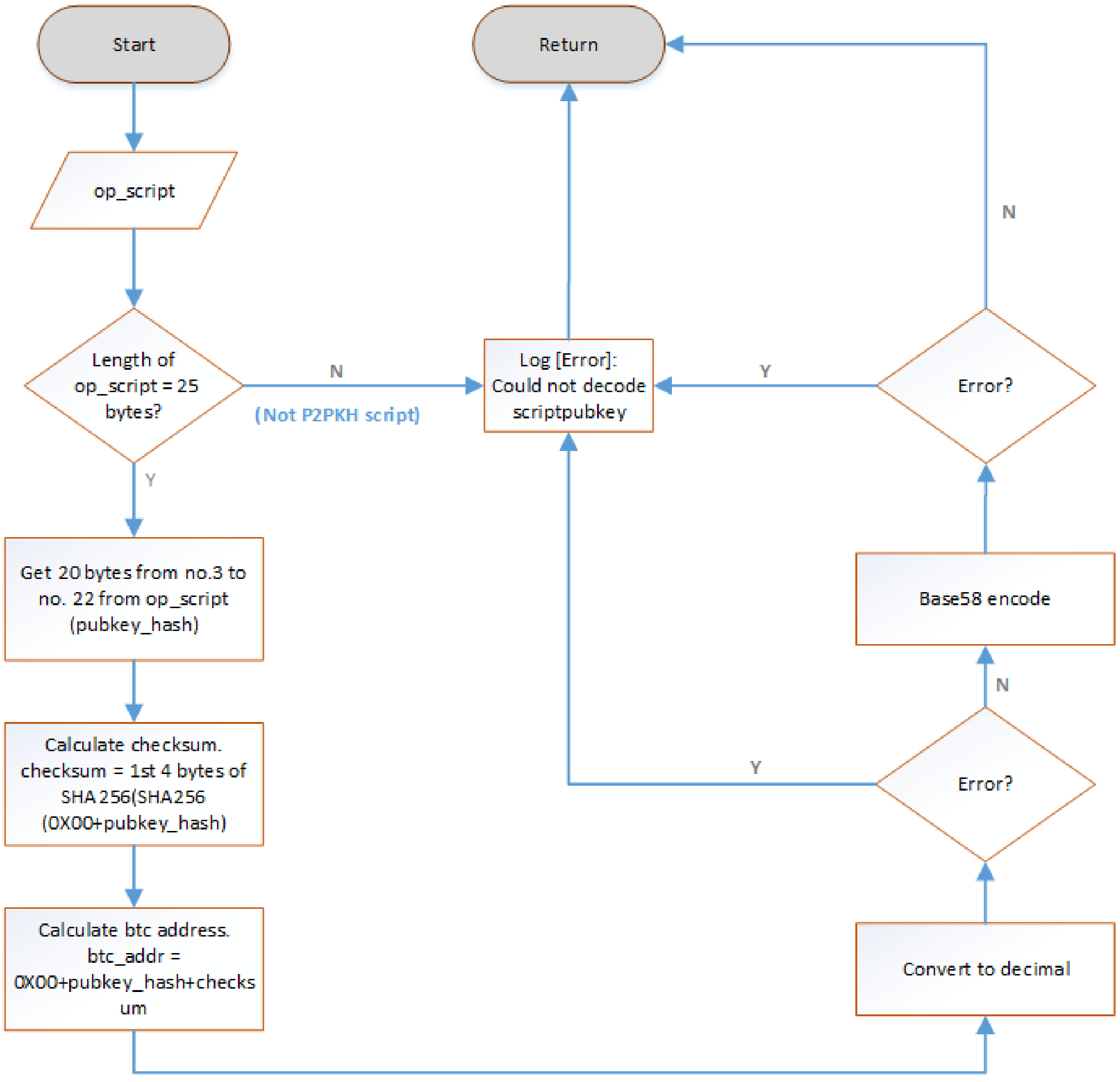
- The btc address can be extracted from a P2PKH op_script as follows:
  - Public key hash (pubkey_hash) = op_script [3] – op_script[22]
  - Checksum hash = Append 0X00 to pubkey_hash. Take double hash of the result.
  - Checksum (checksum) = First 4 bytes of the Checksum hash
  - btc address (btc_addr) = Concatenate 0x00 with pubkey_hash and checksum

i.e. btc_addr = 0x00 + pubkey_hash+checksum

  - Convert the result into decimal and then Base58 encode it to get the final bitcoin address.

*Note: Need to verify this during implementation.*

### Sub process: Decode op_script to get destination address (btc_addr)



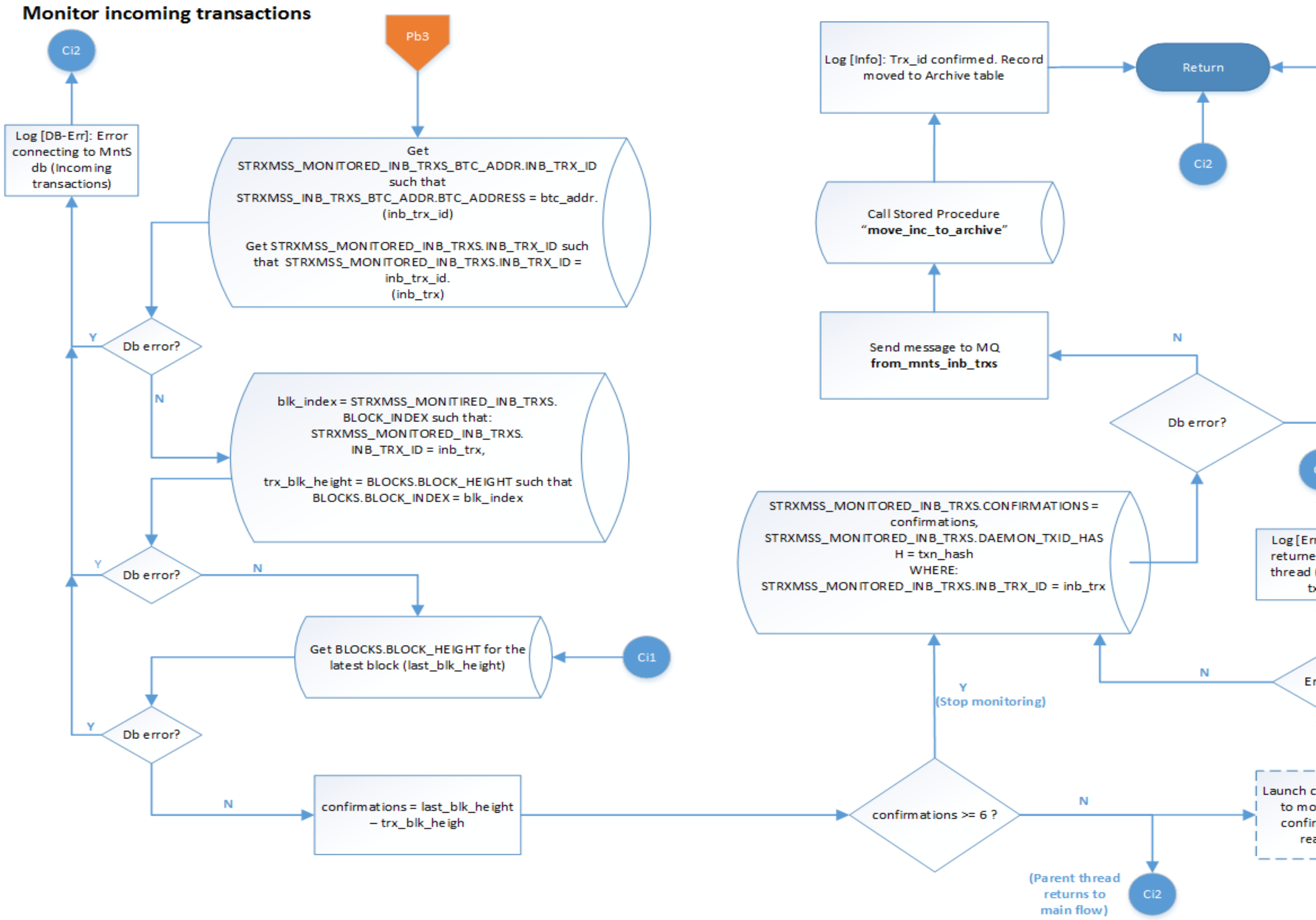
Note: In case of errors (system or database) during scanning transaction data, the flow will just skip further processing and proceed to next output in the transaction. Manual debugging of transaction data based on information in log files might be needed in this case.

### 4.7.3 Monitor incoming transactions

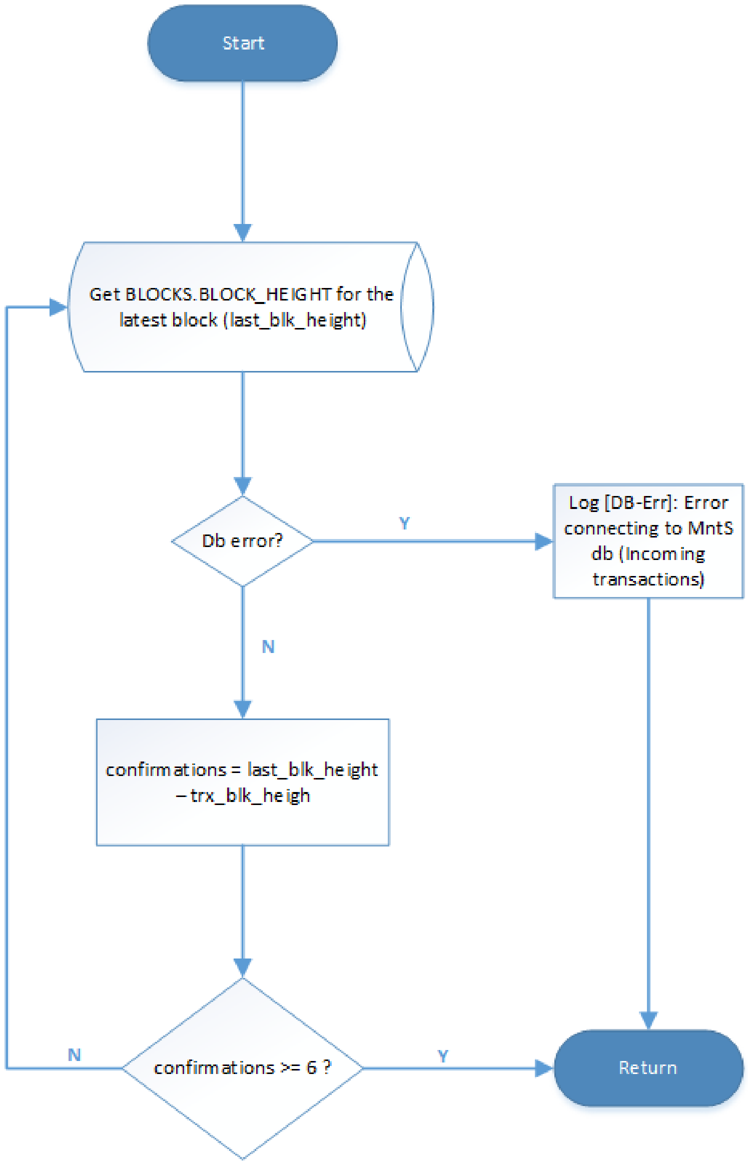
If transaction is found to be incoming, the Inc. Thread will be invoked to monitor confirmations for this transaction. When confirmations reach 6 or more, the transaction will no longer be monitored.

Detailed steps are as follows:

10. Inc. Thread will be invoked for one of the btc_addr that was found to have incoming funds from this transaction. Note that one transaction may have multiple btc_addr that have incoming funds. However any one of these addresses is sufficient for processing following steps. This is because MntS monitors confirmations of a transaction and not of a btc address.
11. Get the corresponding record in STRXMSS_MONITORED_INB_TRXS table (from btc_addr received from sub procedure above).
12. From BLOCK_INDEX value, get corresponding record in BLOCKS table and get the BLOCK_HEIGHT.
13. Get the block height for latest block in BLOCKS table.
14. Calculate confirmation by taking difference of these 2 heights.
15. Launch a child thread to keep checking confirmations if confirmations value is less than 6.
16. When confirmations reach 6 or more, send message to MQ, move records from STRXMSS_MONITORED_BTC_ADDR to Archive and delete corresponding records from STRMSS_MONITORED_INB_TRXS, STRMSS_INB_TRXS_BTC_ADDR tables.


Monitor incoming transactions
Ci2
Pb3
Log [DB-Err]: Error connecting to MntS db (Incoming transactions)
Get STRXMSS_MONITORED_INB_TRXS_BTC_ADDR.INB_TRX_ID such that STRXMSS_INB_TRXS_BTC_ADDR.BTC_ADDRESS = btc_addr. (inb_trx_id)
Get STRXMSS_MONITORED_INB_TRXS.INB_TRX_ID such that STRXMSS_MONITORED_INB_TRXS.INB_TRX_ID = inb_trx_id. (inb_trx)
Y
Db error?
N
blk_index = STRXMSS_MONITIRED_INB_TRXS.BLOCK_INDEX such that: STRXMSS_MONITORED_INB_TRXS.INB_TRX_ID = inb_trx,
trx_blk_height = BLOCKS.BLOCK_HEIGHT such that BLOCKS.BLOCK_INDEX = blk_index
Y
Db error?
N
Get BLOCKS.BLOCK_HEIGHT for the latest block (last_blk_height)
Ci1
Y
Db error?
N
confirmations = last_blk_height – trx_blk_heigh
confirmations >= 6 ?
Y
(Stop monitoring)
N
Ci2
(Parent thread returns to main flow)
STRXMSS_MONITORED_INB_TRXS.CONFIRMATIONS = confirmations, STRXMSS_MONITORED_INB_TRXS.DAEMON_TXID_HASH = txn_hash WHERE: STRXMSS_MONITORED_INB_TRXS.INB_TRX_ID = inb_trx
Db error?
N
Send message to MQ from_mnts_inb_trxs
Call Stored Procedure "move_inc_to_archive"
Log [Info]: Trx_id confirmed. Record moved to Archive table
Return
Ci2
N

**Child thread to monitor confirmations for incoming transactions**



### 4.7.4 Monitor outbound transactions

Outbound transactions are the transactions that are sent from the system.

STrxMSS Thread will pick these transactions from MQ. These transactions will be stored in the STRXMSS_MONITORED_TRXS table until they get 6 confirmations. Upon receiving 6 (or more) confirmations, a message will be sent to MQ and the record will be moved to Archive table.

For each transaction from STRXMSS_MONITORED_TRXS table:

1. Check if the Daemon_Txid_hash value matches with the transaction hash obtained in the main flow.
2. If match not found in step 1, it means that this outbound transaction was not included in this block. Check if the transaction has been in the mempool for more than 1 week (by comparing the created timestamp and current timestamp). If yes, send message to MQ. If not, continue to next record.
3. If match found in step 1, it means that the outbound transaction has been included in this block. Get the block index corresponding to transaction id.
4. Get the block height from BLOCKS table corresponding to the block index.

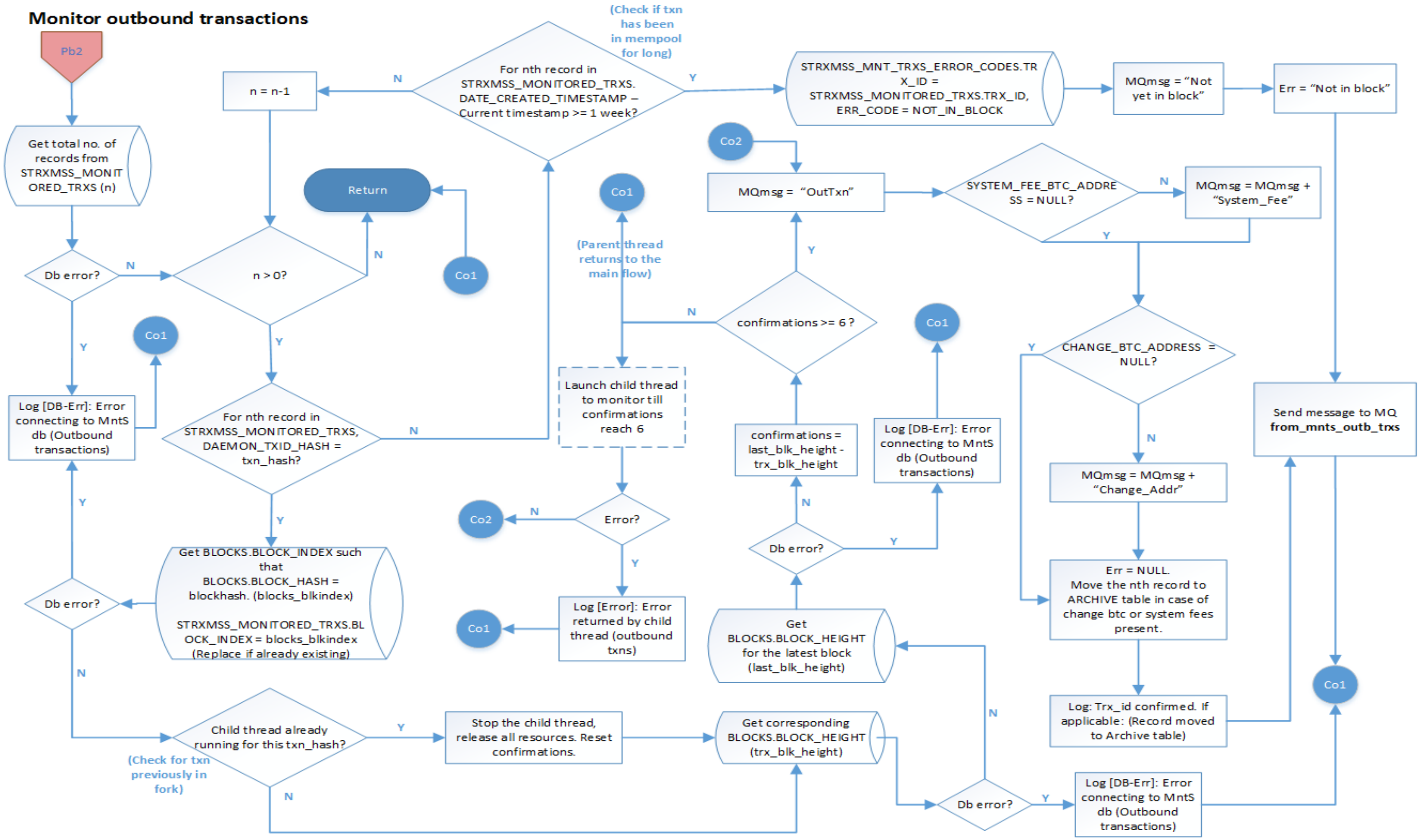
*(Note that STRXMSS_MONITORED_TRXS.BLOCK_INDEX may already be existing if this transaction was part of a fork. Hence replace any existing value during processing this step.)*

5. Get the block height of latest block from the BLOCKS table.
6. Calculate difference in block heights from above steps. Check if difference is greater than or equal to 6.

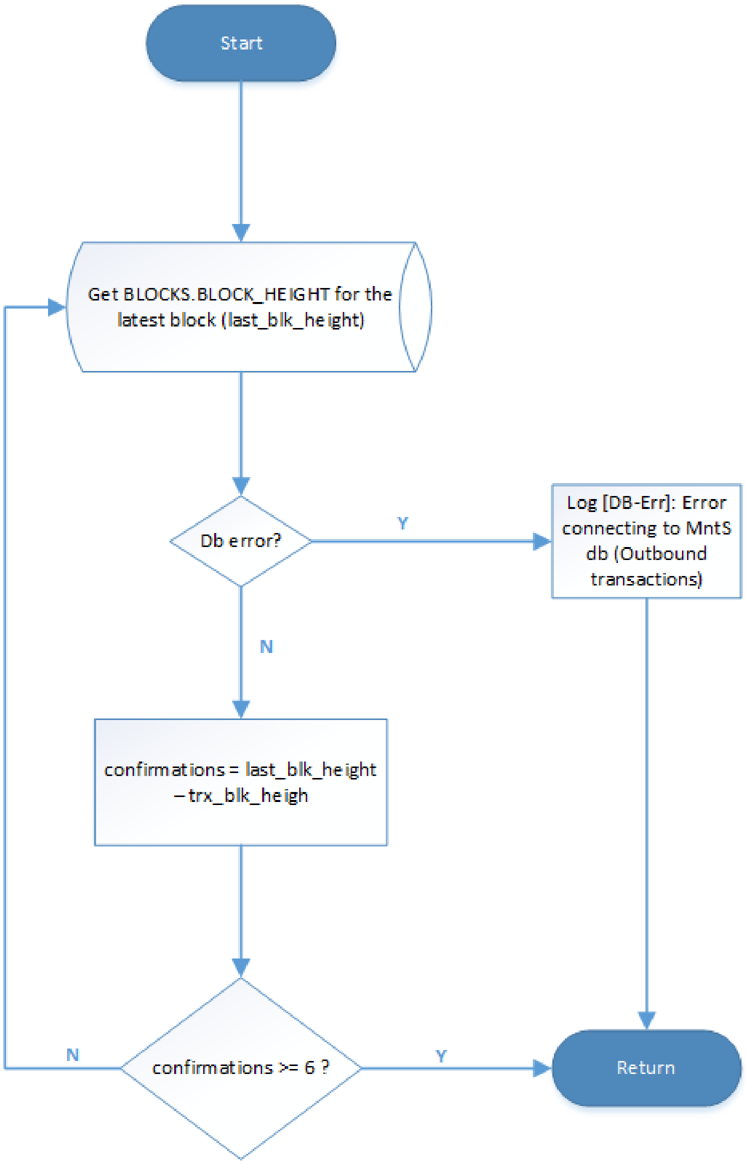
a. If yes:
   - Check if there are any change bitcoins and / or system fees associated with this transaction.
     - if yes, append respective message for MQ
   - Send a message to MQ for transaction confirmation
   - Move the record from STRXMSS_MONITORED_TRXS to ARCHIVE table only if there was change btc or system fees involved. This is because the Archive table will have all btc addresses previously being used for monitoring. If there was no change btc or system fees, no btc address need to be monitored from this transaction.

b. If not, continue monitoring. Do not move the record to ARCHIVE, so that MntS will continue monitoring this transaction.

**Monitor outbound transactions**



**Child thread to monitor confirmations for outbound transcations**



Some Scenarios for Outbound transactions:

**1. Blockchain Fork**

Note that a transaction will be returned to the mempool in the event of a fork.

Example scenario:

An IntDS outbound transaction is included in a block. Another block is mined at the exact same time. The second block becomes part of the main blockchain, making the first block orphan. All transactions that were included in the first block will be returned to the mempool. Thus our outbound transaction will appear in the mempool, as if it was never included in any block.

**MntS handling:**

Since MntS checks for inclusion of outbound transactions in every new block and since the transaction will remain in the STRXMSS_MONITORED_TRXS table till it gets 6 confirmations, this transaction will be available for monitoring when it is part of a new block.

When MntS will begin monitoring this transaction from new block, it will most probably encounter the thread that is still checking for confirmations on the orphan block. When this happens, MntS will stop the previous thread and create new one to monitor the transaction in the main blockchain.

**2. Transaction not included in a block for long time**

Example scenario:

IntDS sends an outbound transaction. But this transaction does not get included in a block for considerable amount of time (1 week or more).

**MntS handling:**

MntS will continue monitoring this transaction. However, MntS will send a message to MQ stating the transaction hash and error code so that STrxMSS knows that this particular transaction has been sitting in the mempool for long. Note that at this point, IntDS will not re-broadcast or do anything else to counteract this situation. It will just store information in the database about this transaction.

Once 6 confirmations are reached, MntS will send appropriate message to MQ just like normal outbound transcation.

**3. Transaction rejected by blockchain**

Refer section 4.7.5.

### 4.7.5 Monitor log files

**Monitor debug logs for reject messages**

An outbound message when sent over the P2P network, can be rejected by one or more peers.

BIP-61 [2.25] was introduced to provide feedback to peers about why their blocks or transactions were rejected. For IntDS, this is of relevance if any of the system’s outbound transactions are rejected. In this case, system will have to scan the reject message received from the peer and store all relevant information in the database.

There are 3 categories for reject message according to BIP-61 [2.25]:

1. version
2. transaction
3. block

Currently, MntS will monitor only transaction reject messages i.e. it will monitor which outbound transactions have been rejected by one or more peers.

The FOS Core Daemon component will be modified to create debug log files for MntS. Refer Section 2.11.2 for details about the log files generated. The reject messages will be logged in MntS-DEBUG log files.

**Record format for reject message:**

Each record in the file will be a separate line. The format of the record will be as follows:

***MessageType: Category: Code: Reason: Hash***

Example:

*Reject: Tx: 0x10: Transaction is invalid:*
*Hash_d1231fa2bcec333cef9565bb26ab2e651d3988a6b4129efddd649c4cea6e3815*

Details of each field in the record:

**MessageType** = the message type of the P2P network message received from peer(s). The first character will be uppercase.

Example: For a BIP-61 message the MessageType will be “Reject”.

**Category** = Specific category of the message.

For example: For a BIP-61 message, there can be following values for Category:

1. version
2. tx
3. block

**Code** = specific code generated for a category.

Refer the table below for codes generated for each of the 3 categories:

| **Category: version** | |
|---|---|
| **Code** | **Description** |
| 0x11 | Client is an obsolete, unsupported version |
| 0x12 | Duplicate version message received |
| **Category: tx** | |
| **Code** | **Description** |
| 0x10 | Transaction is invalid for some reason (invalid signature, output value greater than input, etc.) |
| 0x12 | An input is already spent |
| 0x40 | Not mined/relayed because it is “non-standard” (type or version unknown by the server) |
| 0x41 | One or more output amounts are below the ‘dust’ threshold |
| 0x42 | Transaction does not have enough fee/priority to be relayed or mined |
| **Category: block** | |
| **Code** | **Description** |
| 0x10 | Block is invalid for some reason (invalid proof-of-work, invalid signature, etc) |
| 0x11 | Block’s version is no longer supported |
| 0x43 | Inconsistent with a compiled-in checkpoint |

**Reason** = human readable message for debugging. This can be the text given in the Description column in the above table.

Example: For code 0x12 for category “tx”, reason will be “An input is already spent”.

**Hash** = transaction or block hash that is being rejected. This is an optional field and will be present only in case of transaction or block rejection.

This field will start with the string “Hash” followed by underscore and then the actual transaction hash.

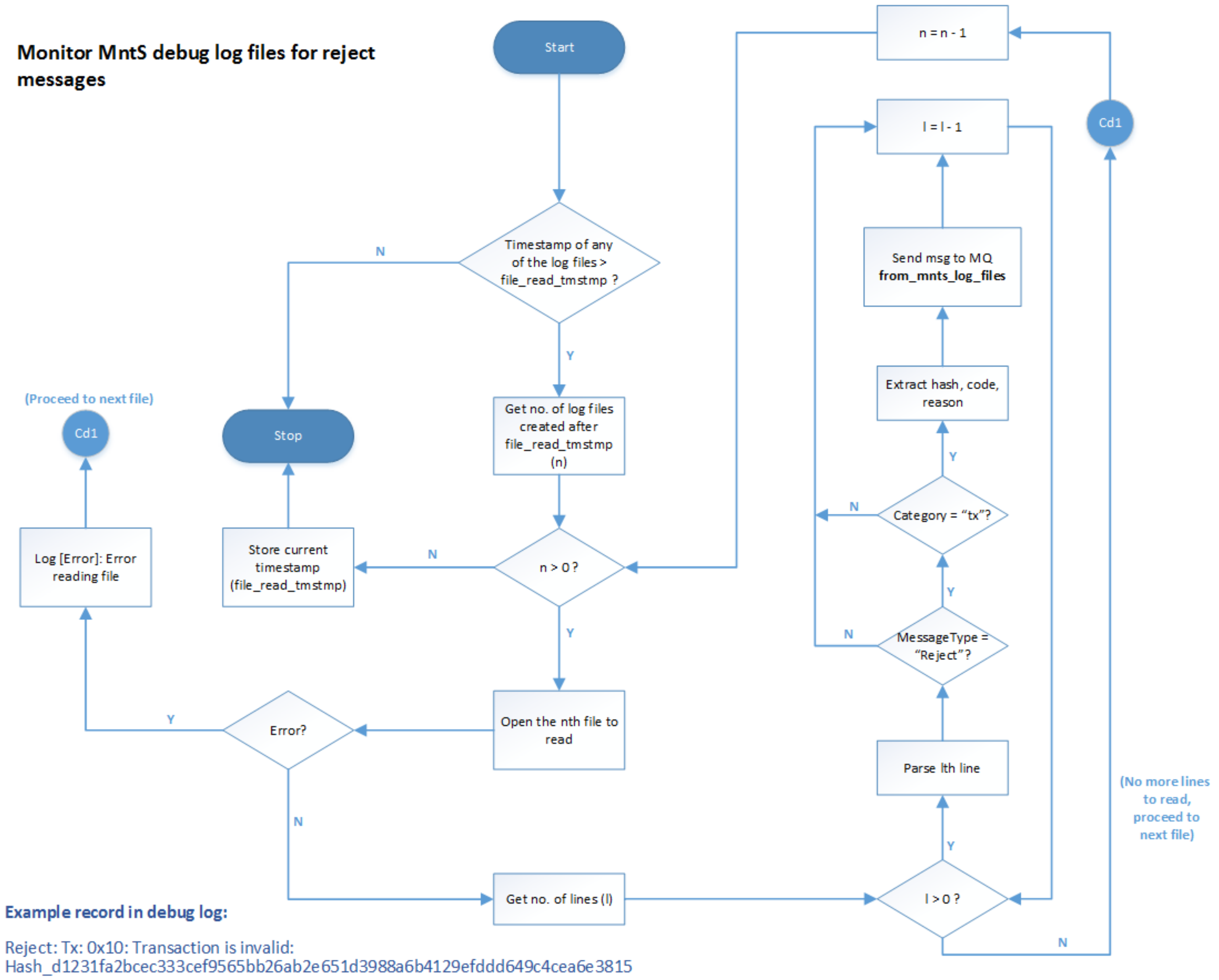
Example: Hash_d1231fa2bcec333cef9565bb26ab2e651d3988a6b4129efddd649c4cea6e3815

Details of the process:

MntS thread (Log Thread) will periodically (every 3 minutes) monitor the MntS debug log files for reject messages and send message to MQ if reject message is found. This monitoring will start after the system sends its first outbound transaction.

Log thread when started, will first check if there are any log files generated after the last read timestamp.

- If not, stop
- If yes, Log thread will scan each file. Every line will be parsed as follows:
    - Check if Message Type = Reject
        - If not, continue
        - If yes, check if Category = 'tx'
            - If not, continue
            - If yes: Extract the code, reason and hash. Send message to MQ.
- Store the timestamp of reading log file(s).



#### 4.7.6 Monitor archived addresses

MntS will also have to monitor all transactions/btc addresses stored in Archive table.

This section will be filled after the archiving policy has been decided for IntDS system.

Archive table / databse should contain all addresses that have been used up previously for receiving btc. This includes:

- btc addresses used for incoming transactions
- system addresses used for receiving system fees
- change addresses used for receiving change btc from an outgoing transaction

Althought IntDS will generate new btc address everytime to receive new btc, sender(s) can still send btc to an old / already used btc address. There is no way to stop anyone to send btc to a valid btc address. Hence IntDS will archive all addresses that the system uses for receiving btc.

# 5. Intelligent Daemon System Interfaces

Some of the sub-systems must provide interfaces to access its functionality according to Microservice architecture approach. The interface implementation must satisfy RESTful specification requirements. Jersey implementation of JAX-RS should be used for development. The sections below provide detailed information about sub-systems interfaces.

The following notation is used

- <…> - required parameters
- […] – optional parameters

The Restful URI call must be in the following format:

1. STrxMSS Interface URI: *https://[Load Balancer Host Name]/StrxMssService/[Function Name]*
2. DmnCS Interface URI: *https://[Load Balancer Host Name]/bitcoinService/[Function Name]*

The following notation is used

- Load Balancer Host Name – hostname that points to the master node in load balancing layer
- Function Name – The name from the first column of the table

All Functions should be called via HTTPS POST

Parameters must be pasted in HTTPS request body in the following format:

[Parameter1], [Parameter2], … , [ParameterN]

## 5.1 Single-sig Transaction Management SubSystem Interface

Current paragraph provides description of RESTful Web Service which is responsible for STrxMSS interface. All the requests to STrxMSS from external systems are coming via this interface. External systems as clients should send POST request to web service.

***Note****: This description should be updated after creation of class diagrams. Mapping between functions names and Java classes/methods should be included later.*

### 5.1.1 Wallet Functions

#### getWalletBalance

Function returns information about Wallet balances and “Warm Storage” flag by given Wallet Identifier from STrxMSS DB or error’s data in case system error.

| Request Parameters | Response Parameters | Response Type | Java Class (including package) | Java Method |
|---|---|---|---|---|
| *<daemonWalletId>* | <daemonWalletId>, <currentBalance>, <availableBalance>, <isLocked><br>**Error Response**: <errCodeId>, <error> | `JSONObject` | | |

**Request Parameters:**

| Parameter | Java Type | Length | Required | STrxMSS DB mapping | Example | Description |
|---|---|---|---|---|---|---|
| *daemonWalletId* | String | min 4 chars, max 60 chars | mandatory | Table: WALLETS<br>Field: WALLET_ID<br>Type: UUID | 067e6162-3b6f-4ae2-a171-2470b63dff00 | Wallet identifier in the STrxMSS DB |

**Response Parameters in JSONObject:**

| Parameter | Java Type | STrxMSS DB mapping | Example | Description |
|---|---|---|---|---|
| daemonWalletId | String | Table: WALLETS<br>Field: WALLET_ID<br>Type: UUID | 067e6162-3b6f-4ae2-a171-2470b63dff00 | Wallet Identifier in the STrxMSS DB. |
| currentBalance | BigDecimal | Table: WALLETS<br>Field: BTC_BALANCE | 1.45 | Total Wallet balance. “0.0” by default |
| availableBalance | BigDecimal | Table: WALLETS<br>Field: BTC_AVAILABLE_FUNDS | 0.45 | Available funds which can be used in new wallet transactions. “0.0” by default. |
| isLocked | int | Table: WALLETS<br>Field: IS_LOCKED | 1 | True (1) if wallet funds are locked in “Warm Storage” trxs otherwise false (0). “0” by default. |
| **Error Response** | | | | |
| errCodeId | int | Table: ERROR_CODES<br>Field: ERR_CODE_ID | 1 | System Error Identifier in the STrxMSS DB. |
| Error | String | Table: ERROR_CODES<br>Field: ERROR_CODE | “Balance calculation error” | System Error Code |

**Examples:**

| Function Call | Java Request Example for POST | Response Example |
|---|---|---|
| *https://[Load Balancer Host Name]/StrxMssService/ getWalletBalance* | String input = “{\”*daemonWalletId* \”:\”067e6162-3b6f-4ae2-a171-2470b63dff00\”}”; | {“daemonWalletId”: “067e6162-3b6f-4ae2-a171-2470b63dff00”, “currentBalance”:”1.45”, “availableBalance”:”0.45”, “isLocked”: 1 } |
| | | Error example: { “errCodeId”: 1, “error”: “Balance calculation error” } |

### getWalletsBalances

Function returns information about Wallets balances and “Warm Storage” flags by given list of Wallets Identifiers from STrxMSS DB or error’s data in case system error.

| Request Parameters | Response Parameters | Response Type | Java Class (including package) | Java Method |
|---|---|---|---|---|
| *<daemonWalletIds>* | <daemonWalletId>, <currentBalance>, <availableBalance>, <isLocked> | Type: `JSONArray` each array member is `JSONObject` | | |
| | **Error Response**: <errCodeId>, <error> | | | |

**Request Parameters:**

| Parameter | Java Type | Length | Required | STrxMSS DB mapping | Example | Description |
|---|---|---|---|---|---|---|
| *daemonWalletIds* | List, Type of each list object is String | For each String: min 4 chars, max 60 chars | mandatory | Table: WALLETS Field: WALLET_ID Type: UUID | 067e6162-3b6f-4ae2-a171-2470b63dff00 | Each list element is Wallet Identifier from STrxMSS DB |

**Response Parameters in JSONObject:**

| Parameter | Java Type | STrxMSS DB mapping | Example | Description |
|---|---|---|---|---|
| daemonWalletId | String | Table: WALLETS Field: WALLET_ID Type: UUID | 067e6162-3b6f-4ae2-a171-2470b63dff00 | Wallet Identifier from STrxMSS DB. |
| currentBalance | BigDecimal | Table: WALLETS Field: BTC_BALANCE | 1.45 | Total Wallet balance. “0.0” by default. |

| | | | | |
|---|---|---|---|---|
| availableBalance | BigDecimal | Table: WALLETS<br>Field:<br>BTC_AVAILABLE_FUNDS | 0.45 | Available funds which can be used in new Wallet transactions. “0.0” by default. |
| isLocked | int | Table: WALLETS<br>Field: IS_LOCKED | 1 | True (1) if wallet funds are locked in “Warm Storage” trxs otherwise false (0). “0” by default. |
| **Error Response** | | | | |
| errCodeId | int | Table: ERROR_CODES<br>Field: ERR_CODE_ID | 1 | System Error Identifier in the STrxMSS DB. |
| Error | String | Table: ERROR_CODES<br>Field: ERROR_CODE | “Balance calculation error” | System Error Code |

**Examples:**

| Function Call | Java Request Example for POST | Response Example |
|---|---|---|
| *https://[Load Balancer Host Name]/StrxMssService/getWalletsBalances* | List<String> daemonWalletIds = Arrays.asList(“067e6162-3b6f-4ae2-a171-2470b63dff00”, “067e6162-3b6f-4ae2-a171-2470b63dff00”, …);<br>JSONObject entity = new JSONObject();<br>JSONArray entityParams = new JSONArray();<br>entityParams.addAll(daemonWalletIds);<br>entity.put(“daemonWalletIds”, entityParams); | JASONArray of JASONObjects<br>[ {“daemonWalletId”: “067e6162-3b6f-4ae2-a171-2470b63dff00”,<br>“currentBalance”:”1.45”,<br>“availableBalance”:”0.45”,<br>“isLocked”: 1<br>}, …] |
| | | Error example:<br>{ “errCodeId”: 1, “error”: “Balance calculation error” } |

### allWalletsBalances

Function returns information about all system Wallets balances and “Warm Storage” flags or error’s data in case system error.

| Request Parameters | Response Parameters | Response Type | Java Class (including package) | Java Method |
|---|---|---|---|---|
| *N/A* | <daemonWalletId>, <currentBalance>, <availableBalance>, <isLocked> | Type: `JSONArray` each array member is `JSONObject` | | |
| | **Error Response**:<br><errCodeId>, <error> | | | |

**Response Parameters in JSONObject:**

| Parameter | Java Type | STrxMSS DB mapping | Example | Description |
|---|---|---|---|---|

| | | | | |
|---|---|---|---|---|
| daemonWalletId | String | Table: WALLETS<br>Field: WALLET_ID<br>Type: UUID | 067e6162-3b6f-4ae2-a171-2470b63dff00 | Wallet Identifier from STrxMSS DB |
| currentBalance | BigDecimal | Table: WALLETS<br>Field: BTC_BALANCE | 1.45 | Total Wallet balance. “0.0” by default. |
| availableBalance | BigDecimal | Table: WALLETS<br>Field:<br>BTC_AVAILABLE_FUNDS | 0.45 | Available funds which can be used in new Wallet transactions. “0.0” by default. |
| isLocked | int | Table: WALLETS<br>Field: IS_LOCKED | 1 | True (1) if Wallet funds are locked in “Warm Storage” trxs otherwise false (0). “0” by default. |
| **Error Response** | | | | |
| errCodeId | int | Table: ERROR_CODES<br>Field: ERR_CODE_ID | 1 | System Error Identifier in the STrxMSS DB. |
| Error | String | Table: ERROR_CODES<br>Field: ERROR_CODE | “Balance calculation error” | System Error Code |

**Examples:**

| Function Call | Response Example |
|---|---|
| *https://[Load Balancer Host Name]/StrxMssService/allWalletsBalances* | JASONArray of JASONObjects<br>[ {“daemonWalletId”: “067e6162-3b6f-4ae2-a171-2470b63dff00”,<br>“currentBalance”:”1.45”,<br>“availableBalance”:”0.45”,<br>“isLocked”: 1<br>}, …] |
| | Error example:<br>{ “errCodeId”: 1, “error”: “Balance calculation error” } |

### addNewWallet

Function creates new Wallet and returns Wallet Identifier from the STrxMSS DB and user’s part of mnemonic seed which should not be stored in the DB or error’s data in case system error.

| Request Parameters | Response Parameters | Response Type | Java Class (including package) | Java Method |
|---|---|---|---|---|
| *isSystemWallet* | <daemonWalletId>, <mnmSeedUserPart> | `JSONObject` | | |
| | **Error Response**: <errCodeId>, <error> | | | |

**Request Parameters:**

| Parameter | Java Type | Length | Required | STrxMSS DB mapping | Example | Description |
|---|---|---|---|---|---|---|
| *isSystemWallet* | int | 0 or 1 | mandatory | Table: WALLETS<br>Field: IS_SYSTEM_WALLET<br>Type: boolean | 0 | True (1) if this Wallet is Company Wallet otherwise false (0). False (0) by default. |

**Response Parameters in JSONObject:**

| Parameter | Java Type | STrxMSS DB mapping | Example | Description |
|---|---|---|---|---|
| daemonWalletId | String | Table: WALLETS<br>Field: WALLET_ID<br>Type: UUID | 067e6162-3b6f-4ae2-a171-2470b63dff00 | Wallet Identifier from STrxMSS DB |
| mnmSeedUserPart | String | N/A | asdfgjhgjads | User's part of mnemonic seed. |
| **Error Response** | | | | |
| errCodeId | int | Table: ERROR_CODES<br>Field: ERR_CODE_ID | 2 | System Error Identifier in the STrxMSS DB. |
| Error | String | Table: ERROR_CODES<br>Field: ERROR_CODE | "Wallet creation error" | System Error Code |

**Examples:**

| Function Call | Response Example |
|---|---|
| *https://[Load Balancer Host Name]/StrxMssService/addNewWallet* | {<br>"daemonWalletId":" 067e6162-3b6f-4ae2-a171-2470b63dff00",<br>"mnmSeedUserPart":" asdfgjhgjads"<br>} |
| | Error example:<br>{ "errCodeId": 2, "error": "New Wallet creation error" } |

## getWalletData

Function returns information about Wallet by given Wallet Identifier from STrxMSS DB or error's data in case system error.

| Request Parameters | Response Parameters | Response Type | Java Class (including package) | Java Method |
|---|---|---|---|---|
| *<daemonWalletId>* | <daemonWalletId>,<br><currentBalance>, | `JSONObject` | | |

| | | | | |
|---|---|---|---|---|
| | <availableBalance>, <isLocked>, <isSystemWallet>, <dateCreated>, <numberInbTrxs>, <numberOutbTrxs><br><br>**Error Response**: <errCodeId>, <error> | | | |

**Request Parameters:**

| Parameter | Java Type | Length | Required | STrxMSS DB mapping | Example | Description |
|---|---|---|---|---|---|---|
| *daemonWalletId* | String | min 4 chars, max 60 chars | mandatory | Table: WALLETS<br>Field: WALLET_ID<br>Type: UUID | 067e6162-3b6f-4ae2-a171-2470b63dff00 | Wallet identifier in the STrxMSS DB |

**Response Parameters in JSONObject:**

| Parameter | Java Type | STrxMSS DB mapping | Example | Description |
|---|---|---|---|---|
| daemonWalletId | String | Table: WALLETS<br>Field: WALLET_ID<br>Type: UUID | 067e6162-3b6f-4ae2-a171-2470b63dff00 | Wallet Identifier in the STrxMSS DB. |
| currentBalance | BigDecimal | Table: WALLETS<br>Field: BTC_BALANCE | 1.45 | Total Wallet balance. “0.0” by default |
| availableBalance | BigDecimal | Table: WALLETS<br>Field: BTC_AVAILABLE_FUNDS | 0.45 | Available funds which can be used in new wallet transactions. “0.0” by default. |
| isLocked | int | Table: WALLETS<br>Field: IS_LOCKED | 1 | True (1) if wallet funds are locked in “Warm Storage” trxs otherwise false (0). “0” by default. |
| isSystemWallet | int | Table: WALLETS<br>Field: IS_SYSTEM_WALLET | 0 | True (1) if wallet owner is IntDS otherwise false (0). “0” by default. |
| dateCreated | String | Table: WALLETS<br>Field: DATE_CREATED | 18:10 25-07-2014 | Wallet creation date and time in IntDS |
| numberInbTrxs | int | STrxMSS calculation according to data from WALLETS_TRANSACTION and TRANSACTIONS tables | 0 | Number of Inbound transactions in this Wallet. “0” by default. |
| numberOutbTrxs | int | STrxMSS calculation according to data from | 1 | Number of Outbound transactions in this Wallet. |

| | | | | |
|---|---|---|---|---|
| | | WALLETS_TRANSACTION and TRANSACTIONS tables | | "0" by default. |
| **Error Response** | | | | |
| errCodeId | int | Table: ERROR_CODES<br>Field: ERR_CODE_ID | 3 | System Error Identifier in the STrxMSS DB. |
| Error | String | Table: ERROR_CODES<br>Field: ERROR_CODE | "Wallet was not found" | System Error Code |

**Examples:**

| Function Call | Java Request Example for POST | Response Example |
|---|---|---|
| *https://[Load Balancer Host Name]/StrxMssService/getWalletData* | String input = "{\" *daemonWalletId*\":\"067e6162-3b6f-4ae2-a171-2470b63dff00\"}"; | {"daemonWalletId": "067e6162-3b6f-4ae2-a171-2470b63dff00",<br>"currentBalance":"1.45",<br>"availableBalance":"0.45",<br>"isLocked": 1,<br>"isSystemWallet": 0,<br>"dateCreated": "18:10 25-07-2014",<br>"numberInbTrxs": 0<br>"numberOutbTrxs": 1<br>} |
| | | Error example:<br>{ "errCodeId": 3, "error": "Wallet was not found" } |
| | | Error example:<br>{ "errCodeId": 4, "error": "Wallet signature validation error" } |

### 5.1.2 Outbound Transaction Functions

#### getTrxStatus

Function returns information about Transaction status by given Trx Identifier from STrxMSS DB or error's data in case system error.

| Request Parameters | Response Parameters | Response Type | Java Class (including package) | Java Method |
|---|---|---|---|---|
| *<daemonTrxId>* | <daemonTrxStatus> | `JSONObject` | | |
| | **Error Response**:<br><errCodeId>, <error> | | | |

**Request Parameters:**

| Parameter | Java Type | Length | Required | STrxMSS DB mapping | Example | Description |
|---|---|---|---|---|---|---|
| *daemonTrxId* | String | min 4 chars, max 60 chars | mandatory | Table: TRANSACTIONS<br>Field: TRX_ID<br>Type: UUID | 067e6162-3b6f-4ae2-a171-2470b63dff00 | Transaction identifier in the STrxMSS DB |

**Response Parameters in JSONObject:**

| Parameter | Java Type | STrxMSS DB mapping | Example | Description |
|---|---|---|---|---|
| daemonTrxStatus | String | Table: TRX_STATUSES<br>Field: STATUS | “Pending” | Transaction Status |
| **Error Response** | | | | |
| errCodeId | int | Table: ERROR_CODES<br>Field: ERR_CODE_ID | 5 | System Error Identifier in the STrxMSS DB. |
| Error | String | Table: ERROR_CODES<br>Field: ERROR_CODE | “Status was not found” | System Error Code |

**Examples:**

| Function Call | Java Request Example for POST | Response Example |
|---|---|---|
| *https://[Load Balancer Host Name]/StrxMssService/getTrxStatus* | String input = “{\”daemonTrxId\”:\”067e6162-3b6f-4ae2-a171-2470b63dff00\”}”; | {“daemonTrxStatus”: “Pending”} |
| | | Error example:<br>{ “errCodeId”: 5, “error”: “Status was not found” } |

### createSingleSigTrx

Function prepared Outbound transaction before send it to the block chain. All Inputs and Outputs of this transaction are correspond to P2PKH type only. Function returns IntDS trx identifier with trx status “In Progress”, Miner and IntDS fees for confirmation by External system or error’s data in case system error.

| Request Parameters | Response Parameters | Response Type | Java Class (including package) | Java Method |
|---|---|---|---|---|
| *<externalTrxId>,<br><fromDaemonWalletId>,<br><toRecepients>(<br>{<toBtcAddress>,<br><btcAmount>}<br>…<br>),* | <externalTrxId>,<br><tempTrxId>,<br><isMinerFeeEnough><br>[minerFee],<br>[intDSFee],<br><daemonTrxStatus> | `JSONObject` | | |

| | | | | |
|---|---|---|---|---|
| *[priorityFee], <externalPartMnmSeed>* | **Error Response**: <errCodeId>, <error> | | | |

**Request Parameters:**

| Parameter | Java Type | Length | Required | STrxMSS DB mapping | Example | Description |
|---|---|---|---|---|---|---|
| *externalTrxId* | String | min 4 chars, max 60 chars | mandatory | N/A | 067e6162-3b6f-4ae2-a171-2470b63dff00 | Transaction identifier from the External system |
| *fromDaemonWalletId* | String | min 4 chars, max 60 chars | mandatory | Table: WALLETS Field: WALLET_ID Type: UUID | 333e6162-3b6f-4ae2-a171-2470b63dff00 | Wallet identifier in the STrxMSS DB. Btc funds is sent from this wallet |
| *toRecepients* | JSONArray | | mandatory | | | JSON array of recepients Btc addresses and Btc amount to be sent. Each array member has a JASONObject type as: {“*toBtcAddress*”: “16UwLL9Risc3QfPqBUvKofHmBQ7wMtjvM”, “*btcAmount*”: “1.1”} |
| *toBtcAddress* | String | min 4 chars, max 50 chars | mandatory | Table: OUTPUTS Field: BTC_ADDRESS Type: varchar(50) | 16UwLL9Risc3QfPqBUvKofHmBQ7wMtjvM | Btc address of the Btc funds recipient |
| *btcAmount* | BigDecimal | | mandatory | Table: OUTPUTS Field: BTC_VALUE Type: numeric | 1 | Btc amount to send |
| *priorityFee* | BigDecimal | | optional | Table: TEMP_OUTB_TRXS | 0.1 | Bitcoins amount can be paid to increase priority and to |

| | | | | | | |
|---|---|---|---|---|---|---|
| | | | | Field: PRIORITY_FEE Type: numeric | | accelerate transaction. This is extra fee for Miner and IntDS |
| *externalPartMnmSeed* | String | min 5 chars, max 500 chars | mandatory | N/A | sdhsakdhsakjhd | User’s part of mnemonic seed for this wallet |

**Response Parameters in JSONObject:**

| Parameter | Java Type | STrxMSS DB mapping | Example | Description |
|---|---|---|---|---|
| externalTrxId | String | Table: TEMP_OUTB_TRXS Field: EXTERNAL_TRX_ID | 067e6162-3b6f-4ae2-a171-2470b63dff00 | Transaction identifier from the External system |
| tempTrxId | String | Table: TEMP_OUTB_TRXS Field: TEMP_TRX_ID | 111e6162-3b6f-4ae2-a171-2470b63dff00 | Temporary transaction identifier in the STrxMSS DB. |
| isMinerFeeEnough | int | N/A | 1 | True (1) if “Priority fee” given by external system >= “Miner fee” which should be paid for the transaction otherwise false (0) |
| minerFee | BigDecimal | Table: TEMP_OUTB_TRXS Field: MINER_FEE | 0.0001 | Bitcoins amount should be paid as Miner fee. “0.0” if there is not Miner fee. |
| intDSFee | BigDecimal | Table: TEMP_OUTB_TRXS Field: INTDS_FEE | 0.00007 | Bitcoins amount should be paid as IntDS fee. “0.0” if there is not Miner fee. |
| daemonTrxStatus | String | Table: TRX_STATUSES Field: STATUS | “In Progress” | Transaction Status |
| **Error Response** | | | | |
| errCodeId | int | Table: ERROR_CODES Field: ERR_CODE_ID | 6 | System Error Identifier in the STrxMSS DB. |
| Error | String | Table: ERROR_CODES Field: ERROR_CODE | “Transaction creation error” | System Error Code |

**Examples:**

| Function Call | Java Request Example for POST | Response Example |
|---|---|---|
| *https://[Load Balancer Host Name]/STrxMssService/createSingleSigTrx* | JSONObject entity = new JSONObject();<br>entity.put(“*externalTrxId*”,“067e6162-3b6f-4ae2-a171-2470b63dff00”);<br>entity.put(“fromDaemonWalletId”, “067e6162-3b6f-4ae2-a171-2470b63dff00”);<br>JSONArray entityArray = new JSONArray(); | {“externalTrxId”: “067e6162-3b6f-4ae2-a171-2470b63dff00”,<br>“tempTrxId”: “111e6162-3b6f-4ae2-a171-2470b63dff00”, |

| | | |
|---|---|---|
| | JSONObject elem = new JSONObject();<br>elem.put(“toBtcAddress”, “16UwLL9Risc3QfPqBUvKofHmBQ7wMtjvM”);<br>elem.put(“btcAmount”, “1.1”);<br>entityArray.add(elem);<br>…<br>entity.put(“toRecepients”, entityArray);<br>[entity.put(“priorityFee”, “0.1”);]<br>entity.put(“externalPartMnmSeed”, “sdhsakdhsakjhd”); | “minerFee”: “0.0001”, “intDSFee”: “0.00007”, “daemonTrxStatus”: “In Progress”}<br>Error example:<br>{ “errCodeId”: 6, “error”: “Transaction creation error” } |

## sendSingleSigTrx

Function is sending prepared transaction to the blockchain and moving trx data from temporary table to the permanent transactions table in the DB with “Pending” status.

| Request Parameters | Response Parameters | Response Type | Java Class (including package) | Java Method |
|---|---|---|---|---|
| *<externalTrxId>, <tempTrxId>* | <externalTrxId>, <daemonTrxId>, <daemonTrxStatus><br>**Error Response**: <errCodeId>, <error> | `JSONObject` | | |

**Request Parameters:**

| Parameter | Java Type | Length | Required | STrxMSS DB mapping | Example | Description |
|---|---|---|---|---|---|---|
| *externalTrxId* | String | min 4 chars, max 60 chars | mandatory | N/A | 067e6162-3b6f-4ae2-a171-2470b63dff00 | Transaction identifier from the External system |
| *tempTrxId* | String | min 4 chars, max 60 chars | mandatory | Table: TEMP_OUTB_TRXS Field: TEMP_TRX_ID Type: UUID | 111e6162-3b6f-4ae2-a171-2470b63dff00 | Temporary transaction identifier in the STrxMSS DB. |

**Response Parameters in JSONObject:**

| Parameter | Java Type | STrxMSS DB mapping | Example | Description |
|---|---|---|---|---|
| externalTrxId | String | N/A | 067e6162-3b6f-4ae2-a171-2470b63dff00 | Transaction identifier from the External system |

| | | | | |
|---|---|---|---|---|
| daemonTrxId | String | Table: TRANSACTIONS<br>Field: TRX_ID | 222e6162-3b6f-4ae2-a171-2470b63dff00 | Transaction identifier in the STrxMSS DB |
| daemonTrxStatus | String | Table: TRX_STATUSES<br>Field: STATUS | “Pending” | Transaction Status |
| **Error Response** | | | | |
| errCodeId | int | Table: ERROR_CODES<br>Field: ERR_CODE_ID | 7 | System Error Identifier in the STrxMSS DB. |
| Error | String | Table: ERROR_CODES<br>Field: ERROR_CODE | “Transaction send error” | System Error Code |

**Examples:**

| Function Call | Java Request Example for POST | Response Example |
|---|---|---|
| *https://[Load Balancer Host Name]/StrxMssService/sendSingleSigTrx* | JSONObject entity = new JSONObject();<br>entity.put(“externalTrxId”, “067e6162-3b6f-4ae2-a171-2470b63dff00”);<br>entity.put(“tempTrxId”, “111e6162-3b6f-4ae2-a171-2470b63dff00”); | {“externalTrxId”: “067e6162-3b6f-4ae2-a171-2470b63dff00”, “daemonTrxId”: “222e6162-3b6f-4ae2-a171-2470b63dff00”, “daemonTrxStatus”: “Pending”}<br>Error example:<br>{ “errCodeId”: 7, “error”: “Transaction send error” } |

### deleteTempTrx

Function deletes temporary transaction data from DB tables. Function returns error’s data in case system error.

| Request Parameters | Response Parameters | Response Type | Java Class (including package) | Java Method |
|---|---|---|---|---|
| *<externalTrxId>, <tempTrxId>* | <externalTrxId>, <isTempTrxDeleted><br>**Error Response**: <errCodeId>, <error> | `JSONObject` | | |

**Request Parameters:**

| Parameter | Java Type | Length | Required | STrxMSS DB mapping | Example | Description |
|---|---|---|---|---|---|---|
| *externalTrxId* | String | min 4 chars, max 60 chars | mandatory | N/A | 067e6162-3b6f-4ae2-a171-2470b63dff00 | Transaction identifier from the External system |

| | | | | | | |
|---|---|---|---|---|---|---|
| *tempTrxId* | String | min 4 chars, max 60 chars | mandatory | Table: TEMP_OUTB_TRXS Field: TEMP_TRX_ID Type: UUID | 111e6162-3b6f-4ae2-a171-2470b63dff00 | Temporary transaction identifier in the STrxMSS DB. |

**Response Parameters in JSONObject:**

| **Parameter** | **Java Type** | **STrxMSS DB mapping** | **Example** | **Description** |
|---|---|---|---|---|
| externalTrxId | String | N/A | 067e6162-3b6f-4ae2-a171-2470b63dff00 | Transaction identifier from the External system |
| isTempTrxDeleted | int | N/A | 1 | True (1) if temporary transaction data was deleted otherwise false (0). |
| **Error Response** | | | | |
| errCodeId | int | Table: ERROR_CODES Field: ERR_CODE_ID | 8 | System Error Identifier in the STrxMSS DB. |
| Error | String | Table: ERROR_CODES Field: ERROR_CODE | "Error of Temp Transaction deleting" | System Error Code |

**Examples:**

| **Function Call** | **Java Request Example for POST** | **Response Example** |
|---|---|---|
| *https://[Load Balancer Host Name]/StrxMssService/deleteTempTrx* | JSONObject entity = new JSONObject(); entity.put("externalTrxId", "067e6162-3b6f-4ae2-a171-2470b63dff00"); entity.put("tempTrxId", "111e6162-3b6f-4ae2-a171-2470b63dff00"); | {"externalTrxId": "067e6162-3b6f-4ae2-a171-2470b63dff00", "isTempTrxDeleted": 1}<br>Error example:<br>{ "errCodeId": 8, "error": "Error of Temp Transaction deleting" } |

## createTransferFundsTrx

Function transfers Btc funds from one user Wallet to another. Fansction creates Temporary Single-sig transaction. Function returns error's data in case system error.

| **Request Parameters** | **Response Parameters** | **Response Type** | **Java Class (including package)** | **Java Method** |
|---|---|---|---|---|
| *<externalTrxId>*<br>*<fromDaemonWalletId>,*<br>*<mnmSeedUserPartFrom>,*<br>*<toDaemonWalletId>,* | <externalTrxId>,<br><tempTrxId>,<br>[minerFee],<br>[intDSFee], | `JSONObject` | | |

| | | | | |
|---|---|---|---|---|
| *<mnmSeedUserPartTo>, <btcAmount>* | <daemonTrxStatus><br>**Error Response**: <errCodeId>, <error> | | | |

**Request Parameters:**

| Parameter | Java Type | Length | Required | STrxMSS DB mapping | Example | Description |
|---|---|---|---|---|---|---|
| *externalTrxId* | String | min 4 chars, max 60 chars | mandatory | N/A | 067e6162-3b6f-4ae2-a171-2470b63dff00 | Transaction identifier from the External system |
| *fromDaemonWalletId* | String | min 4 chars, max 60 chars | mandatory | Table: WALLETS Field: WALLET_ID Type: UUID | 567e6162-3b6f-4ae2-a171-2470b63dff00 | Wallet identifier in the STrxMSS DB. Btc funds are transferred from this Wallet |
| *mnmSeedUserPartFrom* | String | N/A | mandatory | N/A | sdfdsfdsfs | User’s part of mnemonic seed for wallet from which Btc are transferred |
| *toDaemonWalletId* | String | min 4 chars, max 60 chars | mandatory | Table: WALLETS Field: WALLET_ID Type: UUID | 123e6162-3b6f-4ae2-a171-2470b63dff00 | Wallet identifier in the STrxMSS DB. Btc funds are transferred to this Wallet. |
| *mnmSeedUserPartTo* | String | N/A | mandatory | N/A | sdfdsffd | User’s part of mnemonic seed for wallet to which Btc are transferred |
| *btcAmount* | BigDecimal | | mandatory | Table: TEMP_OUTB_TRXS Field: BTC_AMOUNT Type: numeric | 1 | Btc amount to transfer |

**Response Parameters in JSONObject:**

| Parameter | Java Type | STrxMSS DB mapping | Example | Description |
|---|---|---|---|---|
| externalTrxId | String | Table: TEMP_OUTB_TRXS Field: EXTERNAL_TRX_ID | 067e6162-3b6f-4ae2-a171-2470b63dff00 | Transaction identifier from the External system |

| | | | | |
|---|---|---|---|---|
| tempTrxId | String | Table: TEMP_OUTB_TRXS<br>Field: TEMP_TRX_ID | 111e6162-3b6f-4ae2-a171-2470b63dff00 | Temporary transaction identifier in the STrxMSS DB. |
| minerFee | BigDecimal | Table: TEMP_OUTB_TRXS<br>Field: MINER_FEE | 0.0001 | Bitcoins amount should be paid as Miner fee. "0.0" if there is not Miner fee. |
| intDSFee | BigDecimal | Table: TEMP_OUTB_TRXS<br>Field: INTDS_FEE | 0.00007 | Bitcoins amount should be paid as IntDS fee. "0.0" if there is not Miner fee. |
| daemonTrxStatus | String | Table: TRX_STATUSES<br>Field: STATUS | "In Progress" | Transaction Status |
| **Error Response** | | | | |
| errCodeId | int | Table: ERROR_CODES<br>Field: ERR_CODE_ID | 6 | System Error Identifier in the STrxMSS DB. |
| Error | String | Table: ERROR_CODES<br>Field: ERROR_CODE | "Transaction creation error" | System Error Code |

**Examples:**

| Function Call | Java Request Example for POST | Response Example |
|---|---|---|
| *https://[Load Balancer Host Name]/StrxMssService/createTransferFundsTrx* | JSONObject entity = new JSONObject();<br>entity.put ("externalTrxId", "067e6162-3b6f-4ae2-a171-2470b63dff00");<br>entity.put ("fromDaemonWalletId", "567e6162-3b6f-4ae2-a171-2470b63dff00");<br>entity.put ("mnmSeedUserPartFrom", "sdfdsfdsfs");<br>entity.put ("toDaemonWalletId", "123e6162-3b6f-4ae2-a171-2470b63dff00");<br>entity.put ("mnmSeedUserPartTo", "sdfdsffd");<br>entity.put ("btcAmount", "10.3"); | {"externalTrxId": "067e6162-3b6f-4ae2-a171-2470b63dff00", "tempTrxId": "111e6162-3b6f-4ae2-a171-2470b63dff00", "minerFee": "0.0001", "intDSFee": "0.00007", "daemonTrxStatus": "In Progress"}<br>Error example:<br>{ "errCodeId": 9, "error": "Error of Transferring Funds" } |

## getTrxErrors

Function returns errors data of given transaction or error's data in case system error.

| Request Parameters | Response Parameters | Response Type | Java Class (including package) | Java Method |
|---|---|---|---|---|
| *<daemonTrxId>* | <daemonTrxId>,<br><trxErrors>(<br>{<errCodeId>,<br><errDateCreated>,<br><errCode>,<br><errDescr>}<br>)<br>**Error Response**: | `JSONObject`,<br>`JSONArray(`<br>`JSONObject`,<br>`JSONObject`<br>…<br>) | | |

| | | | | |
|---|---|---|---|---|
| | <errCodeId>, <error> | | | |

**Request Parameters:**

| Parameter | Java Type | Length | Required | STrxMSS DB mapping | Example | Description |
|---|---|---|---|---|---|---|
| *daemonTrxId* | String | min 4 chars, max 60 chars | mandatory | Table: TRANSACTIONS Field: TRX_ID Type: UUID | 067e6162-3b6f-4ae2-a171-2470b63dff00 | Transaction identifier in the STrxMSS DB |

**Response Parameters in JSONObject:**

| Parameter | Java Type | STrxMSS DB mapping | Example | Description |
|---|---|---|---|---|
| daemonTrxId | String | Table: TRANSACTIONS Field: TRX_ID | 067e6162-3b6f-4ae2-a171-2470b63dff00 | Transaction identifier in the STrxMSS DB |
| trxErrors | JASONArray | N/A | | Array of transaction errors. Each array member is JSONObject. Empty array if there are not errors associated with this transaction. |
| errCodeId | int | Table: TRANSACTIONS_ERROR_CODES Field: ERR_CODE_ID | 7 | Error code identity number. |
| errDateCreated | String | Table: TRANSACTIONS_ERROR_CODES Field: DATE_CREATED | “02-11-2016 19:21” | Date and time of error creation in the format: [dd-mm-yyyy hh:mm] |
| errCode | String | Table: INTDSYSTEM_ERROR_CODES Field: ERROR_CODE | “Transaction send error” | Error code. |
| errDescr | String | Table: INTDSYSTEM_ERROR_CODES Field: ERROR_DESCR | “STrxMSS error in the sending of transaction to blockchain” | Error description. |
| **Error Response** | | | | |
| errCodeId | int | Table: ERROR_CODES Field: ERR_CODE_ID | 10 | System Error Identifier in the STrxMSS DB. |
| Error | String | Table: ERROR_CODES Field: ERROR_CODE | “STrxMSS error” | System Error Code |

**Examples:**

| Function Call | Java Request Example for POST | Response Example |
|---|---|---|

| | | |
|---|---|---|
| *https://[Load Balancer Host Name]/StrxMssService/getTrxErrors* | String input = “{\”daemonTrxId\”:\”067e6162-3b6f-4ae2-a171-2470b63dff00\”}”; | {“daemonTrxId”: “067e6162-3b6f-4ae2-a171-2470b63dff00”,<br>“trxErrors”: (<br>{“errCodeId”: 7,<br>“errDateCreated”: ”02-11-2016 19:21”,<br>“errCode”: “Transaction send error”,<br>“errDescr”: “STrxMSS error in the sending of transaction to blockchain”<br>},<br>{…} …<br>)<br>} |
| | | Error example:<br>{ “errCodeId”: 10, “error”: “STrxMSS error” } |

## getOutbTrxData

Function returns some data of given outbound transaction or error’s data in case system error.

| Request Parameters | Response Parameters | Response Type | Java Class (including package) | Java Method |
|---|---|---|---|---|
| *<daemonTrxId>* | <daemonTrxId>, <daemonTrxStatus>, <isConfirmed>, <isRejected>, [minerFee], [systemFee] | `JSONObject` | | |
| | **Error Response**: <errCodeId>, <error> | | | |

**Request Parameters:**

| Parameter | Java Type | Length | Required | STrxMSS DB mapping | Example | Description |
|---|---|---|---|---|---|---|
| *daemonTrxId* | String | min 4 chars, max 60 chars | mandatory | Table: TRANSACTIONS<br>Field: TRX_ID<br>Type: UUID | 067e6162-3b6f-4ae2-a171-2470b63dff00 | Transaction identifier in the STrxMSS DB |

**Response Parameters in JSONObject:**

| Parameter | Java Type | STrxMSS DB mapping | Example | Description |
|---|---|---|---|---|
| daemonTrxId | String | Table: TRANSACTIONS<br>Field: TRX_ID | 067e6162-3b6f-4ae2-a171-2470b63dff00 | Transaction identifier in the STrxMSS DB |
| daemonTrxStatus | String | Table: TRX_STATUSES<br>Field: STATUS | “Pending” | Transaction Status |
| isConfirmed | int | N/A | 1 | True (1) if transaction is confirmed otherwise false (0). Trx is confirmed if there are 6 blocks after transaction block. |
| isRejected | int | N/A | 0 | True (0) if transaction is rejected by blockchain otherwise false (0). |
| minerFee | BigDecimal | Table: TRANSACTIONS<br>Field: MINER_FEE | 0.0 | Bitcoins amount should be paid as Miner fee. Zero by default. |
| systemFee | BigDecimal | Table: OUTPUTS<br>Field: BTC_VALUE | 0.0 | Bitcoins amount should be paid as IntDS system fee. Zero by default. |
| **Error Response** | | | | |
| errCodeId | int | Table: ERROR_CODES<br>Field: ERR_CODE_ID | 11 | System Error Identifier in the STrxMSS DB. |
| Error | String | Table: ERROR_CODES<br>Field: ERROR_CODE | “Trx data was not found” | System Error Code |

**Examples:**

| Function Call | Java Request Example for POST | Response Example |
|---|---|---|
| *https://[Load Balancer Host Name]/StrxMssService/getOutbTrxData* | String input = “{\”daemonTrxId\”:\”067e6162-3b6f-4ae2-a171-2470b63dff00\”}”; | {“ daemonTrxId”: “067e6162-3b6f-4ae2-a171-2470b63dff00”, “daemonTrxStatus”: “Pending”, “isConfirmed”: 1, “isRejected”: 0, “minerFee”: ”0.0”, “systemFee”: “0.0”} |
| | | Error example:<br>{ “errCodeId”: 11, “error”: “Trx data was not found” } |

### 5.1.3 Inbound Transaction Functions

#### findInbTrxForBtcAddress

Function searches all Inbound transactions associated with given Btc address. Function returns error’s data in case system error.

| Request Parameters | Response Parameters | Response Type | Java Class (including package) | Java Method |
|---|---|---|---|---|
| *<btcAddress>, <daemonWalletId>* | <btcAddress>, <inbTrxs>( {<daemonTrxId>, <btcAmount>, <dateCreated>} ) | JSONObject, JSONArray( JSONObject, JSONObject … ) | | |
| | **Error Response**: <errCodeId>, <error> | | | |

**Request Parameters:**

| Parameter | Java Type | Length | Required | STrxMSS DB mapping | Example | Description |
|---|---|---|---|---|---|---|
| *btcAddress* | String | min 4 chars, max 50 chars | mandatory | Table: OUTPUTS Field: BTC_ADDRESS Type: varchar | 16UwLL9Risc3QfPqBUvKofHmBQ7wMtjvM | Btc address of the Btc funds recipient |
| *daemonWalletId* | String | min 4 chars, max 60 chars | mandatory | Table: WALLETS Field: WALLET_ID Type: UUID | 067e6162-3b6f-4ae2-a171-2470b63dff00 | Wallet Identifier from STrxMSS DB |

**Response Parameters in JSONObject:**

| Parameter | Java Type | STrxMSS DB mapping | Example | Description |
|---|---|---|---|---|
| btcAddress | String | Table: OUTPUTS Field: BTC_ADDRESS | 16UwLL9Risc3QfPqBUvKofHmBQ7wMtjvM | Btc address of the Btc funds recipient |
| inbTrxs | JASONArray | N/A | | Array of confirmed Inbound transactions. Each array member is JSONObject. Empty array if there are not transactions associated with this Btc address. |
| daemonTrxId | String | Table: TRANSACTIONS Field: TRX_ID | 067e6162-3b6f-4ae2-a171-2470b63dff00 | Transaction identifier in the STrxMSS DB |
| btcAmount | BigDecima | Table: OUTPUTS Field: BTC_VALUE | 1.1 | Btc amount paied to this Btc address. |
| dateCreated | String | Table: TRANSACTIONS Field: DATE_CREATED | “12-01-2016 11:10” | Date and time when transaction record is created in the DB. |

| | | | | |
|---|---|---|---|---|
| | | | | Format: [dd-mm-yyyy hh:mm] |
| **Error Response** | | | | |
| errCodeId | int | Table: ERROR_CODES<br>Field: ERR_CODE_ID | 12 | System Error Identifier in the STrxMSS DB. |
| Error | String | Table: ERROR_CODES<br>Field: ERROR_CODE | "Inbound Trx was not found for Btc address" | System Error Code |

**Examples:**

| Function Call | Java Request Example for POST | Response Example |
|---|---|---|
| *https://[Load Balancer Host Name]/StrxMssService/findInbTrxForBtcAddress* | JSONObject entity = new JSONObject();<br>entity.put("btcAddress", "16UwLL9Risc3QfPqBUvKofHmBQ7wMtjvM");<br>entity.put("daemonWalletId", "067e6162-3b6f-4ae2-a171-2470b63dff00"); | {"btcAddress": "16UwLL9Risc3QfPqBUvKofHmBQ7wMtjvM",<br>"inbTrxs": (<br>{"daemonTrxId": "067e6162-3b6f-4ae2-a171-2470b63dff00",<br>"btcAmount": "1.1",<br>"dateCreated": "02-11-2016 19:21"<br>},<br>{...} ...<br>)<br>} |
| | | Error example:<br>{ "errCodeId": 12, "error": "Inbound Trx was not found for Btc address" } |

## getNewBtcAddress

Function returns new Btc address for given Wallet or error's data in case system error.

| Request Parameters | Response Parameters | Response Type | Java Class (including package) | Java Method |
|---|---|---|---|---|
| *<daemonWalletId>, <externalPartMnmSeed> <isCompressedPubKey>* | <daemonWalletId>, <newBtcAddress><br><br>**Error Response**: <errCodeId>, <error> | `JSONObject` | | |

**Request Parameters:**

| Parameter | Java Type | Length | Required | STrxMSS DB mapping | Example | Description |
|---|---|---|---|---|---|---|

| | | | | | | |
|---|---|---|---|---|---|---|
| *daemonWalletId* | String | min 4 chars, max 60 chars | mandatory | Table: WALLETS Field: WALLET_ID Type: UUID | 067e6162-3b6f-4ae2-a171-2470b63dff00 | Wallet identifier in the STrxMSS DB |
| *externalPartMnmSeed* | String | min 5 chars, max 500 chars | mandatory | N/A | sdhsakdhsakjhd | User's part of mnemonic seed for this wallet |
| *isCompressedPubKey* | int | 0 or 1 only | mandatory | N/A | 0 | True (1) if Compressed Public Key is used for Btc Address creation otherwise false (0). False (0) by default |

**Response Parameters in JSONObject:**

| Parameter | Java Type | STrxMSS DB mapping | Example | Description |
|---|---|---|---|---|
| newBtcAddress | String | Table: SYSTEM_BTC_ADDRESSES Field: BTC_ADDRESS | 16UwLL9Risc3QfPqBUvKofHmBQ7wMtjvM | New Btc address is generated for this wallet |
| daemonWalletId | String | Table: WALLETS Field: WALLET_ID | 067e6162-3b6f-4ae2-a171-2470b63dff00 | Wallet identifier in the STrxMSS DB |
| **Error Response** | | | | |
| errCodeId | int | Table: ERROR_CODES Field: ERR_CODE_ID | 13 | System Error Identifier in the STrxMSS DB. |
| Error | String | Table: ERROR_CODES Field: ERROR_CODE | "Error in the creation of Btc address" | System Error Code |

**Examples:**

| Function Call | Java Request Example for POST | Response Example |
|---|---|---|
| *https://[Load Balancer Host Name]/StrxMssService/getNewBtcAddress* | JSONObject entity = new JSONObject(); entity.put("daemonWalletId", "067e6162-3b6f-4ae2-a171-2470b63dff00"); entity.put("externalPartMnmSeed", "16UwLL9"); | { "daemonWalletId":" 067e6162-3b6f-4ae2-a171-2470b63dff00", "newBtcAddress":"16UwLL9Risc3QfPqBUvKofHmBQ7wMtjvM" } |

| | | |
|---|---|---|
| | | Error example:<br>{ “errCodeId”: 13, “error”: “Error in the creation of Btc address” } |

### 5.1.4 Warm Storage Functions

#### lockWallet

Function creates “Warm Storage” transaction for given Wallet and locks Wallet till specified date or returns error’s data in case system error.

***Note**: “Warm Storage” solution will be developed in the future stages of project according to BIP-0065. lockWallet function will be updated. Current implementation will update only IS_LOCKED flag in the WALLETS table.*

| Request Parameters | Response Parameters | Response Type | Java Class (including package) | Java Method |
|---|---|---|---|---|
| *<daemonWalletId>*, *<dateToUnlock>* | <daemonWalletId>, <isLocked><br><br>**Error Response**:<br><errCodeId>, <error> | `JSONObject` | | |

**Request Parameters:**

| Parameter | Java Type | Length | Required | STrxMSS DB mapping | Example | Description |
|---|---|---|---|---|---|---|
| *daemonWalletId* | String | min 4 chars, max 60 chars | mandatory | Table: WALLETS<br>Field: WALLET_ID<br>Type: UUID | 067e6162-3b6f-4ae2-a171-2470b63dff00 | Wallet identifier in the STrxMSS DB |
| *dateToUnlock* | String | min 16 chars, max 16 chars | mandatory | Table: WALLETS<br>Field: DATE_TO_UNLOCK<br>Type: TIMESTAMP | “12-01-2016 11:10” | Date and time when Wallet should be unlocked. Format: [dd-mm-yyyy hh:mm] |

**Response Parameters in JSONObject:**

| Parameter | Java Type | STrxMSS DB mapping | Example | Description |
|---|---|---|---|---|
| daemonWalletId | String | Table: WALLETS<br>Field: WALLET_ID | 067e6162-3b6f-4ae2-a171-2470b63dff00 | Wallet identifier in the STrxMSS DB |

| | | | | |
|---|---|---|---|---|
| isLocked | int | Table: WALLETS<br>Field: IS_LOCKED | 1 | True (1) if Wallet is locked otherwise false (0). |
| **Error Response** | | | | |
| errCodeId | int | Table: ERROR_CODES<br>Field: ERR_CODE_ID | 14 | System Error Identifier in the STrxMSS DB. |
| Error | String | Table: ERROR_CODES<br>Field: ERROR_CODE | “Error of Locking Wallet” | System Error Code |

**Examples:**

| Function Call | Java Request Example for POST | Response Example |
|---|---|---|
| *https://[Load Balancer Host Name]/StrxMssService/lockWallet* | JSONObject entity = new JSONObject();<br>entity.put(“daemonWalletId”, “067e6162-3b6f-4ae2-a171-2470b63dff00”);<br>entity.put(“12-01-2016 11:10”); | {“ daemonWalletId”: “067e6162-3b6f-4ae2-a171-2470b63dff00”, “isLocked”: 1}<br><br>Error example:<br>{ “errCodeId”: 14, “error”: “Error of Locking Wallet” } |

### unlockWallet

Function unlocks given Wallet from “Warm Storage” or returns error’s data in case system error.

***Note**: This function is temporary. “Warm Storage” solution will be developed in the future stages of project according to BIP-0065. This function will be deleted after that.*

| Request Parameters | Response Parameters | Response Type | Java Class (including package) | Java Method |
|---|---|---|---|---|
| *<daemonWalletId>* | <daemonWalletId>, <isLocked><br><br>**Error Response**: <errCodeId>, <error> | JSONObject | | |

**Request Parameters:**

| Parameter | Java Type | Length | Required | STrxMSS DB mapping | Example | Description |
|---|---|---|---|---|---|---|
| *daemonWalletId* | String | min 4 chars, max 60 chars | mandatory | Table: WALLETS<br>Field: WALLET_ID<br>Type: UUID | 067e6162-3b6f-4ae2-a171-2470b63dff00 | Wallet identifier in the STrxMSS DB |

**Response Parameters in JSONObject:**

| Parameter | Java Type | STrxMSS DB mapping | Example | Description |
|---|---|---|---|---|
| daemonWalletId | String | Table: WALLETS<br>Field: WALLET_ID | 067e6162-3b6f-4ae2-a171-2470b63dff00 | Wallet identifier in the STrxMSS DB |
| isLocked | int | Table: WALLETS<br>Field: IS_LOCKED | 0 | True (1) if Wallet is locked otherwise false (0). |
| **Error Response** | | | | |
| errCodeId | int | Table: ERROR_CODES<br>Field: ERR_CODE_ID | 10 | System Error Identifier in the STrxMSS DB. |
| Error | String | Table: ERROR_CODES<br>Field: ERROR_CODE | “STrxMSS error” | System Error Code |

**Examples:**

| Function Call | Java Request Example for POST | Response Example |
|---|---|---|
| *https://[Load Balancer Host Name]/StrxMssService/lockWallet* | String input = “{\”daemonWalletId\”:\”067e6162-3b6f-4ae2-a171-2470b63dff00\”}”; | {“ daemonWalletId”: “067e6162-3b6f-4ae2-a171-2470b63dff00”, “isLocked”: 0}<br><br>Error example:<br>{ “errCodeId”: 10, “error”: “STrxMSS error” } |

## 5.1.5 Other Functions

### getErrorData

Function returns data of IntD System error by given error identifier or error’s data in case system error.

| Request Parameters | Response Parameters | Response Type | Java Class (including package) | Java Method |
|---|---|---|---|---|
| *<errCodeId>* | <errCodeId>,<br><errCode>,<br><errDescr>,<br>[subSystemAbbr]<br>**Error Response**:<br><errCodeId>, <error> | `JSONObject` | | |

**Request Parameters:**

| Parameter | Java Type | Length | Required | “shared_data” DB mapping | Example | Description |
|---|---|---|---|---|---|---|
| *errCodeId* | int | | mandatory | Table: INTDSYSTEM_ERROR_CODES<br>Field: ERR_CODE_ID<br>Type: int | 1 | Error code identity number. |

**Response Parameters in JSONObject:**

| Parameter | Java Type | “shared_data” DB mapping | Example | Description |
|---|---|---|---|---|
| errCodeId | int | Table: INTDSYSTEM_ERROR_CODES Field: ERR_CODE_ID | 7 | Error code identity number. |
| errCode | String | Table: INTDSYSTEM_ERROR_CODES Field: ERROR_CODE | “Transaction send error” | Error code. |
| errDescr | String | Table: INTDSYSTEM_ERROR_CODES Field: ERROR_DESCR | “STrxMSS error in the sending of transaction to blockchain” | Error description. |
| subSystemAbbr | String | Table: INTDSYSTEM_ERROR_CODES Field: SUBSYSTEM_ABBR | “STrxMSS” | SubSystem abbreviation. Value can be empty or null. |
| **Error Response** | | | | |
| errCodeId | int | Table: ERROR_CODES Field: ERR_CODE_ID | 15 | System Error Identifier in the STrxMSS DB. |
| Error | String | Table: ERROR_CODES Field: ERROR_CODE | “Data of system error was not found” | System Error Code |

**Examples:**

| Function Call | Java Request Example for POST | Response Example |
|---|---|---|
| *https://[Load Balancer Host Name]/StrxMssService/getErrorData* | String input = “{\”errCodeId\”:1}”; | {“errCodeId”: 7, “errCode”: “Transaction send error”, “errDescr”: “STrxMSS error in the sending of transaction to blockchain”, “subSystemAbbr”: “STrxMSS” } |
| | | Error example: { “errCodeId”: 15, “error”: “Data of system error was not found” } |

## getRejectionMsgData

Function returns data of blockchain rejection message by given message identifier or error’s data in case system error.

| Request Parameters | Response Parameters | Response Type | Java Class (including package) | Java Method |
|---|---|---|---|---|
| *<rejectMsgId>* | <rejectMsgId>, <rejectMsgCode>, <rejectMsgDescr>, | `JSONObject` | | |

| | | | | |
|---|---|---|---|---|
| | <rejectCategory> | | | |
| | **Error Response**: <errCodeId>, <error> | | | |

**Request Parameters:**

| Parameter | Java Type | Length | Required | “shared_data” DB mapping | Example | Description |
|---|---|---|---|---|---|---|
| *rejectMsgId* | int | | mandatory | Table: BTC_REJECTION_MSG<br>Field: REJECT_MSG_ID<br>Type: int | 1 | Rejection message identity number. |

**Response Parameters in JSONObject:**

| Parameter | Java Type | “shared_data” DB mapping | Example | Description |
|---|---|---|---|---|
| rejectMsgId | int | Table: BTC_REJECTION_MSG<br>Field: REJECT_MSG_ID | 1 | Rejection message identity number. |
| rejectMsgCode | int | Table: BTC_REJECTION_MSG<br>Field: REJECT_MSG_CODE | 10 | Rejection message code. |
| rejectMsgDescr | String | Table: BTC_REJECTION_MSG<br>Field: REJECT_MSG_DESCR | “Block is invalid for some reason (invalid proof-of-work, invalid signature, etc)” | Rejection message description. |
| rejectCategory | String | Table: BTC_REJECTION_MSG<br>Field: REJECTION_CATEGORY | “Block” | Rejection message category |
| **Error Response** | | | | |
| errCodeId | int | Table: ERROR_CODES<br>Field: ERR_CODE_ID | 16 | System Error Identifier in the STrxMSS DB. |
| Error | String | Table: ERROR_CODES<br>Field: ERROR_CODE | “Rejection message was not found” | System Error Code |

**Examples:**

| Function Call | Java Request Example for POST | Response Example |
|---|---|---|
| *https://[Load Balancer Host Name]/StrxMssService/getErrorData* | String input = “{\”rejectMsgId\”:1}”; | {“rejectMsgId”: 1, “rejectMsgCode”: 10, “rejectMsgDescr”: “Block is invalid for some reason (invalid proof-of-work, invalid signature, etc)”, “rejectCategory”: “Block”<br>} |
| | | Error example:<br>{ “errCodeId”: 16, “error”: “Rejection message was not found” } |

## 5.2 Accounting Transaction Management SubSystem Interface

This point can be done in the scope of future development. Will need some researching activity.

## 5.3 Bank Transaction Management SubSystem Interface

This point can be done in the scope of future development. Will need some researching activity.

## 5.4 Exchange Transaction Management SubSystem Interface

This point can be done in the scope of future development. Will need some researching activity.

## 5.5 Message Transaction Management SubSystem Interface

This point can be done in the scope of future development. Will need some researching activity.

## 5.6 Contracts Management SubSystem Interface

This point can be done in the scope of future development. Will need some researching activity.

## 5.7 Daemon Core System Interface

iDaemon system will use various functions as RPC from the FOS Core Daemon component. The Daemon RPCs [2.16] will be called via a Java Wrapper which is RESFul Java Web Service.

Input parameters, return values and description of these RPCs and Java Wrapper are described below.

***Note:** This section will be updated as development progresses through later phases. Currently, only RPCs related to **single signature transactions and P2PKH addresses** are documented.*

*The wallet related RPCs are not documented here as the open source wallet functionality will not be used. DeMorgan will develop custom wallet software that will use iDaemon for network access and other basic functionality.*

### 5.7.1 Description of commonly used data structures, definitions in bitcoin core RPCs

**Outpoint**: The data structure used to refer to a particular transaction output, consisting of a 32-byte TXID and a 4-byte output index number (vout).

**Output, Transaction Output, TxOut**: An output in a transaction which contains two fields: a value field for transferring zero or more Satoshis and a scriptPubKey for indicating what conditions must be fulfilled for those Satoshis to be further spent.

**Serialized transaction**: Complete transactions in their binary format; often represented using hexadecimal. Sometimes called raw format because of the various Bitcoin Core commands with “raw” in their names.

**Serialized block**: A complete block in its binary format—the same format used to calculate total block byte size; often represented using hexadecimal.

**RPC Byte order**: A hash digest displayed with the byte order reversed; used in Bitcoin Core RPCs, many block explorers, and other software.

### 5.7.2 Remote Procedure Calls

#### Transactions

Use these RPCs to create, sign, send and get information about raw transactions.

***Note:** Although FOS daemon RPCs will be used to sign a transaction, RPCs related to key generation will not be used. DeMorgan will develop custom implementation for key generation.*

#### Createrawtransaction

creates an unsigned serialized transaction (complete transaction in their binary format) that spends a previous output to a new output with a P2PKH or P2SH address. The transaction is not transmitted to the network. The transaction's inputs are not signed.

**Parameters:**

| Param No. | Name | Type | Presence | Description |
|---|---|---|---|---|
| 1 | Outpoints | Array | Required (exactly 1) | An array of outpoints. Each outpoint is an unspent output with 2 arrays as described below. |
| | • Outpoint | Object | Required (1 or more) | An object describing a particular unspent outpoint. Each outpoint is an object with 2 arrays as described below. |
| | - TXID | String (hex) | Required (exactly 1) | The TXID of the outpoint encoded as hex in RPC byte order. 32 bytes |
| | - vout | Number (int) | Required (exactly 1) | The output index number (vout) of the outpoint; the first output in a transaction is index 0. 4 bytes |
| 2 | Outputs | Object | Required (exactly 1) | The addresses and amounts to pay |
| | • Address/Amount | String : number (float) | Required (1 or more) | A key/value pair with the address to pay as a string (key) and the amount to pay that address (value) in bitcoins |

**Return:**

| Result No. | Name | Type | Presence | Description |
|---|---|---|---|---|
| 1 | result | string | Required (exactly 1) | The resulting unsigned raw transaction in serialized transaction format encoded as hex. If the transaction couldn't be generated, this will be set to JSON null and the JSON-RPC error field may contain an error message |

**Usage and Examples:**

createrawtransaction [{"txid":"id","vout":n},...] {"address":amount,...}

***Arguments:***

```
1. "transactions"       (string, required) A json array of json objects
    [
      {
        "txid":"id",  (string, required) The transaction id
        "vout":n       (numeric, required) The output number
      }
      ,...
    ]
2. "addresses"          (string, required) a json object with addresses as keys
 and amounts as values
   {
     "address": x.xxx   (numeric, required) The key is the bitcoin address, the
 value is the btc amount
     ,...
   }
```

***Result:***

"transaction"          (string) hex string of the transaction

***Examples:***

> bitcoin-cli createrawtransaction "[{\"txid\":\"myid\",\"vout\":0}]" "{\"addres
s\":0.01}"

> curl –user myusername –data-binary '{"jsonrpc": "1.0", "id":"curltest", "met hod": "createrawtransaction", "params": ["[{\"txid\":\"myid\",\"vout\":0}]", "{\ "address\":0.01}"] }' –H 'content-type: text/plain;' http://127.0.0.1:8332/

Example from Bitcoin Core 0.10.0 testnet:

```
bitcoin-cli -testnet createrawtransaction ‘’’

  [

    {

      “txid”: “1eb590cd06127f78bf38ab4140c4cdce56ad9eb8886999eb898ddf4d3b28a91d”,

      “vout” : 0

    }

  ]’’’ ‘{ “mgnucj8nYqdrPFh2JfZSB1NmUThUGnmsqe”: 0.13 }’
```

Result (wrapped):

```
01000000011da9283b4ddf8d89eb996988b89ead56cecdc44041ab38bf787f12\

06cd90b51e0000000000ffffffff01405dc600000000001976a9140dfc8bafc8\

419853b34d5e072ad37d1a5159f58488ac00000000
```

### decoderawtransaction

decodes a serialized transaction hex string into a JSON object describing the transaction.

**Parameters:**

| Param No. | Name | Type | Presence | Description |
|---|---|---|---|---|
| 1 | serialized transaction | String (hex) | Required (exactly 1) | The transaction to decode in serialized transaction format. |

**Return:**

| Result No. | Name | Type | Presence | Description |
|---|---|---|---|---|
| 1 | result | object | Required (exactly 1) | An object describing the decoded transaction, or JSON null if the transaction could not be decoded. |
| | • TXID | String (hex) | Required (exactly 1) | The transaction's TXID encoded as hex in RPC byte order |
| | • version | Number (int) | Required (exactly 1) | The transaction format version number |
| | • locktime | Number (int) | Required (exactly 1) | The transaction's locktime: either a Unix epoch date or block height; see the Locktime parsing rules |
| | • vin | array | Required (exactly 1) | An array of objects with each object being an input vector (vin) for this transaction (described below). Input objects will have the same order within the array as they have in the transaction, so the first input listed will be input 0 |
| | - input | object | Required (1 or more) | An object describing one of this transaction's inputs. May be a regular input or a coinbase. Object should contain following members: |

| | | | | |
|---|---|---|---|---|
| | ○ txid | String (hex) | Optional (0 or 1) | The transaction id |
| | ○ vout | Number (int) | Optional (0 or 1) | The output number of the outpoint being spent. The first output in a transaction has an index of 0. Not present if this is a coinbase transaction |
| | ○ scriptSig | Json Object | Optional (0 or 1) | An object describing the signature script of this input. Not present if this is a coinbase transaction |
| | ▪ asm | String (asm) | Required (exactly 1) | The signature script in decoded form with non-data-pushing op codes listed |
| | ▪ hex | String (hex) | Required (exactly 1) | The signature script encoded as hex |
| | ○ coinbase | String (hex) | Optional (0 or 1) | The coinbase (similar to the hex field of a scriptSig) encoded as hex. Only present if this is a coinbase transaction |
| | ○ sequence | Number (int) | Required (exactly 1) | The input sequence number |
| | • vout | Array of json objects | Required (exactly 1) | An array of objects each describing an output vector (vout) for this transaction. Output objects will have the same order within the array as they have in the transaction, so the first output listed will be output 0 |
| | - Output | Json Object | Required (1 or more) | An object describing one of this transaction’s outputs |
| | ○ value | Number (float) | Required (exactly 1) | The number of bitcoins paid to this output. May be 0. |
| | ○ n | Number (int) | Required (exactly 1) | The output index number of this output |
| | ○ scriptPubKey | Json object | Required (exactly 1) | An object describing the pubkey script |
| | ▪ asm | String (asm) | Required (exactly 1) | The pubkey script in decoded form with non-data-pushing op codes listed |

| | | | | |
|---|---|---|---|---|
| | ▪ hex | String (hex) | Required (exactly 1) | The pubkey script in decoded form with non-data-pushing op codes listed |
| | ▪ reqSigs | Number (int) | Optional (0 or 1) | This field is 1 for now, for single signature transactions.It may be greater than 1 for bare multisig. This value will not be returned for nulldata or nonstandard script types (see the type key below) |
| | ▪ type | string | Optional (0 or 1) | The type of script. This will be pubkeyhash for now.<br>• pubkey for a P2PK script<br>• pubkeyhash for a P2PKH script<br>• scripthash for a P2SH script<br>• multisig for a bare multisig script<br>• nulldata for nulldata scripts<br>• nonstandard for unknown scripts |
| | ▪ addresses | Json array of strings | Optional (0 or 1) | The P2PKH or P2SH addresses used in this transaction, or the computed P2PKH address of any pubkeys in this transaction. This array will not be returned for nulldata or nonstandard script types |
| | - address | string | Required (1 or more) | A P2PKH or P2SH address |

**Usage and Examples:**

***Arguments:***
1. “hex”  (string, required) The transaction hex string

***Result:***
```
{
  “txid” : “id”,        (string) The transaction id
  “version” : n,        (numeric) The version
  “locktime” : ttt,     (numeric) The lock time
  “vin” : [             (array of json objects)
```

```
  {
    “txid”: “id”,    (string) The transaction id
    “vout”: n,       (numeric) The output number
    “scriptSig”: {    (json object) The script
      “asm”: “asm”,  (string) asm
      “hex”: “hex”   (string) hex
    },
    “sequence”: n    (numeric) The script sequence number
  }
  ,...
],
“vout” : [           (array of json objects)
  {
    “value” : x.xxx,          (numeric) The value in btc
    “n” : n,                  (numeric) index
    “scriptPubKey” : {        (json object)
      “asm” : “asm”,          (string) the asm
      “hex” : “hex”,          (string) the hex
      “reqSigs” : n,          (numeric) The required sigs
      “type” : “pubkeyhash”,  (string) The type, eg ‘pubkeyhash’
      “addresses” : [         (json array of string)
        “12tvKAXCxZjSmdNbao16dKXC8tRWfcF5oc”   (string) bitcoin address
        ,...
      ]
    }
  }
  ,...
],
}
```

***Examples:***

> bitcoin-cli decoderawtransaction “hexstring”
> curl –user myusername –data-binary ‘{“jsonrpc”: “1.0”, “id”:”curltest”, “met
hod”: “decoderawtransaction”, “params”: [“hexstring”] }’ –H ‘content-type: text/
plain;’ http://127.0.0.1:8332/

Example from Bitcoin Core 0.10.0 testnet:

Decode a signed one-input, three-output transaction:

```
bitcoin-cli -testnet decoderawtransaction 0100000001268a9ad7bfb2\
1d3c086f0ff28f73a064964aa069ebb69a9e437da85c7e55c7d7000000006b48\
```

```
3045022100ee69171016b7dd218491faf6e13f53d40d64f4b40123a2de52560f\
eb95de63b902206f23a0919471eaa1e45a0982ed288d374397d30dff541b2dd4\
5a4c3d0041acc0012103a7c1fd1fdec50e1cf3f0cc8cb4378cd8e9a2cee8ca9b\
3118f3db16cbbcf8f326ffffffff0350ac6002000000001976a91456847befbd\
2360df0e35b4e3b77bae48585ae06888ac80969800000000001976a9142b1495\
0b8d31620c6cc923c5408a701b1ec0a02088ac002d3101000000001976a9140d\
fc8bafc8419853b34d5e072ad37d1a5159f58488ac00000000
```

Result:


```
{
    "txid" : "ef7c0cbf6ba5af68d2ea239bba709b26ff7b0b669839a63bb01c2cb8e8de481e",
    "version" : 1,
    "locktime" : 0,
    "vin" : [
        {
            "txid" : "d7c7557e5ca87d439e9ab6eb69a04a9664a0738ff20f6f083c1db2bfd79a8a26",
            "vout" : 0,
            "scriptSig" : {
                "asm" :
"3045022100ee69171016b7dd218491faf6e13f53d40d64f4b40123a2de52560feb95de63b902206f23a0919471eaa1e45a0982ed
288d374397d30dff541b2dd45a4c3d0041acc001
03a7c1fd1fdec50e1cf3f0cc8cb4378cd8e9a2cee8ca9b3118f3db16cbbcf8f326",
                "hex" :
"483045022100ee69171016b7dd218491faf6e13f53d40d64f4b40123a2de52560feb95de63b902206f23a0919471eaa1e45a0982
ed288d374397d30dff541b2dd45a4c3d0041acc0012103a7c1fd1fdec50e1cf3f0cc8cb4378cd8e9a2cee8ca9b3118f3db16cbbcf
8f326"
            },
            "sequence" : 4294967295
        }
    ],
    "vout" : [
        {
            "value" : 0.39890000,
            "n" : 0,
            "scriptPubKey" : {
                "asm" : "OP_DUP OP_HASH160 56847befbd2360df0e35b4e3b77bae48585ae068 OP_EQUALVERIFY
OP_CHECKSIG",
                "hex" : "76a91456847befbd2360df0e35b4e3b77bae48585ae06888ac",
```

```
                "reqSigs" : 1,
                "type" : "pubkeyhash",
                "addresses" : [
                    "moQR7i8XM4rSGoNwEsw3h4YeuduuP6mxw7"
                ]
            }
        },
        {
            "value" : 0.10000000,
            "n" : 1,
            "scriptPubKey" : {
                "asm" : "OP_DUP OP_HASH160 2b14950b8d31620c6cc923c5408a701b1ec0a020 OP_EQUALVERIFY
OP_CHECKSIG",
                "hex" : "76a9142b14950b8d31620c6cc923c5408a701b1ec0a02088ac",
                "reqSigs" : 1,
                "type" : "pubkeyhash",
                "addresses" : [
                    "mjSk1Ny9spzU2fouzYgLqGUD8U41iR35QN"
                ]
            }
        },
        {
            "value" : 0.20000000,
            "n" : 2,
            "scriptPubKey" : {
                "asm" : "OP_DUP OP_HASH160 0dfc8bafc8419853b34d5e072ad37d1a5159f584 OP_EQUALVERIFY
OP_CHECKSIG",
                "hex" : "76a9140dfc8bafc8419853b34d5e072ad37d1a5159f58488ac",
                "reqSigs" : 1,
                "type" : "pubkeyhash",
                "addresses" : [
                    "mgnucj8nYqdrPFh2JfZSB1NmUThUGnmsqe"
                ]
            }
        }
    ]
```

```
}
```

## signrawtransaction

signs a transaction in the serialized transaction format using private keys stored in the wallet or provided in the call.

**Parameters:**

| Param No. | Name | Type | Presence | Description |
|---|---|---|---|---|
| 1 | Transaction | String (hex) | Required (exactly 1) | The transaction to sign |
| 2 | Unspent outputs | Json Array of json objects | Optional (0 or 1) | Json array of previous dependent transaction outputs. |
| | • Output | Json Object | Optional (0 or 1) | An output being spent |
| | ○ txid | String (hex) | Required (exactly 1) | The TXID of the transaction the output appeared in. |
| | ○ vout | Number (int) | Required (exactly 1) | The index number of the output (vout) as it appeared in its transaction, with the first output being 0 |
| | ○ scriptPubKey | String (hex) | Required (exactly 1) | The output’s pubkey script encoded as hex |
| | ○ redeemScript | String (hex) | Optional (0 or 1) | Not needed for single signature transactions. If the pubkey script was a script hash, this must be the corresponding redeem script |
| 3 | Private keys | Json array | Optional (0 or 1) | Json array of base58-encoded private keys for signing |
| | • Key | String (base58) | Required (1 or more) | A private key in base58check format to use to create a signature for this transaction |
| 4 | • SigHash | string | Optional (0 or 1). Default = ALL | Signature hash type. Must be one of: `ALL`, `NONE`, `SINGLE`, `ALL\|ANYONECANPAY`, `NONE\|ANYONECANPAY`, and `SINGLE\|ANYONECANPAY` |

**Return:**

| Result No. | Name | Type | Presence | Description |
|---|---|---|---|---|
| 1 | result | object | Required (exactly 1) | The results of the signature |
| | Hex | String (hex) | Required (exactly 1) | Raw transaction with signatures inserted. If no signatures were made, this will be the same transaction provided in parameter #1 |
| | Complete | Bool | Required (exactly 1) | True if transaction if fully signed; false if more signatures are required. |

**Usage and Examples:**

***Arguments:***

1. "hexstring" (string, required) The hex string of the raw transaction
2. "prevtxs" (string optional) An json array of previous dependent transaction outputs
   [ (json array of json objects, or 'null' if none provided)
   {
   "txid":"id", (string, required) The transaction id
   "vout":n, (numeric, required) The output number
   "scriptPubKey": "hex", (string, required) script key, "hex" from previous Trx: (…, vout:[…,"scriptPubKey:{.., "hex": value, ..} ..]" )
   "redeemScript": "hex" (string, required) redeem script if the funds is spending from multi-sig btc address, otherwise null
   }
   ,…
   ]
3. "privatekeys" (string, optional) A json array of base58-encoded private keys for signing
   [ (json array of strings, or 'null' if none provided)
   "privatekey" (string) private key in base58-encoding
   ,…
   ]
4. "sighashtype" (string, optional, default=ALL) The signature hash type. Must be one of
   "ALL"
   "NONE"
   "SINGLE"
   "ALL|ANYONECANPAY"

“NONE|ANYONECANPAY”
“SINGLE|ANYONECANPAY”

***Result:***

{
“hex”: “value”, (string) The raw transaction with signature(s) (hex-encoded string)
“complete”: n (numeric) if transaction has a complete set of signature (0 if not)
}

***Examples:***

Create a transaction
> bitcoin-cli signrawtransaction “myhex”

Sign the transaction, and get back the hex
> bitcoin-cli signrawtransaction “myhex”

As a json rpc call
> curl –user myusername –data-binary ‘{“jsonrpc”: “1.0”, “id”:”curltest”, “method”: “signrawtransaction”, “params”: [“myhex”] }’ –H ‘content-type: text/plain; ‘ http://127.0.0.1:8332/

Example from Bitcoin Core 0.10.0 testnet:

```
bitcoin-cli -testnet signrawtransaction 01000000011da9283b4ddf8d\
89eb996988b89ead56cecdc44041ab38bf787f1206cd90b51e0000000000ffff\
ffff01405dc600000000001976a9140dfc8bafc8419853b34d5e072ad37d1a51\
59f58488ac00000000
```

Result:

```
{
    “hex” :
“01000000011da9283b4ddf8d89eb996988b89ead56cecdc44041ab38bf787f1206cd90b51e000000006a47304402200
ebea9f630f3ee35fa467ffc234592c79538ecd6eb1c9199eb23c4a16a0485a20220172ecaf6975902584987d295b8ddd
f8f46ec32ca19122510e22405ba52d1f13201210256d16d76a49e6c8e2edc1c265d600ec1a64a45153d45c29a2fd0228
c24c3a524ffffffff01405dc600000000001976a9140dfc8bafc8419853b34d5e072ad37d1a5159f58488ac00000000”
,
    “complete” : true
```

```
}
```

### sendrawtransaction

validates a transaction, serializes and broadcasts it to the peer-to-peer network.

**Parameters:**

| Param No. | Name | Type | Presence | Description |
|---|---|---|---|---|
| 1 | Transaction | String (hex) | Required (exactly 1) | The transaction to broadcast encoded as hex |
| 2 | Allow High Fees | Bool | Optional (0 or 1). Default = false | Set to true to allow the transaction to pay a high transaction fee. Set to false (the default) to prevent Bitcoin Core from broadcasting the transaction if it includes a high fee |

**Return:**

| Result No. | Name | Type | Presence | Description |
|---|---|---|---|---|
| 1 | result | Null/string(hex) | Required (exactly 1) | If the transaction was accepted by the node for broadcast, this will be the TXID of the transaction encoded as hex in RPC byte order. If the transaction was rejected by the node, this will set to null, the JSON-RPC error field will be set to a code, and the JSON-RPC message field may contain an informative error message |

**Usage and Examples:**

***Arguments:***

1. "hexstring"    (string, required) The hex string of the raw transaction)
2. allowhighfees    (187ubscri, optional, default=false) Allow high fees

***Result:***

"hex"    (string) The transaction hash in hex

***Examples:***

Create a transaction
> bitcoin-cli createrawtransaction “[{\”txid\” : \”mytxid\”,\”vout\”:0}]” “{\”my address\”:0.01}”
Sign the transaction, and get back the hex
> bitcoin-cli signrawtransaction “myhex”

Send the transaction (signed hex)
> bitcoin-cli sendrawtransaction “signedhex”

As a json rpc call
> curl –user myusername –data-binary ‘{“jsonrpc”: “1.0”, “id”:”curltest”, “method”: “sendrawtransaction”, “params”: [“signedhex”] }’ –H ‘content-type: text/plain;’ [http://127.0.0.1:8332/](http://127.0.0.1:8332/)

Example from Bitcoin Core 0.10.0 testnet:

```
bitcoin-cli -testnet sendrawtransaction 01000000011da9283b4ddf8d\
89eb996988b89ead56cecdc44041ab38bf787f1206cd90b51e000000006a4730\
4402200ebea9f630f3ee35fa467ffc234592c79538ecd6eb1c9199eb23c4a16a\
0485a20220172ecaf6975902584987d295b8dddf8f46ec32ca19122510e22405\
ba52d1f13201210256d16d76a49e6c8e2edc1c265d600ec1a64a45153d45c29a\
2fd0228c24c3a524ffffffff01405dc600000000001976a9140dfc8bafc84198\
53b34d5e072ad37d1a5159f58488ac00000000
```

Result:

```
f5a5ce5988cc72b9b90e8d1d6c910cda53c88d2175177357cc2f2cf0899fbaad
```

### getrawtransaction

gets a hex-encoded serialized transaction or a JSON object describing the transaction. By default, Bitcoin Core only stores complete transaction data for UTXOs and your own transactions, so the RPC may fail on historic transactions unless you use the non-default txindex=1 in your Bitcoin Core startup settings.

***Note:*** *Keep default txindex. We will be using this rpc to get decoded transaction (see Param #2).*

**Parameters:**

| Param No. | Name | Type | Presence | Description |
|---|---|---|---|---|
| 1 | Txid | String (hex) | Required (exactly 1) | The txid of the transaction to get. |
| 2 | Verbose | Number (int) | Optional (0 or 1). Default = 0 | Set to 1 to return a decoded transaction, 0 (default) for serialized transaction. |

**Return:**

Null if transaction not found

Serialized transaction if verbose = 0 (not described here)

Decoded transaction if verbose = 1, as described below:

| Result No. | Name | Type | Presence | Description |
|---|---|---|---|---|
| 1 | result | Object | Required (exactly 1) | If the transaction was found, this will be an object describing it |
| | • Txid | String(hex) | Required (exactly 1) | The transaction's id |
| | • Version | Number (int) | Required (exactly 1) | The transaction format version number |
| | • Locktime | Number (int) | Required (exactly 1) | The transaction's locktime |
| | • Vin | Array of input objects | Required (exactly 1) | An array of objects with each object being an input vector (vin) for this transaction. Input objects will have the same order within the array as they |

| | | | | |
|---|---|---|---|---|
| | | | | have in the transaction, so the first input listed will be input 0 |
| | - Input | object | Required (1 or more) | An object describing one of this transaction’s inputs. May be a regular input or a coinbase |
| | ○ Txid | String | Optional (0 or 1) | The TXID of the outpoint being spent, encoded as hex in RPC byte order. Not present if this is a coinbase transaction |
| | ○ Vout | Number (int) | Optional (0 or 1) | The output index number (vout) of the outpoint being spent. The first output in a transaction has an index of 0. Not present if this is a coinbase transaction |
| | ○ scriptSig | Object | Optional (0 or 1) | An object describing the signature script of this input (described below). Not present if this is a coinbase transaction |
| | ▪ asm | String | Required (exactly 1) | The signature script in decoded form with non-data-pushing op codes listed |
| | ▪ hex | String (hex) | Required (exactly 1) | The signature script encoded as hex |
| | ○ Coinbase | String (hex) | Optional (0 or 1) | The coinbase (similar to the hex field of a scriptSig) encoded as hex. Only present if this is a coinbase transaction |
| | ○ Sequence | Number (int) | Required (exactly 1) | The input sequence number |
| | • vout | Array of output objects | Required (exactly 1) | An array of objects each describing an output vector (vout) for this transaction. Output objects will have the same order within the array as they have in the transaction, so the first |

| | | | | |
|---|---|---|---|---|
| | | | | output listed will be output 0 |
| | - Output | Object | Required (1 or more) | An object describing one of this transaction’s outputs |
| | ○ value | Number (float) | Required (exactly 1) | The number of bitcoins paid to this output. May be 0 |
| | ○ n | Number (int) | Required (exactly 1) | The output index number of this output within this transaction |
| | ○ scriptPubKey | Object | Required (exactly 1) | An object describing the pubkey script |
| | ▪ Asm | String | Required (exactly 1) | The pubkey script in decoded form with non-data-pushing op codes listed |
| | ▪ Hex | String (hex) | Required (exactly 1) | The pubkey script encoded as hex |
| | ▪ reqSigs | Number (int) | Optional (0 or 1) | The number of signatures required; this is always 1 for P2PK, P2PKH, and P2SH (including P2SH multisig because the redeem script is not available in the pubkey script). It may be greater than 1 for bare multisig. This value will not be returned for nulldata or nonstandard script types (see the type key below) |
| | ▪ type | String | Optional (0 or 1) | The type of script. This will be one of the following:<br>• pubkey for a P2PK script<br>• pubkeyhash for a P2PKH script<br>• scripthash for a P2SH script<br>• multisig for a bare multisig script<br>• nulldata for nulldata scripts<br>• nonstandard for unknown scripts |
| | ➢ Addresses | Array of strings | Optional (0 or 1) | P2PKH or P2SH addresses used in this transaction. |

| | | | | |
|---|---|---|---|---|
| | | | | This array will not be returned for nulldata or nonstandard script types |
| | ✓ address | | Required (1 or more) | A P2PKH or P2SH address |
| | • Blockhash | String (hex) | Optional (0 or 1) | If the transaction has been included in a block on the local best block chain, this is the hash of that block encoded as hex in RPC byte order |
| | • Confirmations | Number (int) | Required (exactly 1) | If the transaction has been included in a block on the local best block chain, this is how many confirmations it has. Otherwise, this is 0 |
| | • Time | Number (int) | Optional (0 or 1) | If the transaction has been included in a block on the local best block chain, this is the block header time of that block (may be in the future) |
| | • blocktime | Number (int) | Optional (0 or 1) | This field is currently identical to the time field described above |

**Usage and Examples:**

***Arguments:***
1. "txid"  (string, required) The transaction id
2. verbose  (numeric, optional, default=0) If 0, return a string, other ret
urn a json object

Result (if verbose is not set or set to 0):
"data"  (string) The serialized, hex-encoded data for 'txid'

***Result (if verbose > 0):***
{
 "hex" : "data",  (string) The serialized, hex-encoded data for 'txid'
 "txid" : "id",  (string) The transaction id (same as provided)
 "version" : n,  (numeric) The version

```
  "locktime" : ttt,      (numeric) The lock time
  "vin" : [             (array of json objects)
    {
      "txid": "id",    (string) The transaction id
      "vout": n,        (numeric)
      "scriptSig": {     (json object) The script
        "asm": "asm",  (string) asm
        "hex": "hex"   (string) hex
      },
      "sequence": n      (numeric) The script sequence number
    }
    ,...
  ],
  "vout" : [            (array of json objects)
    {
      "value" : x.xxx,          (numeric) The value in btc
      "n" : n,                 (numeric) index
      "scriptPubKey" : {         (json object)
        "asm" : "asm",         (string) the asm
        "hex" : "hex",         (string) the hex
        "reqSigs" : n,          (numeric) The required sigs
        "type" : "pubkeyhash",  (string) The type, eg 'pubkeyhash'
        "addresses" : [          (json array of string)
          "bitcoinaddress"       (string) bitcoin address
          ,...
        ]
      }
    }
    ,...
  ],
  "blockhash" : "hash",   (string) the block hash
  "confirmations" : n,      (numeric) The confirmations
  "time" : ttt,            (numeric) The transaction time in seconds since epoc
h (Jan 1 1970 GMT)
  "blocktime" : ttt        (numeric) The block time in seconds since epoch (Jan
 1 1970 GMT)
}
```

***Examples:***

```
> bitcoin-cli getrawtransaction "mytxid"
> bitcoin-cli getrawtransaction "mytxid" 1
> curl –user myusername –data-binary '{"jsonrpc": "1.0", "id":"curltest", "met
```

hod": "getrawtransaction", "params": ["mytxid", 1] }' –H 'content-type: text/pla
in;' http://127.0.0.1:8332/

Example from Bitcoin Core 0.10.0 testnet:

```
bitcoin-cli -testnet getrawtransaction \
  ef7c0cbf6ba5af68d2ea239bba709b26ff7b0b669839a63bb01c2cb8e8de481e
```

Result:

```
0100000001268a9ad7bfb21d3c086f0ff28f73a064964aa069ebb69a9e437da8\
5c7e55c7d7000000006b483045022100ee69171016b7dd218491faf6e13f53d4\
0d64f4b40123a2de52560feb95de63b902206f23a0919471eaa1e45a0982ed28\
8d374397d30dff541b2dd45a4c3d0041acc0012103a7c1fd1fdec50e1cf3f0cc\
8cb4378cd8e9a2cee8ca9b3118f3db16cbbcf8f326ffffffff0350ac60020000\
00001976a91456847befbd2360df0e35b4e3b77bae48585ae06888ac80969800\
000000001976a9142b14950b8d31620c6cc923c5408a701b1ec0a02088ac002d\
3101000000001976a9140dfc8bafc8419853b34d5e072ad37d1a5159f58488ac\
00000000
```

## Network

Use these RPCs to communicate with the P2P network.

### Addnode

attempts to add or remove a node from the addnode list, or to try a connection to a node once.

**Parameters:**

| Param No. | Name | Type | Presence | Description |
|---|---|---|---|---|
| 1 | Node | String | Required (exactly 1) | The node to add as a string in the form of <IP address>:<port>. The IP address may be a hostname resolvable through DNS, an Ipv4 address, an Ipv4-as-Ipv6 address, or an Ipv6 address |
| 2 | Command | String | Required (exactly 1) | What to do with the IP address above. Options are:<br>• add to add a node to the addnode list. This will not connect immediately if the outgoing connection slots are full<br>• remove to remove a node from the list. If currently connected, this will disconnect immediately<br>• onetry to immediately attempt connection to the node even if the outgoing connection slots are full; this will only attempt the connection once |

**Return:**

| Result No. | Name | Type | Presence | Description |
|---|---|---|---|---|
| 1 | result | null | Required (exactly 1) | Always JSON null whether the node was added, removed, tried-and-connected, or tried-and-not-connected. The JSON-RPC error field will be set only if |

| | | | | |
|---|---|---|---|---|
| | | | | you try removing a node that is not on the addnodes list |

**Usage and Examples:**

***Arguments:***
1. "node"     (string, required) The node (see getpeerinfo for nodes)
2. "command"  (string, required) 'add' to add a node to the list, 'remove' to re move a node from the list, 'onetry' to try a connection to the node once

***Examples:***
> bitcoin-cli addnode "192.168.0.6:8333" "onetry"
> curl –user myusername –data-binary '{"jsonrpc": "1.0", "id":"curltest", "met hod": "addnode", "params": ["192.168.0.6:8333", "onetry"] }' –H 'content-type: t ext/plain;' http://127.0.0.1:8332/

Example from Bitcoin Core 0.10.0 testnet:

```
bitcoin-cli -testnet addnode 192.0.2.113:18333 onetry
```

Result (no output from bitcoin-cli because result is set to null).

### getaddednodeinfo

returns information about the given added node, or all added nodes (except onetry nodes). Only nodes which have been manually added using the addnode RPC will have their information displayed.

**Parameters:**

| Param No. | Name | Type | Presence | Description |
|---|---|---|---|---|
| 1 | Details | Bool | Required (exactly 1) | Set to true to display detailed information about each added node; set to false to only display the IP address or hostname and port added |
| 2 | Node | String | Optional (0 or 1) | The node to get information about in the same <IP address>:<port> format as the addnode RPC. If this parameter is not provided, information about all added nodes will be returned |

**Return:**

| Result No. | Name | Type | Presence | Description |
|---|---|---|---|---|
| 1 | result | array | Required (exactly 1) | An array containing objects describing each added node. If no added nodes are present, the array will be empty. Nodes added with onetry will not be returned |
| | • Added node | Object | Optional (0 or 1) | An object containing details about a single added node |
| | - Addednode | String | Required (exactly 1) | An added node in the same <IP address>:<port> format as used in the addnode RPC. This element is present for any added node whether or not the Details parameter was set to true |
| | - Connected | Bool | Optional (0 or 1) | If the Details parameter was set to true, this will be set to true if the node is currently connected and false if it is not |

| | | | | |
|---|---|---|---|---|
| | - Addresses | Array of objects | Optional (0 or 1) | If the Details parameter was set to true, this will be an array of addresses belonging to the added node |
| | o Address | Object | Optional (0 or more) | An object describing one of this node's addresses |
| | ▪ Address | String | Required (exactly 1) | An IP address and port number of the node. If the node was added using a DNS address, this will be the resolved IP address |
| | ▪ connected | string | Required (exactly 1) | Whether or not the local node is connected to this addnode using this IP address. Valid values are:<br>• false for not connected<br>• inbound if the addnode connected to us<br>• outbound if we connected to the addnode |

**Usage and Examples:**

***Arguments:***
1. dns (198ubscri, required) If false, only a list of added nodes will be p
rovided, otherwise connected information will also be available.
2. "node" (string, optional) If provided, return information about this specif
ic node, otherwise all nodes are returned.

***Result:***
```
[
 {
   "addednode" : "192.168.0.201",   (string) The node ip address
   "connected" : true|false,        (198ubscri) If connected
   "addresses" : [
      {
        "address" : "192.168.0.201:8333",  (string) The bitcoin server host and
 port
        "connected" : "outbound"           (string) connection, inbound or outb
ound
      }
      ,...
    ]
```

```
  }
  ,...
]
```

***Examples:***

> bitcoin-cli getaddednodeinfo true
> bitcoin-cli getaddednodeinfo true “192.168.0.201”
> curl –user myusername –data-binary ‘{“jsonrpc”: “1.0”, “id”:”curltest”, “method”: “getaddednodeinfo”, “params”: [true, “192.168.0.201”] }’ –H ‘content-type: text/plain;’ [http://127.0.0.1:8332/](http://127.0.0.1:8332/)

Example from Bitcoin Core 0.10.0 testnet:

```
bitcoin-cli -testnet getaddednodeinfo true
```

Result:

```
[
    {
        “addednode” : “bitcoind.example.com:18333”,
        “connected” : true,
        “addresses” : [
```

## getconnectioncount

returns the number of connections to other nodes.

**Parameters:** None

**Return:**

| Result No. | Name | Type | Presence | Description |
|---|---|---|---|---|
| 1 | result | Number (int) | Required (exactly 1) | The total number of connections to other nodes (both inbound and outbound) |

**Usage and Examples:**
***Arguments***: none
***Result***:
n (numeric) The connection count

***Examples***:
> bitcoin-cli getconnectioncount

> curl –user myusername –data-binary ‘{“jsonrpc”: “1.0”, “id”:”curltest”, “met

hod”: “getconnectioncount”, “params”: [] }’ –H ‘content-type: text/plain;’ http:

//127.0.0.1:8332/

Example from Bitcoin Core 0.10.0 testnet:

```
bitcoin-cli -testnet getconnectioncount
```

Result:

```
14
```

## getnettotals

returns information about network traffic, including bytes in, bytes out, and the current time.

**Parameters:** None

**Return:**

| Result No. | Name | Type | Presence | Description |
|---|---|---|---|---|
| 1 | result | object | Required (exactly 1) | An object containing information about the node's network totals |
| | • Totalbytesrecv | Number (int) | Required (exactly 1) | The total number of bytes received since the node was last restarted |
| | • Totalbytesent | Number (int) | Required (exactly 1) | The total number of bytes sent since the node was last restarted |
| | • timemillis | Number (int) | Required (exactly 1) | Unix epoch time in milliseconds according to the operating system's clock (not the node adjusted time) |

**Usage and Examples:**

***Result:***

{

"totalbytesrecv": n, (numeric) Total bytes received

"totalbytessent": n, (numeric) Total bytes sent

"timemillis": t (numeric) Total cpu time

}

***Examples:***

> bitcoin-cli getnettotals

> curl –user myusername –data-binary '{"jsonrpc": "1.0", "id":"curltest", "met

hod": "getnettotals", "params": [] }' –H 'content-type: text/plain;' http://127.

0.0.1:8332/

Example from Bitcoin Core 0.10.0 testnet:

```
bitcoin-cli –testnet getnettotals
```

Result:


```
{
    "totalbytesrecv" : 723992206,
    "totalbytessent" : 16846662695,
    "timemillis" : 1419268217354
}
```

### getnetworkinfo

returns information about the node’s connection to the network.

**Parameters:** None

**Return:**

| Result No. | Name | Type | Presence | Description |
|---|---|---|---|---|
| 1 | result | object | Required (exactly 1) | Information about this node’s connection to the network |
| | • version | Number (int) | Required (exactly 1) | This node’s version of Bitcoin Core in its internal integer format. |
| | • subversion | string | Required (exactly 1) | The user agent this node sends in its version message |
| | • protocolversion | Number (int) | Required (exactly 1) | The protocol version number used by this node. |
| | • timeoffset | Number (int) | Required (exactly 1) | The offset of the node’s clock from the computer’s clock (both in UTC) in seconds. The offset may be up to 4200 seconds (70 minutes) |
| | • connections | Number (int) | Required (exactly 1) | The total number of open connections (both outgoing and incoming) between this node and other nodes |
| | • proxy | string | Required (exactly 1) | The hostname/IP address and port number of the proxy, if set, or an empty string if unset |
| | • relayfree | Number (float) | Required (exactly 1) | The minimum fee a low-priority transaction must pay in order for this node to accept it into its memory pool |
| | • localservices | String (hex) | Required (exactly 1) | The services supported by this node as advertised in its version message |
| | • networks | array | Required (exactly 1) | An array with three objects: one describing the Ipv4 connection, one describing |

| | | | | |
|---|---|---|---|---|
| | | | | the Ipv6 connection, and one describing the Tor hidden service (onion) connection |
| | - Network | object | Optional (0 to 3) | An object describing a network. If the network is unroutable, it will not be returned |
| | ○ name | string | Required (exactly 1) | The name of the network. Either ipv4, ipv6, or onion |
| | ○ limited | bool | Required (exactly 1) | Set to true if only connections to this network are allowed according to the –onlynet Bitcoin Core command-line/configuration-file parameter. Otherwise set to false |
| | ○ reachable | bool | Required (exactly 1) | Set to true if connections can be made to or from this network. Otherwise set to false |
| | ○ proxy | string | Required (exactly 1) | The hostname and port of any proxy being used for this network. If a proxy is not in use, an empty string |
| | ○ localaddresses | Array | Required (exactly 1) | An array of objects each describing the local addresses this node believes it listens on |
| | ▪ Address | object | Optional (0 or more) | An object describing a particular address this node believes it listens on |
| | ➢ Addre ss | string | Required (exactly 1) | An IP address or .onion address this node believes it listens on. This may be manually configured, auto detected, or based on version messages this node received from its peers |
| | ➢ port | Number (int) | Required (exactly 1) | The port number this node believes it listens on for the associated address. This may |

| | | | | |
|---|---|---|---|---|
| | | | | be manually configured, auto detected, or based on version messages this node received from its peers |
| | ➢ score | Number (int) | Required (exactly 1) | The self-assigned score this node gives to this connection; higher scores means the node thinks this connection is better |

**Usage and Examples:**

***Result:***

```
{
  "version": xxxxx,                  (numeric) the server version
  "subversion": "/Satoshi:x.x.x/",     (string) the server subversion string
  "protocolversion": xxxxx,            (numeric) the protocol version
  "localservices": "xxxxxxxxxxxxxxxx", (string) the services we offer to the net
work
  "timeoffset": xxxxx,               (numeric) the time offset
  "connections": xxxxx,              (numeric) the number of connections
  "networks": [                      (array) information per network
  {
    "name": "xxx",                   (string) network (ipv4, ipv6 or onion)
    "limited": true|false,           (205ubscri) is the network limited using
-onlynet?
    "reachable": true|false,         (205ubscri) is the network reachable?
    "proxy": "host:port"             (string) the proxy that is used for this
network, or empty if none
  }
  ,...
  ],
  "relayfee": x.xxxxxxxx,            (numeric) minimum relay fee for non-fre
e transactions in btc/kb
```

“localaddresses”: [ (array) list of local addresses

{

“address”: “xxxx”, (string) network address

“port”: xxx, (numeric) network port

“score”: xxx (numeric) relative score

}

,...

]

}

***Examples:***
> bitcoin-cli getnetworkinfo

> curl –user myusername –data-binary ‘{“jsonrpc”: “1.0”, “id”:”curltest”, “met

hod”: “getnetworkinfo”, “params”: [] }’ –H ‘content-type: text/plain;’ http://12

7.0.0.1:8332/

Example from Bitcoin Core 0.10.0 testnet:

```
bitcoin-cli -testnet getnetworkinfo
```

Result:


```
{
    “version” : 100000,
    “subversion” : “/Satoshi:0.10.0/”,
    “protocolversion” : 70002,
    “localservices” : “0000000000000001”,
    “timeoffset” : 0,
    “connections” : 51,
    “networks” : [
        {
            “name” : “ipv4”,
            “limited” : false,
```

```
            “reachable” : true,
            “proxy” : “”
        },
        {
            “name” : “ipv6”,
            “limited” : false,
            “reachable” : true,
            “proxy” : “”
        },
        {
            “name” : “onion”,
            “limited” : false,
            “reachable” : false,
            “proxy” : “”
        }
    ],
    “relayfee” : 0.00001000,
    “localaddresses” : [
        {
            “address” : “192.0.2.113”,
            “port” : 18333,
            “score” : 6470
        },
        {
            “address” : “0600:3c03::f03c:91ff:fe89:dfc4”,
            “port” : 18333,
            “score” : 2029
        }
    ]
}
```

### getpeerinfo

returns data about each connected network node.

**Parameters:** None

**Return:**

| Result No. | Name | Type | Presence | Description |
|---|---|---|---|---|
| 1 | result | array | Required (exactly 1) | An array containing objects each describing one connected node. If no connections present, the array will be empty. |
| | • Node | Object | Optional (0 or more) | An object describing a particular connected node. |
| | - id | Number (int) | Required (exactly 1) | The node's index number in the local node address database |
| | - addr | String | Required (exactly 1) | The IP address and port number used for the connection to the remote node |
| | - addrlocal | String | Optional (0 or 1) | Our IP address and port number according to the remote node. |
| | - services | String (hex) | Required (exactly 1) | The services advertised by the remote node in its version message |
| | - lastsend | Number (int) | Required (exactly 1) | The Unix epoch time when we last successfully sent data to the TCP socket for this node |
| | - lastrecv | Number (int) | Required (exactly 1) | The Unix epoch time when we last received data from this node |
| | - bytesent | Number (int) | Required (exactly 1) | The total number of bytes we've sent to this node |
| | - conntime | Number (int) | Required (exactly 1) | The total number of bytes we've received from this node |
| | - pingtime | Number (float) | Required (exactly 1) | The number of seconds this node took to respond to our last P2P ping message |

| | | | | |
|---|---|---|---|---|
| | - pingwait | Number (float) | Optional (0 or 1) | The number of seconds we’ve been waiting for this node to respond to a P2P ping message. Only shown if there’s an outstanding ping message |
| | - version | Number (int) | Required (exactly 1) | The protocol version number used by this node |
| | - subver | string | Required (exactly 1) | The user agent this node sends in its version message. This string will have been sanitized to prevent corrupting the JSON results. May be an empty string |
| | - inbound | bool | Required (exactly 1) | Set to true if this node connected to us; set to false if we connected to this node |
| | - startingheight | Number (int) | Required (exactly 1) | The height of the remote node’s block chain when it connected to us as reported in its version message |
| | - banscore | Number (int) | Required (exactly 1) | The ban score we’ve assigned the node based on any 209ubscript209ur it’s made. By default, Bitcoin Core disconnects when the ban score reaches 100 |
| | - synced_headers | Number (int) | Required (exactly 1) | The highest-height header we have in common with this node based the last P2P headers message it sent us. If a headers message has not been received, this will be set to -1 |
| | - synced_blocks | Number (int) | Required (exactly 1) | The highest-height block we have in common with this node based on P2P inv messages this node sent us. If no block inv messages have been received from this node, this will be set to -1 |

| | | | | |
|---|---|---|---|---|
| | - syncnode | Bool | Required (exactly 1) | Whether we’re using this node as our syncnode during initial block download |
| | - inflight | array | Required (exactly 1) | An array of blocks which have been requested from this peer. May be empty |
| | ○ Blocks | Number (int) | Optional (0 or more) | The height of a block being requested from the remote peer |
| | - whitelisted | bool | Required (exactly 1) | Set to true if the remote peer has been whitelisted; otherwise, set to false. Whitelisted peers will not be banned if their ban score exceeds the maximum (100 by default). By default, peers connecting from localhost are whitelisted |

**Usage and Examples:**

***Result:***

[
  {
    “id”: n,                  (numeric) Peer index
    “addr”:”host:port”,      (string) The ip address and port of the peer
    “addrlocal”:”ip:port”,   (string) local address
    “services”:”xxxxxxxxxxxxxxxx”,   (string) The services offered
    “lastsend”: ttt,          (numeric) The time in seconds since epoch (Jan 1
1970 GMT) of the last send
    “lastrecv”: ttt,          (numeric) The time in seconds since epoch (Jan 1
1970 GMT) of the last receive
    “bytessent”: n,           (numeric) The total bytes sent
    “bytesrecv”: n,           (numeric) The total bytes received
    “conntime”: ttt,          (numeric) The connection time in seconds since ep
och (Jan 1 1970 GMT)
    “pingtime”: n,            (numeric) ping time
    “pingwait”: n,            (numeric) ping wait
    “version”: v,             (numeric) The peer version, such as 7001
    “210ubscri”: “/Satoshi:0.8.5/”,  (string) The string version
    “inbound”: true|false,     (210ubscri) Inbound (true) or Outbound (false)
    “startingheight”: n,       (numeric) The starting height (block) of the peer

    “banscore”: n,            (numeric) The ban score

"synced_headers": n, (numeric) The last header we have in common with this peer
"synced_blocks": n, (numeric) The last block we have in common with this peer
"inflight": [
n, (numeric) The heights of blocks we're currently asking from this peer
...
]
}
,...
]

***Examples:***
> bitcoin-cli getpeerinfo
> curl –user myusername –data-binary '{"jsonrpc": "1.0", "id":"curltest", "method": "getpeerinfo", "params": [] }' –H 'content-type: text/plain;' http://127.0.0.1:8332/

Example from Bitcoin Core 0.10.0 testnet:

```
bitcoin-cli -testnet getpeerinfo
```

Result:

```
[
    {
        "id" : 9,
        "addr" : "192.0.2.113:18333",
        "addrlocal" : "192.0.2.51:18333",
        "services" : "0000000000000002",
        "lastsend" : 1419277992,
        "lastrecv" : 1419277992,
        "bytessent" : 4968,
        "bytesrecv" : 105078,
        "conntime" : 1419265985,
        "pingtime" : 0.05617800,
        "version" : 70001,
        "subver" : "/Satoshi:0.8.6/",
```

```
        “inbound” : false,
        “startingheight” : 315280,
        “banscore” : 0,
        “synced_headers” : -1,
        “synced_blocks” : -1,
        “inflight” : [
        ],
        “whitelisted” : false
    }
]
```

## Blocks

Use these RPCs to get information / statistics about the blocks and blockchain.

### Getbestblockhash

returns the header hash of the most recent block on the best block chain.

**Parameters:** None

**Return:**

| Result No. | Name | Type | Presence | Description |
|---|---|---|---|---|
| 1 | result | String (hex) | Required (exactly 1) | The hash of the block header from the most recent block on the best block chain, encoded as hex in RPC byte order |

**Usage and Examples:**

***Result***
"hex" (string) the block hash hex encoded

***Examples***
> bitcoin-cli getbestblockhash
> curl –user myusername –data-binary '{"jsonrpc": "1.0", "id":"curltest", "met hod": "getbestblockhash", "params": [] }' –H 'content-type: text/plain;' http:// 127.0.0.1:8332/

Example from Bitcoin Core 0.10.0 testnet:

```
bitcoin-cli -testnet getbestblockhash
```

Result:

```
0000000000075c58ed39c3e50f99b32183d090aefa0cf8c324a82eea9b01a887
```

## getblock

gets a block with a particular header hash from the local block database either as a JSON object or as a serialized block.

**Parameters:**

| Param No. | Name | Type | Presence | Description |
|---|---|---|---|---|
| 1 | Header hash | String (hex) | Required (exactly 1) | The hash of the header of the block to get, encoded as hex in RPC byte order |
| 2 | Format | Bool | Optional (0 or 1) | Set to false to get the block in serialized block format; set to true (the default) to get the decoded block as a JSON object. We will use true for our implementation. |

**Return:**

Serialized block if format = false

Decoded transaction if format = true or omitted, as described below:

| Result No. | Name | Type | Presence | Description |
|---|---|---|---|---|
| 1 | result | Object / null | Required (exactly 1) | An object containing the requested block, or JSON null if an error occurred |
| | • hash | String(hex) | Required (exactly 1) | The hash of this block's block header encoded as hex in RPC byte order. This is the same as the hash provided in parameter #1 |
| | • confirmations | Number (int) | Required (exactly 1) | The number of confirmations the transactions in this block have, starting at 1 when this block is at the tip of the best block chain. This score will be -1 if the the block is not part of the best block chain |

| | | | | |
|---|---|---|---|---|
| | • size | Number (int) | Required (exactly 1) | The size of this block in serialized block format, counted in bytes |
| | • height | Number (int) | Required (exactly 1) | The height of this block on its block chain |
| | • version | Number (int) | Required (exactly 1) | This block’s version number. |
| | • merkelroot | String (hex) | Required (exactly 1) | The merkle root for this block, encoded as hex in RPC byte order |
| | • tx | array | Required (exactly 1) | An array containing the TXIDs of all transactions in this block. The transactions appear in the array in the same order they appear in the serialized block |
| | - txid | String (hex) | Required (1 or more) | The TXID of a transaction in this block, encoded as hex in RPC byte order |
| | • time | Number (int) | Required (exactly 1) | The value of the time field in the block header, indicating approximately when the block was created |
| | • nonce | Number (int) | Required (exactly 1) | The nonce which was successful at turning this particular block into one that could be added to the best block chain |
| | • bits | String (hex) | Required (exactly 1) | The value of the nBits field in the block header, indicating the target threshold this block’s header had to pass |
| | • difficulty | Number (float) | Required (exactly 1) | The estimated amount of work done to find this block relative to the estimated amount of work done to find block 0 |
| | • chainwork | String (hex) | Required (exactly 1) | The estimated number of block header hashes miners had to check from the genesis block to this block, encoded as big-endian hex |

| | | | | |
|---|---|---|---|---|
| | • previousblockhash | String (hex) | Required (exactly 1) | The hash of the header of the previous block, encoded as hex in RPC byte order |
| | • nextblockhash | String (hex) | Optional (0 or 1) | The hash of the next block on the best block chain, if known, encoded as hex in RPC byte order |

**Usage and Examples:**

***Result***

“hex” (string) the block hash hex encoded

***Examples***

> bitcoin-cli getbestblockhash

> curl –user myusername –data-binary ‘{“jsonrpc”: “1.0”, “id”:”curltest”, “met

hod”: “getbestblockhash”, “params”: [] }’ –H ‘content-type: text/plain;’ http://

127.0.0.1:8332/

Example from Bitcoin Core 0.10.0 testnet:

```
bitcoin-cli -testnet getblock \
            000000000fe549a89848c76070d4132872cfb6efe5315d01d7ef77e4900f2d39 \
            true
```

Result:

```
{
    “hash” : “000000000fe549a89848c76070d4132872cfb6efe5315d01d7ef77e4900f2d39”,
    “confirmations” : 88029,
    “size” : 189,
    “height” : 227252,
    “version” : 2,
    “merkleroot” : “c738fb8e22750b6d3511ed0049a96558b0bc57046f3f77771ec825b22d6a6f4a”,
    “tx” : [
```

```
        “c738fb8e22750b6d3511ed0049a96558b0bc57046f3f77771ec825b22d6a6f4a”
    ],
    “time” : 1398824312,
    “nonce” : 1883462912,
    “bits” : “1d00ffff”,
    “difficulty” : 1.00000000,
    “chainwork” : “000000000000000000000000000000000000000000000000083ada4a4009841a”,
    “previousblockhash” : “00000000c7f4990e6ebf71ad7e21a47131dfeb22c759505b3998d7a814c011df”,
    “nextblockhash” : “00000000afe1928529ac766f1237657819a11cfcc8ca6d67f119e868ed5b6188”
}
```

## getblockchaininfo

provides information about the current state of the block chain.

**Parameters:** None

**Return:**

| Result No. | Name | Type | Presence | Description |
|---|---|---|---|---|
| 1 | result | Object | Required (exactly 1) | A Json object containing information about the current state of the local block chain. |
| | • chain | String | Required (exactly 1) | The name of the block chain. One of: main for mainnet, test for testnet, or regtest for regtest. |
| | • blocks | Number (int) | Required (exactly 1) | The number of validated blocks in the local best block chain. For a new node with just the hardcoded genesis block, this will be 0 |
| | • headers | Number (int) | Required (exactly 1) | The number of validated headers in the local best headers chain. For a new node with just the hardcoded genesis block, this will be zero. This number may be higher than the number of blocks |
| | • bestblockhash | String (hex) | Required (exactly 1) | The hash of the header of the highest validated block in the best block chain, encoded as hex in RPC byte order. |
| | • difficulty | Number (float) | Required (exactly 1) | The difficulty of the highest-height block in the best block chain |
| | • verificationprogress | Number (float) | Required (exactly 1) | Estimate of what percentage of the block chain transactions have been verified so far, starting at 0.0 and increasing to 1.0 for fully verified. May slightly exceed |

| | | | | |
|---|---|---|---|---|
| | | | | 1.0 when fully synced to account for transactions in the memory pool which have been verified before being included in a block |
| | • chainwork | String (hex) | Required (exactly 1) | The estimated number of block header hashes checked from the genesis block to this block, encoded as big-endian hex |

**Usage and Examples:**

***Result:***

{

"chain": "xxxx", (string) current network name as defined in BIP70 (mai

n, test, regtest)

"blocks": xxxxxx, (numeric) the current number of blocks processed in

the server

"headers": xxxxxx, (numeric) the current number of headers we have vali

dated

"bestblockhash": "...", (string) the hash of the currently best block

"difficulty": xxxxxx, (numeric) the current difficulty

"verificationprogress": xxxx, (numeric) estimate of verification progress [0..

1]

"chainwork": "xxxx" (string) total amount of work in active chain, in hexa

decimal

}

***Examples***:

> bitcoin-cli getblockchaininfo

> curl –user myusername –data-binary ‘{“jsonrpc”: “1.0”, “id”:”curltest”, “met

hod”: “getblockchaininfo”, “params”: [] }’ –H ‘content-type: text/plain;’ http:/

/127.0.0.1:8332/

Example from Bitcoin Core 0.10.0 testnet

```
bitcoin-cli -testnet getblockchaininfo
```

Result


```
{
    “chain” : “test”,
    “blocks” : 315280,
    “headers” : 315280,
    “bestblockhash” : “000000000ebb17fb455e897b8f3e343eea1b07d926476d00bc66e2c0342ed50f”,
    “difficulty” : 1.00000000,
    “verificationprogress” : 1.00000778,
    “chainwork” : “000000000000000000000000000000000000000000000000015e984b4fb9f9b350”
}
```

## getblockcount

returns the number of blocks in the local best block chain.

**Parameters**: None

**Return**:

| Result No. | Name | Type | Presence | Description |
|---|---|---|---|---|
| 1 | result | Number (int) | Required (exactly 1) | The number of blocks in the local best block chain. For a new node with only the hardcoded genesis block, this number will be 0 |

**Usage and Examples:**

***Result***:

n (numeric) The current block count

***Examples***:

> bitcoin-cli getblockcount

> curl –user myusername –data-binary '{"jsonrpc": "1.0", "id":"curltest", "met

hod": "getblockcount", "params": [] }' –H 'content-type: text/plain;' http://127

.0.0.1:8332/

Example from Bitcoin Core 0.10.0 testnet:

```
bitcoin-cli -testnet getblockcount
```

Result:

```
315280
```

## getblockhash

returns the header hash of a block at the given height in the local best block chain.

**Parameters**:

| Param No. | Name | Type | Presence | Description |
|---|---|---|---|---|
| 1 | Block height | Number (int) | Required (exactly 1) | The height of the block whose header hash should be returned. The height of the hardcoded genesis block is 0 |

**Return**:

| Result No. | Name | Type | Presence | Description |
|---|---|---|---|---|
| 1 | result | String (hex) / null | Required (exactly 1) | The hash of the block at the requested height, encoded as hex in RPC byte order, or JSON null if an error occurred |

***Arguments***:

1. index (numeric, required) The block index

***Result***:

"hash" (string) The block hash

***Examples***:

> bitcoin-cli getblockhash 1000

> curl –user myusername –data-binary '{"jsonrpc": "1.0", "id":"curltest", "met

hod": "getblockhash", "params": [1000] }' –H 'content-type: text/plain;' http://

127.0.0.1:8332/

Example from Bitcoin Core 0.10.0 testnet:

```
bitcoin-cli –testnet getblockhash 240886
```

Result:

```
00000000a0faf83ab5799354ae9c11da2a2bd6db44058e03c528851dee0a3fff
```

### verifychain

verifies each entry in the local block chain database.

**Parameters**:

| Param No. | Name | Type | Presence | Description |
|---|---|---|---|---|
| 1 | Check level | Number (int) | Optional (0 or 1) | How thoroughly to check each block, from 0 to 4. Default is the level set with the –checklevel command line argument; if that isn’t set, the default is 3. Each higher level includes the tests from the lower levels<br><br>Levels are:<br>0. Read from disk to ensure the files are accessible<br>1. Ensure each block is valid<br>2. Make sure undo files can be read from disk and are in a valid format<br>3. Test each block undo to ensure it results in correct state<br>4. After undoing blocks, reconnect them to ensure they reconnect correctly |
| 2 | Number of blocks | Number (int) | Optional (0 or 1) | The number of blocks to verify. Set to 0 to check all blocks. Defaults to the value of the –checkblocks command-line argument; if that isn’t set, the default is 288 |

**Return**:

| Result No. | Name | Type | Presence | Description |
|---|---|---|---|---|
| 1 | result | bool | Required (exactly 1) | Set to true if verified; set to false if verification failed for any reason |

**Usage and Examples:**

***Arguments***:

1. checklevel (numeric, optional, 0-4, default=3) How thorough the block verif

ication is.

2. numblocks (numeric, optional, default=288, 0=all) The number of blocks to

check.

***Result***:

true|false (225ubscri) Verified or not

***Examples***:

> bitcoin-cli verifychain

> curl –user myusername –data-binary '{"jsonrpc": "1.0", "id":"curltest", "met

hod": "verifychain", "params": [] }' –H 'content-type: text/plain;' http://127.0

.0.1:8332/

Example from Bitcoin Core 0.10.0 testnet:

```
bitcoin-cli -testnet verifychain 4 10000
```

Result:

```
true
```

### getchaintips

returns information about the highest-height block (tip) of each local block chain, including the main chain as well as orphaned branches.

**Parameters**: None

**Return**:

| Result No. | Name | Type | Presence | Description |
|---|---|---|---|---|
| 1 | result | array | Required (exactly 1) | An array of JSON objects, with each object describing a chain tip. At least one tip—the local best block chain—will always be present |
| | • tip | object | Required (1 or more) | An object describing a particular chain tip. The first object will always describe the active chain (the local best block chain) |
| | - height | Number (int) | Required (exactly 1) | The height of the highest block in the chain. A new node with only the genesis block will have a single tip with height of 0 |
| | - hash | String (hex) | Required (exactly 1) | The hash of the highest block in the chain, encoded as hex in RPC byte order |
| | - branchlen | Number (int) | Required (exactly 1) | The number of blocks that are on this chain but not on the main chain. For the local best block chain, this will be 0; for all other chains, it will be at least 1 |
| | - status | string | Required (exactly 1) | The status of this chain. Valid values are:<br>• active for the local best block chain<br>• invalid for a chain that contains one or more invalid blocks<br>• headers-only for a chain with valid headers whose corresponding blocks both |

| | | | | |
|---|---|---|---|---|
| | | | | haven’t been validated and aren’t stored locally<br>• valid-headers for a chain with valid headers whose corresponding blocks are stored locally, but which haven’t been fully validated<br>• valid-fork for a chain which is fully validated but which isn’t part of the local best block chain (it was probably the local best block chain at some point)<br>• unknown for a chain whose reason for not being the active chain is unknown |

**Usage and Examples:**

***Result***:

[

{

“height”: xxxx,        (numeric) height of the chain tip

“hash”: “xxxx”,        (string) block hash of the tip

“branchlen”: 0         (numeric) zero for main chain

“status”: “active”      (string) “active” for the main chain

},

{

“height”: xxxx,

“hash”: “xxxx”,

“branchlen”: 1         (numeric) length of branch connecting the tip to the

main chain

“status”: “xxxx”       (string) status of the chain (active, valid-fork, va

lid-headers, headers-only, invalid)

}

]

Possible values for status:

1. "invalid" This branch contains at least one invalid block

2. "headers-only" Not all blocks for this branch are available, but th

e headers are valid

3. "valid-headers" All blocks are available for this branch, but they w

ere never fully validated

4. "valid-fork" This branch is not part of the active chain, but is

fully validated

5. "active" This is the tip of the active main chain, which is c

ertainly valid

***Examples***:

> bitcoin-cli getchaintips

> curl –user myusername –data-binary '{"jsonrpc": "1.0", "id":"curltest", "met

hod": "getchaintips", "params": [] }' –H 'content-type: text/plain;' http://127.

0.0.1:8332/

Example from Bitcoin Core 0.10.0 testnet:

```
bitcoin-cli -testnet getchaintips


[
    {
        "height" : 312647,
        "hash" : "000000000b1be96f87b31485f62c1361193304a5ad78acf47f9164ea4773a843",
        "branchlen" : 0,
        "status" : "active"
    },
    {
        "height" : 282072,
```

```
        "hash" : "00000000712340a499b185080f94b28c365d8adb9fc95bca541ea5e708f31028",
        "branchlen" : 5,
        "status" : "valid-fork"
    },
    {
        "height" : 281721,
        "hash" : "000000006e1f2a32199629c6c1fbd37766f5ce7e8c42bab0c6e1ae42b88ffe12",
        "branchlen" : 1,
        "status" : "valid-headers"
    },
]
```

## Miscellaneous

Other general purpose RPCs.

### Verifymessage

verifies a signed message.

**Parameters**:

| Param No. | Name | Type | Presence | Description |
|---|---|---|---|---|
| 1 | Address | String (base58) | Required (exactly 1) | The P2PKH address (in Base58Check encoding) corresponding to the private key which made the signature. A P2PKH address is a hash of the public key corresponding to the private key which made the signature. When the ECDSA signature is checked, up to four possible ECDSA public keys will be reconstructed from the signature; each key will be hashed and compared against the P2PKH address provided to see if any of them match. If there are no matches, signature validation will fail |
| 2 | signature | String (base64) | Required (exactly 1) | The signature created by the signer encoded as base-64 [3.11] |
| 3 | Message | String | Required (exactly 1) | The message exactly as it was signed (e.g. no extra whitespace) |

**Return**:

| Result No. | Name | Type | Presence | Description |
|---|---|---|---|---|
| 1 | result | Bool/null | Required (exactly 1) | Set to true if the message was signed by a key corresponding to the |

| | | | | |
|---|---|---|---|---|
| | | | | provided P2PKH address; set to false if it was not signed by that key; set to JSON null if an error occurred |

***Note:*** *Base58Check-encoding is a modified Base 58 encoding. There is a difference between these formats [2.21]. Please be sure which encoding can be used in case "String (base58)" parameter type.*

**Usage and Examples:**

***Arguments***:

1. "bitcoinaddress" (string, required) The bitcoin address to use for the signa

ture.

2. "signature" (string, required) The signature provided by the signer in

base 64 encoding (see signmessage).

3. "message" (string, required) The message that was signed.

***Result***:

true|false (232ubscri) If the signature is verified or not.

***Examples***:

> bitcoin-cli verifymessage "1D1ZrZNe3Juo7ZycKEYQQiQAWd9y54F4XZ" "signature" "my

message"

As json rpc

> curl –user myusername –data-binary '{"jsonrpc": "1.0", "id":"curltest", "met

hod": "verifymessage", "params": ["1D1ZrZNe3Juo7ZycKEYQQiQAWd9y54F4XZ", "signatu

re", "my message"] }' –H 'content-type: text/plain;' http://127.0.0.1:8332/

Example from Bitcoin Core 0.10.0 testnet:

```
bitcoin-cli –testnet verifymessage \
  mgnucj8nYqdrPFh2JfZSB1NmUThUGnmsqe \
  IL98ziCmwYi5pL+dqKp4Ux+zCa4hP/xbjHmWh+Mk/lefV/0pWV1p/gQ94jgExSmgH2/+PDcCCrOHAady2IEySSI= \
  ‘Hello, World!’
```

Result:

```
true
```

## validateaddress

returns information about the given bitcoin address.

**Parameters**:

| Param No. | Name | Type | Presence | Description |
|---|---|---|---|---|
| 1 | Address | String (base58) | Required (exactly 1) | The P2PKH address to validate (in Base58Check encoding). |

**Return**:

| Result No. | Name | Type | Presence | Description |
|---|---|---|---|---|
| 1 | result | Object | Required (exactly 1) | Information about the address |
| | • isvalid | Bool | Required (exactly 1) | Set to true if address is valid; false otherwise |
| | • address | String (base58) | Optional (0 or 1) | If the address is valid, this is the address |
| | • ismine | Bool | Optional (0 or 1) | NA (wallet support required) |
| | • iswatchonly | Bool | Optional (0 or 1) | NA (wallet support required) |
| | • isscript | Bool | Optional (0 or 1) | NA (wallet support required) |
| | • script | String | Optional (0 or 1) | NA (Only returned for P2SH addresses) |
| | • hex | String (hex) | Optional (0 or 1) | NA (Only returned for P2SH addresses) |
| | • addresses | array | Optional (0 or 1) | NA (Only returned for P2SH addresses) |
| | - Address | String | Optional (0 or more) | |
| | • sigrequired | Number (int) | Optional (0 or 1) | NA (Only returned for P2SH addresses) |
| | • pubkey | String (hex) | Optional (0 or 1) | NA (wallet support required) |
| | • iscompressed | Bool | Optional (0 or 1) | NA (wallet support required) |
| | • account | String | Optional (0 or 1) | NA (wallet support required) |

**Usage and Examples:**

***Arguments***:

2. "bitcoinaddress" (string, required) The bitcoin address to validate

***Result***:

{

"isvalid" : true|false, (235ubscri) If the address is valid or not. If n

ot, this is the only property returned.

"address" : "bitcoinaddress", (string) The bitcoin address validated

"ismine" : true|false, (235ubscri) If the address is yours or not

"isscript" : true|false, (235ubscri) If the key is a script

"pubkey" : "publickeyhex", (string) The hex value of the raw public key

"iscompressed" : true|false, (235ubscri) If the address is compressed

"account" : "account" (string) The account associated with the address

, "" is the default account***Examples***:

> bitcoin-cli verifymessage "1D1ZrZNe3Juo7ZycKEYQQiQAWd9y54F4XZ" "signature" "my

message"

}

As json rpc

> curl –user myusername –data-binary '{"jsonrpc": "1.0", "id":"curltest", "met

hod": "validateaddress", "params": ["1PSSGeFHDnKNxiEyFrD1wcEaHr9hrQDDWc"] }' –H

'content-type: text/plain;' http://127.0.0.1:8332/

Example from Bitcoin Core 0.10.0 testnet:

```
bitcoin-cli –testnet validateaddress mgnucj8nYqdrPFh2JfZSB1NmUThUGnmsqe
```

Result:


```
{
    “isvalid” : true,
    “address” : “mgnucj8nYqdrPFh2JfZSB1NmUThUGnmsqe”,
    “ismine” : false
}
```


### 5.7.3 Java Wrapper of Daemon Core RPC

Draft list of functions:

- validateBtcAddress [btcAddress (request param as String) -> response: isValid (1-true, 0-false)]

Under construction…

# 6. Digital Algorithms and Schemes

## 6.1 Mnemonic Code Generation Scheme

A mnemonic code or mnemonic sentence is a group of easy to remember words. This can be further used for various purposes like generation of private keys for Type 1 deterministic wallets [Refer section 6.3.1]. The scheme described here is in accordance with BIP-0039 [2.17]. According to BIP-0039 the number of words in a mnemonic sentence can range from 12 to 24 [Refer

Table 6.1.1]. Wordlist with 2048 pre-defined English words will be used for this implementation [2.18].

***Note:** We will be using wordlist for English initially. Support for other languages will be considered in later phases.*

Steps for mnemonic code generation:

The mnemonic code can be generated by a sequence of steps that include generation of entropy, checksum and then finally the mnemonic code. Following steps describe the criteria that entropy, checksum and mnemonic code need to satisfy. Each step describes the output expected for that particular step.

1. Generate **initial entropy** $InitENT$ of size 128-256 bits (could use a secure random number generator for this part). Note that larger entropy leads to greater security, but it also leads to larger sentence length.

   Properties of entropy length $ENT$:

   - should be multiple of 32
   - minimum value = 128 bits
   - maximum value = 256 bits

   Output of this step will be a $[ENT]$ bits random number.

2. Hash the initial entropy with SHA256. The output will be a string of 256 bits $ENTHash$.

   $$ENTHash = SHA256(InitENT)$$

3. Generate **checksum**. Let's denote checksum length as $CS$.

   $$CS = ENT / 32$$

   $Checksum$ = first $[CS]$ bits of the hash $ENTHash$ obtained in step 2.

   Output of this step will be $Checksum$ of $[CS]$ bits size.

4. Derive **final entropy** by appending checksum to the end of initial entropy.

   $$FinalEntropy = [InitENT] + [Checksum]$$

Output of this step will be entropy string *FinalEntropy* of *[ENT + CS]* bits size.

5. The final entropy bit string is divided into 11 bit long chunks. This will be give us output containing *N* chunks of 11 bits each, where *N = [ENT + CS]/11*
6. Each chunk encodes a number from 0 to 2047 which is used as an index to the wordlist.

Let’s denote Mnemonic Sentence length as *MS*.

After indexing the wordlist (2048 pre-defined words), we will obtain *[MS]* number of words. Properties of MS:

$$MS = (ENT + CS)/11$$

- should have minimum value of 12
- should be divisible by 3

Thus, the final output will be a set of *[MS]* number of words.

The following table describes the relation between the initial entropy length *ENT*, the checksum length *CS* and the length of the generated mnemonic sentence *MS* in words.

$$CS = ENT / 32$$

$$MS = (ENT + CS) / 11$$

| ENT | CS | ENT+CS | MS |
|---|---|---|---|
| 128 | 4 | 132 | 12 |
| 160 | 5 | 165 | 15 |
| 192 | 6 | 198 | 18 |
| 224 | 7 | 231 | 21 |
| 256 | 8 | 264 | 24 |

Table 6.1.1

## 6.2 Shamir's Secret Sharing Scheme

Shamir's secret sharing scheme (or 4S for short) is an algorithm that divides a secret into shares. Secret can be recovered by combining certain numbers of shares.

4S will be used to split the mnemonic seed [6.1] into number of parts. It will also be used to regenerate the mnemonic seed from certain number of parts.

### 6.2.1 Basic Terms

**Secret (S)**: Secret is a secret number that you want to share with others securely.

**Share**: Share is a piece of secret. Secret is divided into pieces and each piece is called share. It is computed from given secret. In order to recover the secret, you need to get certain numbers of shares.

**Threshold (k)**: Threshold is the number of shares you need at least in order to recover your secret. You can restore your secret only when you have more than or equal to the number of threshold.

**Prime (p)**: A random prime number.

Basically, 4S is a method to give *n* people, each a part of a secret so that any *k* of the recipients (k<n) can reveal the secret.

### 6.2.2 Split Secret into shares

Given a secret value S, the number of participants n, the threshold number k, and some prime number p, we construct a polynomial:

$$y = f(x) \text{ of degree } k-1 \text{ (modulo our prime } p)$$

with constant term $S$.

Next we choose n unique random integers between 1 and p−1, inclusive, and evaluate the polynomial at those n points. Each of the n participants is given a (x, y) pair.

Steps in detail:

**1. Convert into Integer**

For 4S, the secret needs to be an integer. Hence if the secret is in some other format (ex. String, hex etc.) convert it into integer first. Note that depending on the programming language chosen, there might be inbuilt package / functions to achieve this.

For example, if the secret is a string, just convert the string into a byte array so that we can treat is as a number. Steps:

i. Convert string to byte array
ii. Convert the byte array into integer

For ex, in Java this can be done as follows:

```
String mnm_seed = “abc def ghi jkl mno pqr”;
byte[] byteArray = mnm_seed.getBytes();
BigInteger S = new BigInteger(byteArray);
```

If the secret is an integer, skip this step.

For this example, let $S = 1234.$

**2. Decide number of shares (n) and threshold (k)**

Note that $k$ parts will be required to regenerate the secret. Hence, chose $s$ and $k$ such that $k$ parts can always be obtained while recovering the secret.

For this example, let $n = 6, k =3.$

**3. Create polynomial**

We need to create a polynomial of the form: $y = f(x) \bmod p$

i. Determine constant term and degree of polynomial

$f(x) = a_0 + a_1x + a_2x^2 + a_3x^3 + \ldots + a_{k-1}x^{k-1}$

- The constant term $a_0 = S$
- degree of polynomial = $k-1$

Hence for $k = 3$ and $S = 1234$, we need to build a polynomial with $degree\ 2$ and $a_0 = 1234$

$f(x) = 1234 + a_1x + a_2x^2$

ii. Determine coefficients

Chose $k-1$ random numbers (use a Random Number Generator) such that:

$0 < a_n < S$

$Let\ a_1 = 166;\ a_2 = 94$

$Hence,\ f(x) = 1234 + 166x + 94\ x^2$

iii. Select a random prime number

Chose a random prime number (p) such that:

$p > max(S,n)$

*Let* $p = 1613$

iv. Final polynomial

$$y = f(x) \bmod p$$

$$y = (1234 + 166x + 94\, x^2) \bmod 1613$$

### 6.2.3 Create shares

To divide the secret into $n$ shares, we need to construct $n$ points (shares) using the polynomial:

$$y = (1234 + 166x + 94\, x^2) \bmod 1613$$

Since $n = 6$ for this example, we will have 6 points. Note that start with $x = 1$ *and NOT* $x = 0$.

For $x = 1$ *to* $6$, the 6 points are as follows:

$$(1, 1494); (2,329); (3,965); (4,176); (5, 1188); (6,775)$$

Out of these n (6) points, any k (3) points can be used to regenerate the secret.

### 6.2.4 Reconstruct Secret from given number of shares

i. Get the secret integer

To reconstruct the secret, we need following information:

$n = 6, k =3, p =1613,$

*k shares:*

$(x0, y0) = ( 1, 1494); \; (x1, y1) = (2,329); (x2, y2) = (3,965)$

Once we have the above information, we can use Lagrange Interpolation [3.7]. This technique can rebuild entire polynomial. The coefficients can be calculated according to formula below:

$$a_i\,(x) = [\, \Sigma^{k-1}_{i=0}\; y_i\; \Pi_{\,0<=j<=k-1,\, j\neq i}\, (x-x_j)/(\, x_i - x_j) \;\; ] \bmod p$$

but since $S = a_0$, we only need to find $a_0 = a_0\,(0)$

$$a_0 = \left[ \sum_{i=0}^{k-1} y_i \prod_{\substack{0 \le j \le k-1 \\ j \ne i}} \frac{-x_j}{x_i - x_j} \right] \bmod p$$

where $x_i - x_j \neq 0$

Pseudo code for above equation:

*$a_0 = 0$*

*For i = 0 to k-1*

*z = 1*

*For j = 0 to k-1*

*If j != i*

*$z = z * (-x[j])*(x[i]-x[j])^{-1}$*

*End If*

*End For*

*$a_0 = a_0 + (y[i] * z)$*

*End For*

*$a_0 = a_0 \bmod p$*

We get $a_0 = 1234$ after solving for above values.

***Note**: the exponent −1 signifies taking the multiplicative inverse. Most of the programming languages will have inbuilt packages to perform mathematical operations such as multiplicative inverse.*

ii. Convert integer to desired format
If Step 1 from 6.2.2 was executed to convert a specific format to integer, follow the reverse procedure to convert the integer back to the desired format.

Ex. Integer to string

- Convert string to byte array
- Convert the byte array into integer

Example code in Java:

```
    BigInteger bigInt = BigInteger.valueOf(S);
    byte[] buffer = bigInt.toByteArray();
    String secretString = new String(buffer, StandardCharsets.UTF_8);
```

## 6.3 Elliptic Curve Digital Signature Algorithm in case Bitcoins

ECDSA (X9.62 standard digital signature scheme) [3.1], [3.2] implementation in the Btc protocol [2.13] uses elliptic curve on over finite field $F_p$ where $p$ is a prime number greater than 3.

The elliptic curve $E$ defined over $F_p$ can be expressed by the Weierstra equation:

$y^2 = (x^3+ax+b) \bmod p$ where $a, b \in F_p$ and $4a^2 + 27b^2 \neq 0$. All the points satisfying the equation together with the identity element $O$ (point of infinity) form group. Different curves will have different domain parameters to form different elliptic curve groups.

The **Domain Parameters** on the curve over $F_p$ are a sextuple, expressed as $T = \{p, a, b, G, n, h\}$, where

> the integer $p$ specifying the finite field $F_p$, $p$ is prime modulo [3.8], [3.9]
> $a, b$ are constants defining the equation,
> $G$ is the base point on the curve, of order $n$
> $n$ is $G$'s order, a sufficiently large prime number (at least 160 bits), and integer $h$ is its the cofactor.

Btc protocol [2.13] uses 256-bit elliptic curve (Koblitz curve), where
curve name ID is *secp256k1*
curve equation:

(f.3.1) $y^2 = (x^3+7) \bmod p$, where $a=0, b=7$

domain parameters associated with curve in the Hex representation:

$p$ = FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFE FFFFFC2F
$= 2^{256} - 2^{32} - 2^9 - 2^8 - 2^7 - 2^6 - 2^4 - 1$

$a$ = 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
$b$ = 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000007

$G$ **in compressed form**

$G$ = 02 79BE667E F9DCBBAC 55A06295 CE870B07 029BFCDB 2DCE28D9 59F2815B 16F81798
where

$G_x$ = 79BE667E F9DCBBAC 55A06295 CE870B07 029BFCDB 2DCE28D9 59F2815B 16F81798
$G_y$ is even and can be calculated from the equation (f.3.1)

**$G$ in uncompressed form**

$G$ = 04 79BE667E F9DCBBAC 55A06295 CE870B07 029BFCDB 2DCE28D9 59F2815B 16F81798 483ADA77 26A3C465 5DA4FBFC 0E1108A8 FD17B448 A6855419 9C47D08F FB10D4B8

where

$G_x$ = 79BE667E F9DCBBAC 55A06295 CE870B07 029BFCDB 2DCE28D9 59F2815B 16F81798
$G_y$ = 483ADA77 26A3C465 5DA4FBFC 0E1108A8 FD17B448 A6855419 9C47D08F FB10D4B8

$n$ = FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFE BAAEDCE6 AF48A03B BFD25E8C D0364141
$h$= 01

**There are three main processes in Btc system based on ECDSA:**

1. Private/Public Key Generation
2. Trx (Message) Signature/Encryption
3. Signature Verification/Decryption

### 6.3.1 Points operations:

Given two points $P(x_1, y_1)$ and $Q(x_2, y_2)$ (with Affine coordinates [3.10]) on the curve.

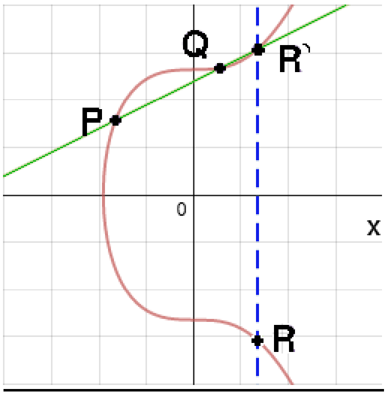
#### Point Addition

**Point Addition** is defined as the reflection through the *x-axis* of the third intersecting point $R'$ on a line that includes $P$ and $Q$ (Pic. 6.3.1). $P \neq Q$

$$R(x_3, y_3) = P(x_1, y_1) + Q(x_2, y_2)$$

(f.6.3.1)
$$x_3 = (\lambda^2 - (x_1 + x_2)) \bmod p$$
$$y_3 = (\lambda (x_1 - x_3) - y_1) \bmod p$$

where $\lambda = (y_2 - y_1/x_2 - x_1) \bmod p$




Pic. 6.3.1

## Point Doubling

**Point Doubling** is defined by finding the line tangent to the point to be doubled, *P*, and taking reflection through the *x-axis* of the intersecting point *R'* on the curve to get *R* (Pic. 6.3.2).

$$R(x_3, y_3) = 2P = P(x_1, y_1) + P(x_1, y_1)$$

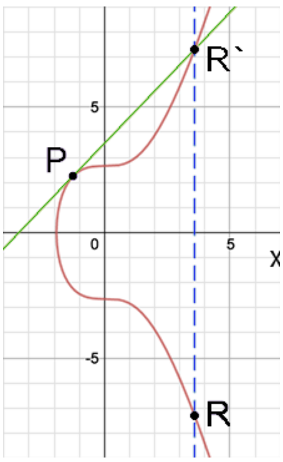
(f.6.3.2) $$x_3 = (\lambda^2 - 2x_1) \bmod p$$

$$y_3 = (\lambda (x_1 - x_3) - y_1) \bmod p$$

where $\lambda = ((3x_1^2 + a)/2y_1) \bmod p$

for secp256k1 curve (see f.3.1):

(f.6.3.2') $$\lambda = (3x_1^2/2y_1) \bmod p$$




Pic. 6.3.2

## Point Multiplication

**<u>Scalar Point Multiplication</u>** is repeated ***Point Additions*** and ***Point Doubling*** operations.

$$R(x_3, y_3) = a\ P(x_1, y_1)$$

where ***Scalar Point Multiplication*** defined by adding the point $P$ to itself $a$ times. For example:

$$R = 7P$$

$$R = P + (P + (P + (P + (P + (P + P)))))$$

The process of scalar multiplication is normally simplified by using a combination of ***Point Addition*** and ***Point Doubling*** operations.

$$R = 7P$$

$$R = P + 6P$$

$$R = P + 2\ (3P)$$

$$R = P + 2\ (P + 2P)$$

Here, $7P$ has been broken down into two ***Point Doubling*** steps and two ***Point Addition*** steps.

**<u>Unified Formula for Point's Addition and Doubling [4.3], [4.4]:</u>**

Given two points $(x_1, y_1)$ and $(x_2, y_2)$ on the curve using parameters $secp256k1$, whether they are equal or not, both point addition and doubling can be calculated as follows:

$$x_3 = \lambda^2 - (x_1 + x_2)$$
$$y_3 = \lambda\ (x_1 - x_3) - y_1$$

where

(f.6.3.3) $\lambda = ((x_1 + x_2)^2 - x_1x_2 + a)/(y_1 + y_2)$

$y^1+y^2 \neq 0$ (it is not applicable to all point additions)

for secp256k1 curve (see f.3.1)

(f.6.3.3') $\lambda = ((x_1 + x_2)^2 - x_1x_2)/(y_1 + y_2)$

**<u>Most efficient and secure Unified Formula [4.1], [4.2]:</u>**

(f.6.3.4) $\lambda =[\ (x_1 + x_2)^2 - x_1x_2 + a + (-1)^{\delta}(y_1 - y_2)\ ]/[\ y_1 + y_2 + (-1)^{\delta}(x_1 - x_2)\ ]$

$$y_1 + y_2 + (-1)^{\delta}(x_1 - x_2) \neq 0$$

where

$\delta = 0$ when $y_1 + y_2 + x_1 - x_2 \neq 0$ and $\delta = 1$ otherwise

or a randomized choice of $\delta$ when both choices give nonzero values.

For secp256k1 curve (see f.3.1)

(f.6.3.4’) $\lambda =[ (x_1 + x_2)^2 - x_1x_2 + (-1)^{\delta}(y_1 - y_2) ]/[ y_1 + y_2 + (-1)^{\delta}(x_1 - x_2) ]$

Therefore (f.6.3.5)

$\lambda = [ (x_1 + x_2)^2 - x_1x_2 + (y_1 - y_2) ]/[ (y_1 + y_2) + (x_1 - x_2) ]$ when $x_2 \neq x_1 + (y_1 + y_2)$

$\lambda = [ (x_1 + x_2)^2 - x_1x_2 + (y_2 - y_1) ]/[ (y_1 + y_2) + (x_2 - x_1) ]$ otherwise

**Note:** 1. *If any fixed $\lambda$ is used, then it may be that the attack can still be applied.*

*2. All formulas given above uses Affine coordinates.*

*3. “mod p” operations are omitted in both types of the Unified Formula. There is small risk of errors derived from improper modular usage. Firstly, developer should try Unified Formula with “mod p” operations if errors is received in the implementation stage. Secondly, developer can come back to general ECDSA implementation with Points Doubling and Points Additions formulas instead of Unified Formula if errors is still received.*

### Simultaneous Elliptic Scalar Multiplication

**Simultaneous Elliptic Scalar Multiplication** is a method to calculate curve point $C = k\,G + l\,Q$

Note that using modification of **Shamir’s Trick** (also known as **Straus’s algorithm**) [4.5], a sum of two scalar multiplications can be calculated faster than two scalar multiplications done independently. Using a **Straus’s algorithm** [4.6] to process $k\,G + l\,Q$ in parallel can reduce the number of operations needed. The algorithm uses a **2**$NAF$ (*see Glossary*) representation of integers $k$ and $l$.

**2$NAF$ conversion algorithm:**

***Input:*** $d$ - $m$-bit integer

***Output:*** **2**$NAF(d)$ where

$$d = 2^{m-1}d_{m-1} + 2^{m-2}d_{m-2} + \ldots + 2^2d_2 + 1^1d_1 + 2^0d_0,\ [d_0, d_m] \in \{0,-1, 1\}$$

***Pseudo Code:***

```
i = 0
While (d > 0) do
        If (d mod 2) == 1
```

*$d_i$ = d mods 4*

*d = d – $d_i$*

*Else*

*$d_i$ = 0*

*End if*

*d = d/2*

*i = i + 1*

*End while*

*Return ($d_{i-1}$ , $d_{i-2}$., …, $d_0$)*

***Where “mods” Pseudo Code:***

*If (d mod 4) >= 2*

*Return (d mod 4) – 4*

*Else*

*Return (d mod 4)*

*End if*

**Example:**

*d = 7*

$2^3 d_3 + 2^2 d_2 + 2^1 d_1 + 2^0 d_0 = 2^3 x1 + 2^2 x0 + 2^1 x0 + 2^0 x(-1) = 8 + 0 + 0 - 1 = 7$

$_2NAF(d) = 1\ 0\ 0\ -1$

**Straus’ Algorithm:**

***Input:*** two points $G(x_1, y_1)$ and $Q(x_2, y_2)$ on the curve.
*m*-bit integers *k* and *l*

***Output:*** Curve point $C(x_3, y_3) = k\,G + l\,Q$

***Precomputations:***

1. Compute $xG + yQ$ where any $x, y \in \{0, -1, 1\}$
2. Compute **2**$NAF(k)$ and **2**$NAF(l)$

***Pseudo Code:***

*C = ∞* // Point at infinity

*For i = m-1 to 0*

*C = 2C* // using Point Doubling formula (f.6.3.2, f.6.3.2’)

$C = C + (k_i G + l_i Q)$ // using precomputations

*End For*

*Return C*

**Example:**

***Computing:*** $C = k\ G + l\ Q$ where $k = 13, l = 7$

***Precomputations:***

1. Compute $(G + Q), (G – Q)$

2. Compute

**2**$NAF(k) = 10\text{-}101\ [2^4x1 + 2^3x0 + 2^2x(-1) + 2^1x0 + 2^0x1 = 16 + 0 -4 + 0 + 1 = 13]$

where $m_k=4$

**2**$NAF(l) = 100\text{-}1\ [2^3x1 + 2^2x0 + 2^1x0 + 2^0x(-1) = 8 + 0 + 0 -1 = 7]$

where $m_l=3$

add $(m_k - m_l)$ zeros in the beginning of **2**$NAF(l)$, because $m_k > m_l$ => **2**$NAF(l) = 0100\text{-}1$

***Code start***:

*C = ∞*

i = 4:

*C = 2C = ∞*

*C = ∞ + G = G*

i = 3:

$C = 2G$

$C = 2G + Q$

i = 2:

$C = 2(2G + Q) = 4G + 2Q$

$C = 4G + 2Q – G = 3G + 2Q$

i = 1:

$C = 2(3G + 2Q) = 6G + 4Q$

$C = 6G + 4Q + \infty = 6G + 4Q$

i = 0:

$C = 2(6G + 4Q) = 12G + 8Q$

$C = 12G + 8Q + (G – Q) = \mathbf{13G + 7Q}$

### 6.3.2 Private/Public Key Generation

**Private Key (PrvKey)**: a random 256-bit integer $d$ in the range of $[1, n – 1]$.

$N=1.156*10^{77}$ , $n$ is slightly less than $2^{256}$

***Note:*** *The visible universe is estimated to contain* $10^{80}$ *atoms.* ☺

IntDS will use “type 1 deterministic wallet” approach to generate private key.

**Steps to produce Private Key:**

1. To generate a private key take $SHA256(string + i)$,

where

$i$ – ASCII-coded number that starts from 1 and increments as additional keys are needed.

***Note:*** $i$ *will be equal a consecutive incremental number which is captured in “trx_management” DB (SYSTEM_BTC_ADDRESSES table, SEQUENCE_NUMBER field) in case Single-sig Transaction Management SubSystem.*

*String* – Mnemonic code. Mnemonic code are English word sequence of 12 to 24 words. It is important that mnemonic code string should not have spaces between words. The way to generate mnemonic code is described in *the Paragraph 6.1*.

Output of this step will be *HashedSeed* which is 256 binary digits shown as 64 hexadecimal digits, each 4 bits.

2. Convert *HashedSeed* in hexadecimal representation to big integer in decimal representation.

Output of this step will be Private Key $d$.

3. Check the result $d < n – 1$.

***Note:*** *Private Key in a script should be Base58Check encoding string with the prefix used when encoding a private key is 128 (0x80 in hex)*

**Public Key (PubKey)**: $Q = dG$ where $Q$ is point on the curve, $G$ is the base point on the curve, of order $n$
A Private Key can be converted into a Public Key, but a Public Key cannot be converted back into a Private Key because the math only works one way and extremely difficult to determine what $d$ was.

The Public Key is derived from the Private Key by **Scalar Point Multiplication** of the base point $G$ a number of times equal to the value of the Private Key $d$.

***Note:*** *EC Point multiplication is core operation in the ECDSA theory. The straightforward way of computing a point multiplication is through repeated addition. There are algorithms to make multiplication more efficient than repeated addition:*

- *Double-and-add*
- *Double-and-add-always*
- *Windowed method*
- *Sliding-window method*
- *wNAF method*
- *Montgomery ladder (implemented in OpenSSL, good performance, low security)*

  *etc.*

*There are two important criteria for iDaemon System: performance and security.*
*Each algorithm should be reviewed to make a right choice. Currently system implements simplest method "**Double-and-add**" approach which is algorithm with low security. "**Double-and-add**" algorithm is vulnerable to Side Channel Attacks as:*
*- Fault Analysis attacks*
*- Power Consumption attacks (SPA and DPA)*
*- Timing attacks*
*The solutions to protect IntDS from Side Channel Attacks should be considered later. Not in this stage of project implementation!*

**Steps to produce Public Key from Private Key:**

1. Convert the Private Key $d$ from decimal to binary representation (*see Appendix H*).

For example $d$ is:

105 in decimal representation, which is 1101001 in binary

25 in decimal representation, which is 11001 in binary

2. The binary number is a sequence of digits. Gets reversed sequence of digits:

for 1101001 reversed sequence is 1001011

for 11001 reversed sequence is 10011

3. Represents each digit place as powers of 2 in the reversed sequence.

| For 1101001 | | For 11001 | |
|---|---|---|---|
| 1 | $2^0 = 1$ | 1 | $2^0 = 1$ |
| 0 | $2^1 = 2$ | 0 | $2^1 = 2$ |
| 0 | $2^2 = 4$ | 0 | $2^2 = 4$ |
| 1 | $2^3 = 8$ | 1 | $2^3 = 8$ |
| 0 | $2^4 = 16$ | 1 | $2^4 = 16$ |
| 1 | $2^5 =32$ | | |
| 1 | $2^6 =64$ | | |

4. Private Key can be represented as sum of points which have a ‘1’:

   105 = 1 + 8 + 32 + 64

   25 = 1 + 8 + 16

5. Public Key $Q$ calculation can be simplified by using combinations of ***Point Doubling*** and ***Point Addition*** operations instead of ***Scalar Point Multiplication*** defined by adding the point $G$ to itself $d$ times. Number of total operations will be decreased as:

totally 9 operations = 6 Point Doubling and 3 Point Addition operations for 105*G

105*G = 1*G + 8*G + 32*G + 64*G

totally 6 operations = 4 Point Doubling and 2 Point Addition operations for 25 *G

25*G = 1*G + 8*G + 16*G

where

2*G = Point Doubling (G)

4*G = Point Doubling (2*G)

8*G = Point Doubling (4*G)

16*G = Point Doubling (8*G)

etc.

6. Use “Double-and-add” algorithm to calculate Public key $Q = dG$

The formula for pseudo code is using reversed sequence of digits:

$$d = 2^0 d_0 + 2^1 d_1 + 2^2 d_2 + \ldots + 2^m d_m,\ [d_0, d_m] \in \{0,1\}$$

where *d* is Private Key in decimal representation

**Pseudo Code:**

$Q = 0$ // Point at infinity

*For i = 0 to m*

*If* $d_i = 1$

$Q = Q + G$ // using Unified formula (f.6.3.5) instead of Point Addition formula (f.6.3.1)

*End if*

$G = 2G$ // using Unified formula (f.6.3.5) instead of Point Doubling formula (f.6.3.2, f.6.3.2')

*End For*

*Return Q*

7. Final stage: Checking if PubKey point ($Q = dG$) is on curve.

$Q\ (x, y) \Rightarrow y^2 = (x^3+7) \bmod p$

***Note**: System is using Java (J2SE) BigInteger implementation of Modular arithmetic and Arithmetic primitives. To do so: Domain parameters should be converted from Hexadecimal to BiInteger representation before Unified formula is used*

**Pseudo Code for Efficient Unified Formula (f.6.3.5)**

*Defined: P(x1, y1) and P(x2, y2)*

*addY12 = y1 + y2*

*addX12 = x1 + x2*

*multiplyX12 = x1 * x2*

*subtractX12 = x1 – x2*

*subtractY12 = y1 – y2*

*If (addY12 – subtractX12) = 0*

*subtractX12 = negate(subtractX12)* // - (x1-x2) = x2-x1

*subtractY12 = negate(subtractY12)* // - (y1-y2) = y2-y1

*End if*

*Divisor = addY12 + subtractX12*

*Lambda = addX12^2 – multiplyX12 + subtractY12*

*Lambda = Lambda * Divisor ^ (-1)*

*x3 = Lambda^2 – addX12*

*subtractX13 = x1 – x3*

*y3 = (Lambda* subtractX13 – y1)*

*Return P(x3, y3)*

**Pseudo Code for Points Additions (f.6.3.1)**

*Defined: P(x1, y1), P(x2, y2), POINT_INFINITY*

*If P(x1, y1) equal POINT_INFINITY or P(x2, y2) equal POINT_INFINITY*

*Return POINT_INFINITY*

*End if*

*subtractX12 = x2 –x1*

*subtractY12 = y2 – y1*

*modInverse = subtractX12^ (-1) mod p*

*Lambda = subtractY12 * modInverse*

*Lambda = Lambda mod p*

*addX12 = x1 + x2*

*x3 = (Lambda^2 – addX12) mod p*

*subtractX13 = x1 – x3*

*y3 = (Lambda* subtractX13 – y1) mod p*

*Return P(x3, y3)*

**<u>Pseudo Code for Points Doubling (f.6.3.2, f.6.3.2')</u>**

*Defined: P(x1, y1)*

*modInverse = (2*y1)^(-1) mod p*

*Lambda = 3*x1^2* modInverse*

*Lambda = Lambda mod p*

*x3 = (Lambda^2 – 2*x1) mod p*

*subtractX13 = x1 – x3*

*y3 = (Lambda* subtractX13 – y1) mod p*

*Return P(x3, y3)*

***Note**: The results of all operations in the formulas must always be an integer*

**There are two forms of Public Key in scripts:**

Public Key should be presented in hexadecimal format

1. **Uncompressed PubKey** (old version) – are given as *04[x][y]*, 65 bytes, consisting of constant prefix *0x04*, followed by two 256-bit integers $x$ and $y$ (2 * 32 bytes), where $x$ and $y$ are 32 byte big-endian integers (as byte array) representing the Affine coordinates of $Q$ point on the curve
2. **Compressed PubKey** – are given as *[sign][x]*, 33 bytes, where *[sign]* is *0x02* if $y$ is even and *0x03* if $y$ is odd, 256-bit integer $x$ , where $x$ is 32 byte big-endian integer (as byte array) representing $x$ Affine coordinate of a $Q$ point on the curve.

### 6.3.3 Transaction (Message) Signature Generation

Given a message $m$ (Btc transaction Input) as a string in hexadecimal representation to be signed, the private key $d$ and $G$ , where $G$ is the base point on the curve, of order $n$

1. Choose a cryptographically secure random integer $k \in [1, n - 1]$. It is important that $k$ not be repeated in different signatures and that it not be guessable by a third party.

2. Compute the curve point $R(x_1, y_1) = kG$ using Scalar Point Multiplication (*see "Points operations:" 6.3.1*), where $x_1$ and $y_1$ are Affine coordinates of point $R$

3. Convert $x_1$ into integer and calculate $r = x_1 \mod n$. (Return to step 1 if $r = 0$)

4. Calculate $e = HASH(m)$, where $HASH$ is a cryptographic hash function, such as $SHA1$. Output of this step will be byte array in big-endian byte-order with length $L_{array}$

***Note**: The resulting sequence $q$ is converted to an integer value using the big-endian convention: if input bits are called $b\_0$ (leftmost) to $b\_(qLen-1)$ (rightmost), then the resulting value is*

$$b\_0 * 2^{(qLen-1)} + b\_1 * 2^{(qLen-2)} + \ldots + b\_(qLen-1) * 2^0$$

*where $qLen$ is the binary length of $q$*

5. Let's denote $z$ as $L_n$ leftmost bits of $e$ , where $L_n$ is the bit length of the group order $n$. To do so:

   5.1 Calculate the $L_e$ bit length of $e$ as integer $L_e = L_{array} * 8$ (1 byte = 8 bit)

   5.2 Convert $n$ from hexadecimal representation to big integer in decimal representation.

   5.3 Calculate $L_n$ bit length of $n$

   5.4 Convert $e$ to big integer in decimal representation.

   5.4 Make an integer $z$ as "*right shift*" of $e$ by $(L_e – L_n)$ bits if $L_n < L_e$

***Note**: 1. $Z$ can be greater than $n$ but not longer.*

*2. "right shift" is equivalent to operation when the resulting integer is divided by $2^{(Le – Ln)}$ (Euclidian division: the remainder is discarded)*

6. Compute $s = k^{-1}(z + dr) \mod n$ (Return to step 1 if $s = 0$). Do not forget to use "*modInverse*" operation in $s$ calculation.

7. Signature is $(r, s)$ pair of 256-bit numbers.

***Note**: 1. How a signature is to be encoded is not covered by the ECDSA standards themselves. A common way is to use a SEQUENCE of two INTEGERs, for r and s, in that order.*
*2. There is a Deterministic [3.5] approach to select random k, which is more secure. Deterministic means, instead of selecting a random scalar k in signing process, k is fixed with the same message and private key during signature generation but it is indistinguishable with random generated ones. The generation of k uses the hash of the message HASH(m) and private key as input to a deterministic pseudorandom number generator HMAC-DRBG, and output of the generation is used to yield k.*
*3. iDaemon system uses RNG hardware instead of Deterministic approach to enforce security.*

### DER-encoding of signature par (r, s) in the script

Given $r, s$ pair of 256-bit numbers (unsigned binary big integers) and $sighash$. DER encoded signature $sig$ can be calculated according to formula below:

*[sig] = [len_sig] [sequence = 0x30] [len_rs] [integer = 0x02] [len_r] [r_value] [integer = 0x02] [len_s][s_value][sighash]*

All elements are 1 byte except *r* & *s* which will be 32 or 33 bytes.
Where:

*r_value* – unsigned binary int, big-endian.

***Note**: some sources converts r into a little endian (see Appendix G) byte array. If the leading bit is not zero then prepend a zero value byte.*

*S_value* – unsigned binary int, big-endian.

***Note**: some sources converts **S** into a little endian (see Appendix G) byte array. If the leading bit is not zero then prepend a zero value byte.*
***Note**: the highest bit to be zero, if it isn't an extra zero byte is added.*

*Len_r* – number of bytes for *r* (always 20 or 21)

*len_s* – number of bytes for *s* (always 20 or 21)

*sequence* – always 0x30, ASN.1 tag identifier (20h = constructed + 10h = SEQUENCE and SEQUENCE OF)

*integer* – always 0x02, ASN.1 tag identifier

*len_rs = len_r + len_s + 2* (two extra bytes for the two integer bytes)

*len_sig = len_rs + 3* (three extra bytes for the *len_rs* byte, the sequence byte and the *sighash* byte)

*sighash* – A flag to Bitcoin signatures that indicates what parts of the transaction the signature signs. The unsigned parts of the transaction may be modified. Sighash Type codes see in *Appendix K*.

***Note**: r & S usually are 32 or 33 bytes. But can be smaller.*

- *If highest bit of 256-bit integer is set system has 33 bytes (probability is 1/2)*
- *If highest byte is greater than 0 and smaller than 128 system has 32 bytes (probability 127/256)*
- *If highest byte is 0 – system should take R as 248-bit integer and repeat these steps*

*IntDS will use **SIGHASH_ALL** sighash for normal single signature transactions.*

The signature *[sig]* is a first part of scriptSig.

### 6.3.4 Signature Verification

Given the signature pair $(r, s)$ on message $m$ (Btc transaction), public key $Q$, elliptic curve $E$, base point $G$, and $G$'s order $n$

1. Check that integers $r, s \in [1, n-1]$. If not, the signature is invalid.

2. Calculate $e = HASH(m)$, where $HASH$ is the same function used in the signature generation, such as *SHA1.*

3. Let's denote $z$ as $L_n$ leftmost bits of $e$ , where $L_n$ is the bit length of the group order $n$. (*see step 5 in point 6.3.3*)

2. Compute $w = s^{-1} \bmod n$. Do not forget to use "*modInverse*" operation in $w$ calculation.

3. Compute $u_1 = zw \bmod n$

   $u_2 = rw \bmod n$.

4. Calculate the curve point $C(x_1, y_1) = u_1\ G + u_2\ Q$, sing Simultaneous Elliptic Scalar Multiplication (*see "Points operations:" 6.3.1*)

5. Convert $x_1$ into integer and calculate $v = x_1 \bmod n$.

6. Compare $v$ and $r$, accept the signature only if $v = r$.

***Note**: Signature verification will be done in the last step of P2PKH Script implementation.*

# 7. Ways to Create Bitcoin Address

Bitcoin address is an identifier of 26-35 alphanumeric characters [2.15].

## 7.1 Single signature Btc Address

In summary, Btc address is the double hash of the public key for internal representation. The Btc address is represented externally in ASCII using Base58Check encoding and can be shared with others.

The algorithms used to make a Btc address from a public key are the Secure Hash Algorithm (SHA) and the RACE Integrity Primitives Evaluation Message Digest (RIPEMD), specifically SHA256 and RIPEMD160.

To create a Btc address, public key $PubKey$ is a mandatory input. Bitcoin originally only used uncompressed keys, but since v0.6 compressed are now used. Btc address is a prefix byte of $0x00$, the $RIPEMD160(SHA256(PubKey))$ hash and then a checksum postfix.

The checksum $checksum$ is the first 4 bytes of the

$$checksum = SHA256(SHA256(0x00 <RIPEMD160(SHA256(PubKey))>))$$

The full byte string of Btc Address is

$$0x00 <RIPEMD160(SHA256(PubKey))> <checksum>$$

which is then encoded using Base58:

$$BtcAddress = Base58(0x00 <RIPEMD160(SHA256(PubKey))> <checksum>)$$

Steps involved in creating a version 1 single signature Btc address are as follows:

1. **Double hash the public key**

   Starting with the public key PubKey, we compute the SHA256 hash and then compute the RIPEMD160 hash of the result, producing a 160-bit (20-byte) number:

   $$dbleHash = RIPEMD160(SHA256(PubKey))$$

   where $PubKey$ is the public key in hexadecimal representation (*see point 6.3.2*)

   Example:

   Assume Uncompressed Public key PubKey =
   0450863AD64A87AE8A2FE83C1AF1A8403CB53F53E486D8511DAD8A04887E5B23522CD470243453A299FA9E77237716103ABC11A1DF38855ED6F2EE187E9C582BA6

   SHA256 (PubKey) = 600FFE422B4E00731A59557A5CCA46CC183944191006324A447BDB2D98D4B408

RIPEMD160 hashing of the result of SHA256 gives:

dbleHash = 010966776006953D5567439E5E39F86A0D273BEE

**2. Base58Check encoding**

A modified Base58 binary-to-text encoding known as Base58Check is used for encoding Bitcoin addresses.

Base58Check is a Base58 encoding format, which has a built-in error-checking code i.e. the checksum. The checksum is an additional four bytes added to the end of the data that is being encoded.

Following are the detailed steps for Base58Check encoding:

**i. Append version byte**

Add a prefix called “version byte” in front of output from step 1. Version byte identifies the type of data that is encoded. In case of a bitcoin address the prefix is zero (0x00 in hex). Refer Appendix J for commonly used prefixes.

Appending version byte in front of output from Step 1 gives us:

*beforeChkSum = version byte (+) dbleHash*

Example:

beforeChkSum = 00 (+) 010966776006953D5567439E5E39F86A0D273BEE

beforeChkSum = 00010966776006953D5567439E5E39F86A0D273BEE

**ii. Double hash the extended result with version byte**

Perform SHA256 hash twice on the result from above step.

*chksumHash = SHA256(SHA256 (beforeChkSum))*

Example:

chksumHash = SHA256(SHA256(00010966776006953D5567439E5E39F86A0D273BEE))

chksumHash = D61967F63C7DD183914A4AE452C9F6AD5D462CE3D277798075B107615C1A8A30

**iii. Get the checksum**

Take the first four bytes from the above output. This is the checksum.

*Checksum = first four bytes of chksumHash*

Example:

checksum = D61967F6

iv. **Get the final result**
Append the checksum at the end of output from step i.

*result = beforeChkSum (+) checksum*

<u>Example</u>:

result = 00010966776006953D5567439E5E39F86A0D273BEE (+) D61967F6
result = 00010966776006953D5567439E5E39F86A0D273BEED61967F6

v. **Get the Btc address**
Base58 encode the above result from a byte string into Base58 string to get the Bitcoin address

*btcAddress = Base58(result)*

<u>Example</u>:

btcAddress = Base58(00010966776006953D5567439E5E39F86A0D273BEED61967F6)
btcAddress = 16UwLL9Risc3QfPqBUvKofHmBQ7wMtjvM

## 7.2 Multi signature Btc Address

This point can be done in the scope of future development. Will need some researching activity.

Base58Check encoding [3.3]

Pay-to-script-hash (P2SH): payload is RIPEMD160(SHA256(redeemScript the wallet knows how to spend)); version 0x05 (these addresses begin with the digit ‘3’)

# 8. Stack-Based Btc Scripting Language

There are a small programs inside each transaction which should be executed to decide if transaction is valid. This program is written in **Stack-Based Btc Scripting Languge** (Script). IntDS should verify and implement Scripts to produce valid transactions. Valid transaction is valid if other systems in the Btc network will also verify and accept it.
The **Stack-Based Btc Scripting Languge** is stateless, in that there is no state prior to execution of the Script, or state saved after execution of the Script. Therefore, all the information needed to execute a Script is contained within the script. Btc Scripting Language is called a Stack-Based language because it uses a data structure called a stack.

A stack allows two operations: ***Push*** and ***Pop***. ***Push*** adds an item on top of the stack. ***Pop*** removes the top item from the stack. The Script language has approximately 80 different Opcodes (*see Appendix D*). It includes arithmetic, bitwise operations, string operations, conditionals, stack manipulation and etc.

This paragraph will describe some standard types of transaction Scripts.

A ***Locking Script*** (*scriptPubKey*) and an ***Unlocking Script*** *(scriptSig)* are two types of scripts to validate transactions. A *Locking Script* is an encumbrance placed on an Output, and it specifies the conditions that must be met to spend the Output in the future. An *Unlocking Script* is a script that “solves”, or satisfies, the conditions placed on an Output by a *Locking Script* and allows the Output to be spent. Unlocking Script are part of every transaction Input and most of the time they contain a digital signature produced from their private key.
IntDS will validate transactions by executing the Locking and Unlocking scripts together. For each input in the transaction, the validation functionality will first retrieve the UTXO referenced by the Input. That UTXO contains a Locking Script defining the conditions required to spend it. The system will then take the Unlocking Script contained in the input that is attempting to spend this UTXO and execute the two Scripts.

## 8.1 Script for Pay to Public Key Hash (P2PKH) Transaction

An ***Unlock Script*** (***scriptSig***) is provided by IntDS to resolve encumbrance. A ***Lock Script*** (***scriptPubKey***) is found in a Trx Output and is the encumbrance that must be fulfilled to spend the Output. The two scripts together would form the ***Combined Validation Script***:

*[scriptSig] [scriptPubKey]*

The result will be **TRUE** if Unlock Script has a valid signature from Private Key which corresponds to Public Key and Public Key corresponds to Btc Address from Lock Script.

### 8.1.1 “scriptSig” structure in case P2PKH

scriptSig formula: *scriptSig = PUSHDATA [sig] PUSHDATA [pubKey]* where:

*[sig] = [signature][sighash]*

*PUSHDATA* is the next byte contains the number of bytes to be pushed onto the stack.

The signature is encoded with DER (*see point 6.3.3*). Public key is represented as plain bytes (*see point 6.3.2*). Table below describes the example of scriptSig on the byte-level for one Input:

| | | |
|---|---|---|
| Signature Length in hex | PUSHDATA 48 | 48 *($48_{16} = 4x16^1 + 8x16^0 = 72$ bytes)* |
| [signature] (DER [1.15]) | sequence = 0x30 | 30 |
| | length *rs* | 45 |
| | integer = 0x02 | 02 |
| | length *r* | 20 |
| | *r* value | 26 33 25 fc bd 57 9f 5a 3d 0c 49 aa 96 53 8d 95 62 ee 41 dc 69 0d 50 dc c5 a0 af 4b a2 b9 ef cf |
| | integer = 0x02 | 02 |
| | length *s* | 21 |
| | *s* value | **00** fd 8d 53 c6 be 9b 3f 68 c7 4e ed 55 9c ca 31 4e 71 8d f4 37 b5 c5 c5 76 68 c5 93 0e 14 14 05 02 |
| [sighash] | SIGHASH_ALL | 01 |
| Public Key length in hex: | PUSHDATA 41 | 41 *($41_{16} = 4x16^1 + 1x16^0 = 65$ bytes)* |
| Public Key [pubKey] | type = 0x04 for uncompressed key type = 0x02 if Y is even and 0x03 if Y is odd for compressed key | 04 |
| | X | 14 e3 01 b2 32 8f 17 44 2c 0b 83 10 d7 87 bf 3d 40 4c fb d0 70 4f 13 5b 6a d4 b2 d3 ee 75 13 |
| | Y | 10 f9 81 92 6e 53 a6 e8 c3 9b d7 d3 fe fd 57 6c 54 3c ce 49 3c ba c0 63 88 f2 65 1d 1a ac bf cd |

### 8.1.2 "scriptPubKey" structure in case P2PKH

scriptPubKey formula:

*scriptPubKey = OP_DUP OP_HASH160 PUSHDATA [pubKeyHash] OP_EQUALVERIFY OP_CHECKSIG*

where:

*[pubKeyHash] = RIPEMD160(SHA256(PubKey))* is a part of Btc address (*see point 7.1*)

*PUSHDATA* is the next byte contains the number of bytes to be pushed onto the stack.
*OP_DUP, OP_HASH160, OP_EQUALVERIFY, OP_CHECKSIG* are Opcodes. Opcodes values can be found in *Appendix D*.
Table below describes the example of scriptPubKey on the byte-level for one Output.
In this example, Btc address is **1KKKK6N21Xko48zWKuQKXdvSsCf95ibHFa**

| | | |
|---|---|---|
| OP_DUP | 0x76 | 76 |
| OP_HASH160 | 0xa9 | a9 |
| [pubKeyHash] length in hex | PUSHDATA 14 | 14 *($14_{16} = 1x16^1 + 4x16^0 = 20$ bytes)* |
| [pubKeyHash] 20 byte | | c8 e9 09 96 c7 c6 08 0e e0 62 84 60 0c 68 4e d9 04 d1 4c 5c |
| OP_EQUALVERIFY | 0x88 | 88 |
| OP_CHECKSIG | 0xac | ac |

Steps to make scriptPubKey from given Btc address:

1. Decode the base58 encoding (similar to Base64). You should have 25 bytes.
2. Check that the 1st byte is 0x00 (the version byte of Bitcoin)
3. Check that the last 4 bytes are a correct checksum of the rest. (Or, "take the first 4 bytes of a double-SHA256 of the first 21 bytes of the decoded data.")
4. Take the middle 20 bytes and insert it into the following scriptPubKey.

```
OP_DUP OP_HASH160 <x> OP_EQUALVERIFY OP_CHECKSIG
```

### 8.1.3 Execution Steps of Combined Validation Script in case P2PKH

***Combined Validation Script*** formula:

*PUSHDATA [sig] PUSHDATA [pubKey] OP_DUP OP_HASH160 PUSHDATA [pubKeyHash] OP_EQUALVERIFY OP_CHECKSIG*

where *[pubKeyHash]* is Btc Address from Output of previous Trx which should be spent in current Trx.

Script Execution Steps:

| Step | Stack | Script | Execution Pointer |
|---|---|---|---|
| 1 | *[sig]* | ***PUSHDATA [sig]*** *PUSHDATA [pubKey] OP_DUP OP_HASH160 PUSHDATA [pubKeyHash] OP_EQUALVERIFY OP_CHECKSIG* | **Execution starts.** **PUSHDATA** pushes the value ***[sig]*** to the top of the stack |
| 2 | *[pubKey]* *[sig]* | *PUSHDATA [sig]* ***PUSHDATA [pubKey]*** *OP_DUP OP_HASH160 PUSHDATA [pubKeyHash] OP_EQUALVERIFY OP_CHECKSIG* | Script execution moving to the right with each step. **PUSHDATA** pushes the value ***[pubKey]*** to the top of the stack, on top of *[sig]* |

| | | | |
|---|---|---|---|
| 3 | *[pubKey]*<br>*[pubKey]*<br>*[sig]* | *PUSHDATA [sig] PUSHDATA [pubKey]* ***OP_DUP*** *OP_HASH160 PUSHDATA [pubKeyHash] OP_EQUALVERIFY OP_CHECKSIG* | ***OP_DUP*** operator duplicates the top item in the stack, the resulting value is pushed to the top of the stack |
| 4 | *[pubKeyHash]*<br>*[pubKey]*<br>*[sig]* | *PUSHDATA [sig] PUSHDATA [pubKey] OP_DUP* ***OP_HASH160*** *PUSHDATA [pubKeyHash] OP_EQUALVERIFY OP_CHECKSIG* | ***OP_HASH160*** operator hashed the top item in the stack with RIPEMD160(SHA256(pubKey)), the resulting value *[pubKeyHash]* is pushed to the top of the stack. |
| 5 | *[pubKeyHash]*<br>*[pubKeyHash]*<br>*[pubKey]*<br>*[sig]* | *PUSHDATA [sig] PUSHDATA [pubKey] OP_DUP OP_HASH160* ***PUSHDATA [pubKeyHash]*** *OP_EQUALVERIFY OP_CHECKSIG* | **PUSHDATA** pushes the value ***[pubKeyHash]*** from the script on top of the value *[pubKeyHash]* calculated previously from OP_HASH160 of the *[pubKey]* |
| 6a | *[pubKey]*<br>*[sig]* | *PUSHDATA [sig] PUSHDATA [pubKey] OP_DUP OP_HASH160 PUSHDATA [pubKeyHash]* ***OP_EQUALVERIFY*** *OP_CHECKSIG* | ***OP_EQUALVERIFY*** operator compares the *[pubKeyHash]* encumbering the transaction with *[pubKeyHash*] calculated from the user's *[pubKey*]. This proves that Public Key is valid. Both are removed and execution continues in **step 7a** if they match, if not match go to **step 6b**. |
| 6b | *FALSE*<br>*[pubKey]*<br>*[sig]* | *PUSHDATA [sig] PUSHDATA [pubKey] OP_DUP OP_HASH160 PUSHDATA [pubKeyHash]* ***OP_EQUALVERIFY*** *OP_CHECKSIG* | Pushes FALSE to the top of the stack. **Execution stops.** |
| 7a | TRUE | *PUSHDATA [sig] PUSHDATA [pubKey] OP_DUP OP_HASH160 PUSHDATA [pubKeyHash] OP_EQUALVERIFY* ***OP_CHECKSIG*** | ***OP_CHECKSIG*** operator checks that the signature *[sig]* matches the *[pubKey]* and pushes TRUE to the top of the stack if true. This proves that the signature is valid. **Execution stops.**<br>Go to **step 7b** if false. |
| 7b | FALSE | *PUSHDATA [sig] PUSHDATA [pubKey] OP_DUP OP_HASH160 PUSHDATA [pubKeyHash] OP_EQUALVERIFY* ***OP_CHECKSIG*** | Pushes FALSE to the top of the stack. **Execution stops.** |

## 8.2 Pay to Public Key (P2PK)

This point can be done in the scope of future development. Will need some researching activity.

## 8.3 Multi-Signature Transaction Script

The general form of a **Locking Script** (scriptPubKey or redeemScript) setting an M-of-N multi-signature condition is:

*M <Public Key 1> <Public Key 2> ... <Public Key N> N OP_CHECKMULTISIG*

where N is the total number of listed public keys and M is the threshold of required signatures to spend the output. Signatures must be placed in the scriptSig using the same order as their corresponding public keys were placed in the scriptPubKey or redeemScript.

A Locking Script setting a 2-of-3 multi-signature condition looks like this:

3. *<Public Key A> <Public Key B> <Public Key C> 3 OP_CHECKMULTISIG*

The Locking script above can be satisfied with an **Unlocking Script** (scriptSig) containing pairs of signatures and public keys:

*OP_0 <Signature B> <Signature C>*

or any combination of two signatures from the private keys corresponding to the three listed public keys.

Signatures must be placed in the scriptSig using the same order as their corresponding public keys were placed in the scriptPubKey or redeemScript.

**Note:** The prefix OP_0 is required because of a bug in the original implementation of CHECKMULTISIG where one item too many is popped off the stack. It is ignored by CHECKMULTISIG and is simply a placeholder.

The two scripts together would form the combined validation script below:

*OP_0 <Signature B> <Signature C>*

4. *<Public Key A> <Public Key B> <Public Key C> 3 OP_CHECKMULTISIG*

This point can be done in the scope of future development. Will need some researching activity.

## 8.4 Data Output (OP_RETURN) Script

This point can be done in the scope of future development. Will need some researching activity.

## 8.5 Pay to Script Hash (P2SH)

This point can be done in the scope of future development. Will need some researching activity.

# 9. Methods of the Creation of Different Type’s Transactions.

This Paragraph describes the transaction’s creation of the each type step by step with examples according to trxs types list (see Appendix B).

## 9.1 Block’s Anatomy

| Field No. | Field | Size | Description | Value or Value example |
|---|---|---|---|---|
| | Magic no | 4 | The first element of the block is a 4 byte magic number, whose value is always 0xD9B4BEF9. Bitcoin protocol uses little-endian representation for integers, therefore reading the file as binary would result in following sequence of bytes: 0xF9 0xBE 0xB4 0xD9 | value always 0xD9B4BEF9 |
| | Blocksize | 4 | The magic number is then followed by 4 bytes that denote the length of the block in bytes. The block length is the number of bytes following up to the end of block. | If the 4 bytes following the magic no. are 30 75 00 00, converting them to little endian gives us: 0x00007530 which is 30000 bytes. |
| | Blockheader | 80 | consists of 6 items | |
| ***Note that all fields in the block header are represented in little-endian format. The examples below have been converted from little-endian format to normal integer representation.*** | | | | |
| Blkhdr-1 | Version | 4 | Block version number. A new version number will be specified when the software (FOS Daemon) is upgraded. | Version 1 Blocks will have version number 0x00000001 |
| Blkhdr-2 | hashPrevBlock | 32 | 256-bit hash of the previous block header. | Example: |

| | | | | |
|---|---|---|---|---|
| Blkhdr-3 | hashMerkleRoot | 32 | 256-bit hash based on all of the transactions in the block. | |
| Blkhdr-4 | Time | 4 | Current timestamp as seconds since 1970-01-01T00:00 UTC. | If the time field is 0x4e24878f (example), it means 1311016847 in decimal. An epoch converter [..] will show this time in human readable format as Mon, 18 Jul 2011 19:20:47 GMT |
| Blkhdr-5 | Bits | 4 | Current target in compact format. | Example value for Bits: 0x1a0abbcf.<br>This is the compact format of target is a special kind of floating-point encoding using 3 bytes mantissa, the leading byte as exponent (where only the 5 lowest bits are used) and its base is 256.<br>So, in this case the exponent is 0x1a = 26<br><br>The mantissa is 0x0abbcf<br>So the exponent says this is a 26 byte base 256 integer. To convert this into it's integer value, we would have pad it with 23 zeros to get:<br>0a bb cf 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |

| | | | | |
|---|---|---|---|---|
| | | | | This large number is an even larger number when converted from base 256 to decimal. i.e. 0x0a * 256^26 + 0xbb * 256^25 + 0xcf*256^24 which in decimal representation is close to 4.4155582e+63 |
| Blkhdr-6 | Nonce | 4 | 32-bit number (starts at 0). It is the number that is incremented/changed in mining to create different block headers, hence different block hashes. | Example: 0x0aa64562 |
| | Transaction counter | 1 – 9 bytes | Non negative integer. VarInt: 1, 3, 5 or 9 bytes depending on size. Denotes the number of transactions in this block. | 40. The first byte is < 0xfd, therefore the storage length for this integer is 1 byte and the value is in-fact represented by the first byte itself i.e. 0x40 (or 64 in decimal). |
| | Transactions | Variable | (non empty) list of transactions. | For structure of each transaction, refer section 9.2. |

## 9.2 Introduction in a Transaction's Anatomy

**Transactions** [2.1] are cryptographically signed records that reassign ownership of Bitcoins to new addresses. Transactions have **Inputs** – records which reference the unspent funds from other previous transactions – and **Outputs** – records which determine the new owner of the transferred Bitcoins, and which will be referenced as inputs in future transactions as those funds are respent. Outputs are tied to transaction identifiers (TXIDs), which are the hashes of signed transactions.

Each **Input** must have a cryptographic digital signature (`scriptSig`) that unlocks the funds from the prior transaction. Only the person possessing the appropriate private key is able to create a satisfactory signature; this in effect ensures that funds can only be spent by their owners.

Each **Output** determines which Bitcoin address (or other criteria, `scriptPubKey`) is the recipient of the funds.

The full value of an **Input** is always spent; a Trx cannot spend part of the value. Likewise all **Outputs** are either spent or unspent, they can't be partially spent. A Trx "spends" the **Outputs** which are referenced in the input portion of the Trx. A Trx creates new spendable "unspent outputs" listed in the output portion of the Trx.

In a transaction, the sum of all **Inputs** must be equal to or greater than the sum of all **Outputs**. If the Inputs exceed the Outputs, the difference is considered a transaction fee, and is redeemable by whoever first includes the transaction into the block chain.

The IntDS supports different transaction types which are described in Appendix B.

Bitcoin uses a Stack-Based Btc Scripting Language (*see point 8*) for transactions. Forth-like, Script is simple, stack-based, and processed from left to right. It is purposefully not Turing-complete, with no loops.

A new transaction is valid if

- `scriptSig` of the current input,

***Note****: scriptSig contains a signature and a public key in case P2PKH. scriptSig = [sig] [pubKey]*

combinated with

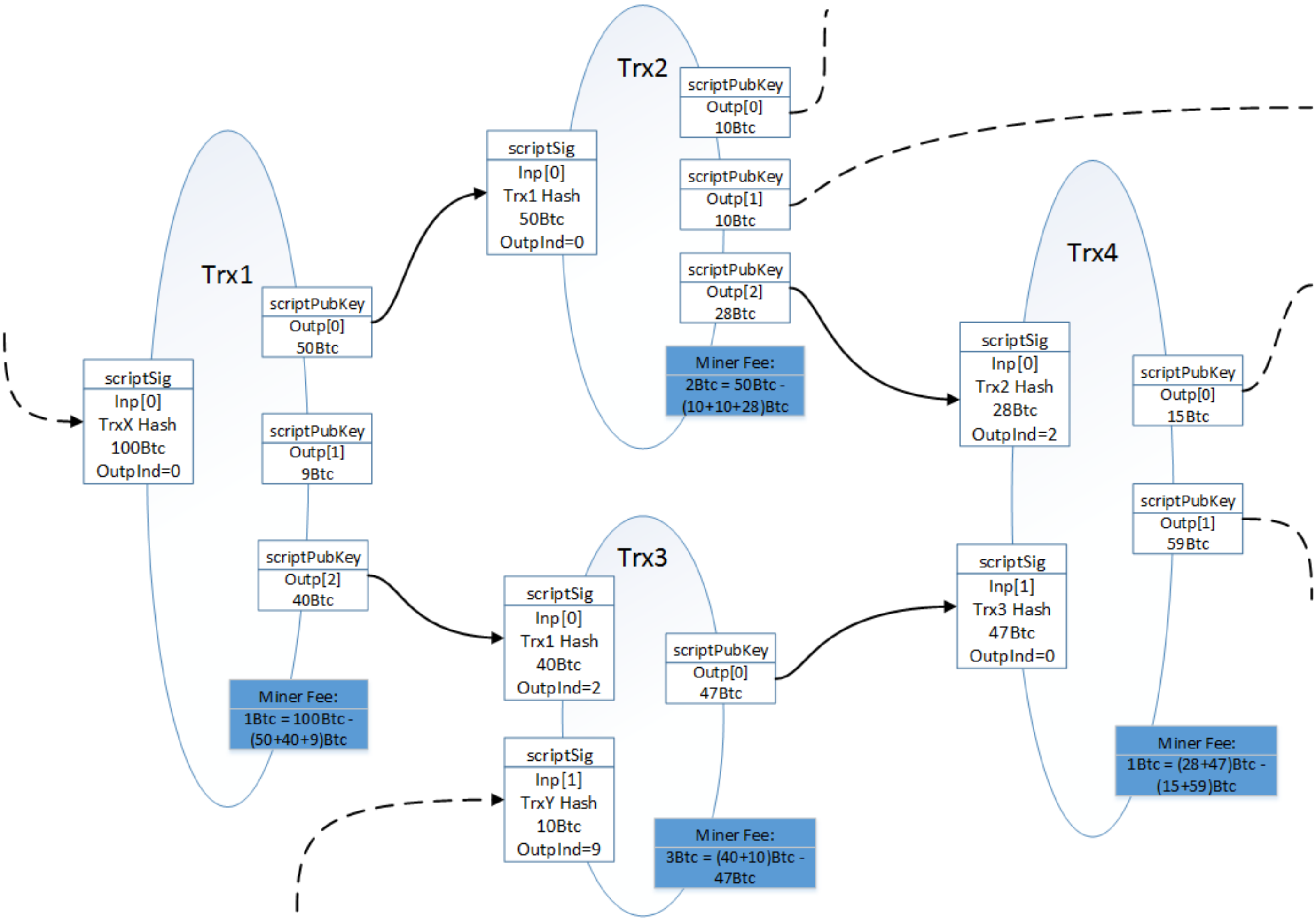
- `scriptPubKey` of the previous output,

evaluates to true.

Script: `scriptSig + scriptPubKey == true`

The IntDS supports different types of script pairs (scriptSig/scriptPubKey) which are described in Appendix E.

The diagram below (Pic. 9.2.1) shows sequence of transactions with different numbers of **Inputs** and **Outputs**. The difference between sums of **Inputs** and **Outputs** is considered a transaction fee, and is redeemable by first Miner which includes the transaction into the block chain.




Pic. 9.2.1

## General Format of Btc Transaction

Table 9.2.1 shows General Format of a Btc transaction (inside a block) [2.1]:

Table 9.2.1

| **Field Order** | **Field** | **Size in bytes** | **Value or Value example** | **Description** |
|---|---|---|---|---|
| **1** | Version number | 4 bytes | 01 00 00 00 | Currently 1 |
| **2** | Number of Inputs | 1, 3, 5 or 9 bytes | 01 (1 byte, 1 Input example) | Positive integer. VarInt: 1, 3, 5 or 9 bytes depending on size |
| **3** | List of Inputs | <Number of Inputs>- Inputs Lengths, where each Input Length > 41-49 bytes | | |

| **For each Input:** from 0 to <Number of Inputs> - 1 | | | | |
|---|---|---|---|---|
| Inp-1 | Previous Transaction hash | 32 bytes | `Example:`<br>`ec cf 7e 30 34 18 9b 85 19 85 d8 71 f9 13 84 b8 ee 35 7c d4 7c 30 24 73 6e 56 76 eb 2d eb b3 f2` | Doubled SHA256-hashed of all of the raw byte data of (previous) to-be-used transaction. This value is not stored directly in the block-chain and should be computed by IntDS |
| Inp-2 | Previous Trx Output index | 4 bytes | `Example:`<br>`01 00 00 00`<br>(Output number 2 = Output index 1) | Non negative integer, index refers to an Output in the previous transaction which we want to redeem. Counting from zero. |
| Inp-3 | Input scriptLength | 1, 3, 5 or 9 bytes | `Example: 8c`<br>The length is 140 bytes, or 0x8c ($8c_{16} = 8x16^1 + 12x16^0 = 140$ bytes) | Non negative integer. VarInt: 1, 3, 5 or 9 bytes depending on size.<br>Length of Script of the current Input. |
| Inp-4 | Input Script / scriptSig | <Input scriptLength> bytes | `Example:`<br>`49 30 46 02 21 00 9e 03 39 f7 2c 79 3a 89 e6 64 a8 a9 32 df 07 39 62 a3 f8 4e da 0b d9 e0 20 84 a6 a9 56 7f 75 aa 02 21 00 bd 9c ba ca 2e 5e c1 95 75 1e fd fa c1 64 b7 62 50 b1 e2 13 02 e5 1c a8 6d d7 eb d7 02 0c dc 06 01 41 04 50 86 3a d6 4a 87 ae 8a 2f e8 3c 1a f1 a8 40 3c b5 3f 53 e4 86 d8 51 1d ad 8a 04 88 7e 5b 23 52 2c d4 70 24 34 53 a2 99 fa 9e 77 23 77 16 10 3a bc 11 a1 df 38 85 5e d6 f2 ee 18 7e 9c 58 2b a6` | Raw byte code data for the current Input Script. (*See 8.1.1 scriptSig structure*) |
| | **scriptSig structure in case P2PKH:** | | | |
| Scrl-1 | PUSHDATA Signature Length | 1 byte | Example:<br>49 | The next byte contains the number of bytes (Signature Length in hex) to be |

| | | | | |
|---|---|---|---|---|
| | | | *($49_{16} = 4x16^1 + 9x16^0 = 73$ bytes)* | pushed onto the stack. Signature Length = length ***rs*** + 3 (three extra bytes for the <length ***rs***> byte, the <sequence> byte and the [sighash] byte) |
| Scrl-2 | **[signature] , DER-encoded:** | | | |
| | sequence | 1 byte | 30 | ASN.1 tag identifier (20h = constructed + 10h = SEQUENCE and SEQUENCE OF) sequence = always 0x30 |
| | length ***rs*** | 1 byte | Example: 46 | length ***rs*** = length ***r*** + length ***s*** + 2 (two extra bytes for the two integer bytes) |
| | integer | 1 byte | 02 | ASN.1 tag identifier, integer = always 0x02 |
| | length ***r*** | 1 byte | Example: 21 | number of bytes for ***r*** (always 20 or 21) |
| | ***r*** value | <=32-33 bytes | Example: **00** 9e 03 39 f7 2c 79 3a 89 e6 64 a8 a9 32 df 07 39 62 a3 f8 4e da 0b d9 e0 20 84 a6 a9 56 7f 75 aa | Signature ***r*** value, unsigned binary int, big-endian<br>***Note:*** *some sources converts* ***r*** *into a little endian byte array. If the leading bit is not zero then prepend a zero value byte.* |
| | Integer | 1 byte | 02 | ASN.1 tag identifier, integer = always 0x02 |
| | length ***s*** | 1 byte | Example: 21 | number of bytes for ***s*** (always 20 or 21) |
| | ***s*** value | <=32-33 bytes | Example: **00** bd 9c ba ca 2e 5e c1 95 75 1e fd fa c1 64 b7 62 50 b1 e2 13 02 e5 1c a8 6d d7 eb d7 02 0c dc 06 | Signature ***s*** value, unsigned binary int, big-endian<br>***Note:*** *some sources converts* ***r*** *into a little endian byte array. If the leading bit is not zero then prepend a zero value byte.* |
| | [sighash] | 1 byte | Example: SIGHASH_ALL 01 | A flag to Bitcoin signatures that indicates what parts of the transaction the signature signs. The unsigned parts of the transaction may be modified. There are three base [sighash] types: SIGHASH_ALL = 01, SIGHASH_NONE, SIGHASH_SINGLE |
| Scrl-3 | PUSHDATA<br>Public Key Length | 1 byte | Example: 41 *($41_{16} = 4x16^1 + 1x16^0 = 65$ bytes)* | The next byte contains the number of bytes (Public Key Length in hex) to be pushed onto the stack. |
| Scrl-4 | **Public Key [pubKey]:** | | | |

| | | | | |
|---|---|---|---|---|
| | Public Key type | 1 byte | Example:<br>04 | type = 0x04 for uncompressed key<br>type = 0x02 if Y is even and 0x03 if Y is odd for compressed key |
| | *X* value | 32 bytes | Example:<br>50 86 3a d6 4a 87 ae 8a 2f e8 3c 1a f1 a8 40 3c b5 3f 53 e4 86 d8 51 1d ad 8a 04 88 7e 5b 23 52 | Public key *X* value as 32 byte big-endian integers, plain bytes representation |
| | *Y* value | 32 bytes | Example:<br>2c d4 70 24 34 53 a2 99 fa 9e 77 23 77 16 10 3a bc 11 a1 df 38 85 5e d6 f2 ee 18 7e 9c 58 2b a6 | Public key *Y* value as 32 byte big-endian integers, plain bytes representation |
| Inp-5 | sequenceNumber | 4 bytes | FFFFFFFF | Always expected to be 0xFFFFFFFF; irrelevant unless transaction’s LockTime is > 0 |
| **4** | Number of Outputs | 1, 3, 5 or 9 bytes | 01 (1 byte, 1 Output example) | Positive integer. VarInt: 1, 3, 5 or 9 bytes depending on size |
| **5** | List of Outputs | <Number of Outputs>- Outputs Lengths, where each Output Length > 9-18 bytes | | |
| **For each Output:** from 0 to <Number of Outputs> - 1 | | | | |
| Otp-1 | value | 8 bytes | `Example: 60 5a f4 05 00 00 00 00`<br>99900000 Satoshis = 0.999 Btc | Non negative 8 bytes integer (64 bit integer). The value of output is the number of Satoshi (1Btc=$10^8$ Sat) in hex, which is stored in the value field in little-endian form. |
| Otp-2 | Output scriptLength | 1, 3, 5 or 9 bytes | `Example: 19`<br>The length is 25 bytes, or 0x19 ($19_{16} = 1 \times 16^1 + 9 \times 16^0 = 25$ bytes) | Non negative integer. VarInt: 1, 3, 5 or 9 bytes depending on size.<br>Length of Script of the current Output. |
| Otp-3 | Output Script / scriptPubKey | <Output scriptLength> bytes | `Example: 76 a9 14 09 70 72 52 44 38 d0 03 d2 3a 2f 23 ed b6 5a ae 1b b3 e4 69 88 ac` | Output Script (*See 8.1.2 scriptPubKey structure*) |
| | **scriptPubKey structure in case P2PKH:** | | | |
| Scr-1 | OP_DUP | 1 byte | 76 | Opcode = 0x76 dduplicates the top stack item. |

| | | | | |
|---|---|---|---|---|
| Scr-2 | OP_HASH160 | 1 byte | a9 | Opcode = 0xa9, the input is hashed twice: first with SHA-256 and then with RIPEMD-160. |
| Scr-3 | PUSHDATA [pubKeyHash] Length | 1 byte | Example: 14 *($14_{16} = 1x16^1 + 4x16^0 = 20$ bytes)* | The next byte contains the number of bytes ([pubKeyHash] Length in hex) to be pushed onto the stack. |
| Scr-4 | [pubKeyHash] | 20 bytes | Example: 09 70 72 52 44 38 d0 03 d2 3a 2f 23 ed b6 5a ae 1b b3 e4 69 | [pubKeyHash] = RIPEMD160(SHA256(PubKey)) is a part of Btc address |
| Scr-5 | OP_EQUALVERIFY | 1 byte | 88 | Opcode = 0x88, Returns 1 if the inputs are exactly equal, 0 otherwise. Marks transaction as invalid if top stack value is not true (1). |
| Scr-6 | OP_CHECKSIG | 1 byte | ac | Opcode = 0xac, The signature used by OP_CHECKSIG must be a valid signature for this hash and public key. |
| **6** | LockTime | 4 bytes | 00 00 00 00 | if non-zero and sequence numbers are <0xFFFFFFFF>: block height or timestamp when transaction is final |

The first Input of the first transaction in the block is also called “coinbase” (its content was ignored in earlier versions). The Outputs of the first transaction spend the mined bitcoins for the block. See “Coinbase Transaction” definition in the *Glossary*.

## 9.3 Transaction Fees and Priority (default settings)

A transaction may be safely sent without fees if these conditions are met:

- It is smaller than 1,000 bytes.
- All outputs are 0.01 Btc or larger.
- Its priority is large enough.

Otherwise, the reference implementation will round up the transaction size to the next thousand bytes and add a fee of 0.1 mBtc (0.0001 Btc) per thousand bytes [2.11]. As an example, a fee of 0.1 mBtc (0.0001 Btc) would be added to a 746 byte transaction, and a fee of 0.2 mBtc (0.0002 Btc) would be added to a 1001 byte transaction. Users may increase the default 0.0001 Btc/kB fee setting, but cannot control transaction fees for each transaction. Note that a typical transaction is 500 bytes, so the typical transaction fee for low-priority transactions is 0.1 mBtc (0.0001 Btc), regardless of the number of bitcoins sent.

50,000 bytes in the block are set aside for the highest-priority transactions, regardless of transaction fee. Transactions are added highest-priority-first to this section of the block.

Then transactions that pay a fee of at least 0.00001 Btc/kb are added to the block, highest-fee-per-kilobyte transactions first, until the block is not more than 750,000 bytes big.

The remaining transactions remain in the Miner’s “memory pool”, and may be included in later blocks if their priority or fee is large enough. All of the default settings may be changed if a miner wants to create larger or smaller blocks containing more or fewer free transactions.

Transactions need to have a priority above 57,600,000 to avoid the enforced limit. This threshold is written in the code as COIN * 144 / 250, suggesting that the threshold represents a one day old, 1 Btc coin (144 is the expected number of blocks per day) and a transaction size of 250 bytes.

Transaction priority is calculated as a value-weighted sum of input age, divided by transaction size in bytes:

```
priority = sum(input_value_in_base_units * input_age)/trx_size_in_bytes
```

where:

`input_value_in_base_units` – Btc value of Input is multiplied by $10^8$. All values in the Bitcoin network are integers in Satoshis (1E-8 BTC).

`Input_age` – number of confirmations. Number of blocks are published to the block-chain after a Trx with this Input was included in a block that is published to the block-chain.

`Trx_size_in_bytes` – size of current transaction in bytes for which priority should be calculated.

So, for example, a transaction that has 2 Inputs, one of 5 Btc with 10 confirmations, and one of 2 Btc with 3 confirmations, and has a size of 500bytes, will have a priority of

```
(500000000 * 10 + 200000000 * 3) / 500 = 11,200,000
```

Currently, the minimum Btc amount per Trx is 0.0000546 Btc.

## 9.4 Steps to Create Usual Single-Sig Transactions

A single signature Btc address is an address that is associated with one ECDSA private key. Sending bitcoins from this address requires signature from the associated private key.

**Single-sig Transaction** is defined in the IntDS as transaction which is sending some Bitcoins from the Single-sig Btc address to the Single-sig Btc address. All Inputs and Outputs of this Trx should correspond to **Pay-to-Public-Key-Hash (P2PKH)** type of script Pairs.

Steps to create a single signature transaction involve identifying the inputs that can be used and then sending the bitcoins to the desired Single-sig Btc address.

### 9.4.1 Steps to create Transaction by using RPC from FOS Daemon

#### Identify the inputs

To spend a certain number of bitcoins, we need to calculate if there is sufficient balance in the form of unspent transaction outputs.

Use the Single-sig Transaction Management SubSystem (STrxMSS) to get a list of UTXOs. Identify the ones that can be used to generate the required output.

Ex: Alice needs to send 0.15 BTC to Bob.

Alice’s wallet application gets the list of UTXOs available to Alice as follows:

| No. | Transaction ID | Index no. | Value in Bitcoins |
|---|---|---|---|
| 1 | ebadfaa92f1fd29e2fe296eda702c48bd11ffd52313e986e99ddad9084062167 | 1 | 0.08 |
| 2 | 6596fd070679de96e405d52b51b8e1d644029108ec4cbfe451454486796a1ecf | 0 | 0.165 |
| 3 | 74d788804e2aae10891d72753d1520da1206e6f4f20481cc1555b7f2cb44aca0 | 1 | 0.05 |
| 4 | b2affea89ff82557c60d635a2a3137b8f88f12ecec85082f7d0a1f82ee203ac4 | 1 | 0.1 |

Alice’s wallet application can select combination of transactions from the above list (no. 3 and no.4) or use just one transaction (no. 2) as input(s) to spend 0.15 BTC.

Note that the inputs selected should account for transaction fee as well. Hence the combination of transactions 3 & 4 will not work as it will not include transaction fees.

#### Calculate fees associated with this transaction

**Transaction fee:** This is calculated based on the size of the transaction in kilobytes, not the value of transaction in bitcoins. Refer section [8.2] for details of calculating transaction fees.

Transaction fees are implied as the difference between the sum of inputs and the sum of outputs i.e. the data structure of a transaction does not have a field for fees.

*Transaction Fees = Sum (Inputs) – Sum (Outputs)*

Hence, the sum of value of UTXOs should be greater than or equal to the sum of bitcoins to send and the transaction fee.

*Total value of selected UTXOs > = Bitcoins to spend + Transaction fee*

**System Fees (optional):** Depending on the application logic, there might be an additional component of system fees. For example, Alice's wallet application charges a minimal amount as company fees.

Hence,

*Total value of selected UTXOs > = Bitcoins to spend + Transaction fees + System fees (optional)*

### Calculate change associated with this transaction

For this example, let's assume Alice's wallet application selects transaction no. 2 to spend 0.15 BTC. Also, assume that the transaction fee is 0.01 BTC and system fee is 0.001 BTC

Hence,

Total value of UTXO(s) = 0.165

BTC to spend = 0.15

System fee = 0.001

Transaction fee = 0.01 (Implied)

So, total BTC Alice would spend = 0.15 + 0.001 + 0.01 = 0.161

Thus, Alice should get 0.004 (0.165 – 0.161) change back, when the transaction is done.

*Change = Total value of UTXOs used as inputs – Total BTC that would be spend*

Note that change can be 0 in some cases.

### Change address

A change address is the address which receives the excess BTC that is leftover after spending the required amount of BTC and transaction fees.

We need to create a change address for the user to get the reminder of BTC back. Since transaction fee is implied, failure of explicitly stating the change address will lead to counting the “leftover” BTC as transaction fee.

Note that there is no need for a change address if there is no “change” left to give back to the user.

### Finalize inputs and outputs

In our example,

Input(s) = Transaction no. 2 from list of UTXOs:

| Input No. | Txn ID | Index no. | Value in Bitcoins |
|---|---|---|---|
| 1 | 6596fd070679de96e405d52b51b8e1d644029108ec4cbfe451454486796a1ecf | 0 | 0.165 |
| **Total** | | | **0.165** |

Outputs = 0.15 BTC to Bob,

0.004 BTC to Alice’s change address (optional)

0.001 BTC as company fee (optional)

0.01 BTC as transaction fee (Implied. **DO NOT** include while making the raw transaction)

| Output No. | BTC Address | Value in Bitcoins | Description |
|---|---|---|---|
| 1 | 1Bobadd4RoXcnBv9hnQ4Y2C1an6NJ4UrjX | 0.15 | Bob’s payment |
| 2 | 1ChngaddabccnBm9ikK4J6C5rdloNJ4Klop | 0.004 | Alice’s change address |
| 3 | 1Cmpyaddalliou89ikkk0iouiy67ttN9iKkojgh | 0.001 | Wallet company fee |
| **Total** | | **0.155** | |

### Create transaction

Now that we have a list of inputs and outputs, we can create the raw transaction. Refer [5.7.2] for the RPC ***createrawtransaction***.

```
bitcoin-cli -testnet createrawtransaction ‘’
  ‘[
    {
      “txid”: “6596fd070679de96e405d52b51b8e1d644029108ec4cbfe451454486796a1ecf”,
```

```
        "vout" : 0
      }
   ]'
  '{"1Bobadd4RoXcnBv9hnQ4Y2C1an6NJ4UrjX":0.15,
  "1ChngaddabccnBm9ikK4J6C5rdloNJ4Klop":0.004,
  "1Cmpyaddalliou89ikkk0iouiy67ttN9iKkojgh":0.001}'

```

Result:

```
01000000011da9283b4ddf8d89eb996988b89ead56cecdc44041ab38bf787f1206cd90b51e0000000000ffffffff
01405dc600000000001976a9140dfc8bafc8419853b34d5e072ad37d1a5159f58488ac00000000
```

The ***createrawtransaction*** RPC will usually return a serialized transaction format encoded as hex. In case of error, it will return NULL.

### Sign transaction

The raw transaction hex code obtained as output from above step will be signed with Alice’s private key.

**IntDS** will probably be using its own implementation for signing transactions [6.3.2]. For this example, consider the ***signrawtransaction*** RPC from FOS Daemon.

***Note****: The second optional argument (may be null) for signrawtransaction is an array of previous transaction outputs that this transaction depends on but may not yet be in the block chain. We assume that we will be dealing with confirmed transactions only. Hence, this argument can be omitted. However, in case we want to include this argument, we can get the scriptPubKey of the output by using the decoderawtransaction RPC [5.7.2].*

For this example, let’s assume:

- scriptPubKey for the previous transaction output (with txid “6596fd070679de96e405d52b51b8e1d644029108ec4cbfe451454486796a1ecf “) is “76a9144a06df74729aef1dce5e4641960da3a439d2460b88ac”
- Alice’s private key is “93Fu1spd9rCgBc4RbdkxxGcznA4bnQXM6mebzpYqaFFT2P89Cqi”

Hence, signrawtransaction will be as follows:

```
bitcoin-cli –testnet signrawtransaction
‘01000000011da9283b4ddf8d89eb996988b89ead56cecdc44041ab38bf787f1206cd90b51e0000000000ffffffff014
05dc600000000001976a9140dfc8bafc8419853b34d5e072ad37d1a5159f58488ac00000000’
‘[
{“txid”:”6596fd070679de96e405d52b51b8e1d644029108ec4cbfe451454486796a1ecf”,
“vout”:0,
“scriptPubKey”:”76a9144a06df74729aef1dce5e4641960da3a439d2460b88ac”},
]’
‘[
“93Fu1spd9rCgBc4RbdkxxGcznA4bnQXM6mebzpYqaFFT2P89Cqi”
]’
```

This step will return a raw hex code that can be broadcasted to the network.

```
{
    “hex” :
“01000000011da9283b4ddf8d89eb996988b89ead56cecdc44041ab38bf787f1206cd90b51e000000006a4730440
2200ebea9f630f3ee35fa467ffc234592c79538ecd6eb1c9199eb23c4a16a0485a20220172ecaf6975902584987d
295b8dddf8f46ec32ca19122510e22405ba52d1f13201210256d16d76a49e6c8e2edc1c265d600ec1a64a45153d4
5c29a2fd0228c24c3a524ffffffff01405dc600000000001976a9140dfc8bafc8419853b34d5e072ad37d1a5159f
58488ac00000000”,
    “complete” : true
}
```

### Send transaction

Use the ***sendrawtransaction*** RPC [5.7.2] to broadcast the signed transaction to the peer-to-peer network.

```
bitcoin-cli –testnet sendrawtransaction
01000000011da9283b4ddf8d89eb996988b89ead56cecdc44041ab38bf787f1206cd90b51e000000006a47304402
200ebea9f630f3ee35fa467ffc234592c79538ecd6eb1c9199eb23c4a16a0485a20220172ecaf6975902584987d2
95b8dddf8f46ec32ca19122510e22405ba52d1f13201210256d16d76a49e6c8e2edc1c265d600ec1a64a45153d45
```

```
c29a2fd0228c24c3a524ffffffff01405dc60000000000001976a9140dfc8bafc8419853b34d5e072ad37d1a5159f5
8488ac00000000
```

This step will return a transaction id for this transaction.

```
f5a5ce5988cc72b9b90e8d1d6c910cda53c88d2175177357cc2f2cf0899fbaad
```

### 9.4.2 Steps to crate Raw-Transaction in case IntDS implementation

All Inputs and Outputs of new Trx are correspond to **P2PKH** type only. It means that **scriptSig** and **scriptPubkey** should be calculated for each Input and Output according to formulas from paragraph “8.1”.

Some data should be calculated and prepared before IntDS starts creating of new Trx.

**Preparation steps**:

1. Get unspent Outputs (list of UTXOs) for addresses you want to send money from.
2. Ensure you have the private/public keys pairs for every addresses you want to send money from.
3. Determine the right Btc amount value per each recipient Btc address.
4. Calculate miner fees associated with this transaction (*see point 9.3.1*)
5. Calculate IntDS fees associated with this transaction if needed (optional)
6. Ensure you have the private/public keys and Btc address for IntDS fees if fees exists (optional)
7. Calculate change associated with this transaction (*see point 9.3.1*) if needed (optional)
8. Ensure you have the private/public keys and Btc address for change if change exists (optional)

**Steps to create new Transaction, which should be hashed and signed** (see general transaction format in the *Table 9.2.1*):

1. Consider that IntDS has necessary data from preparation stage. **TrxNew** is new transaction which should be created. **TrxPrev** is previos transaction from which IntDS want to redeem an Outputs.
2. Add 4 bytes version number. Currently is 1.
   **TrxNew Result**:

| | |
|---|---|
| version | `01 00 00 00` |

3. Add 1, 3, 5 or 9 bytes (depending on integer size) Inputs number.
   For example 2 Inputs is 1 byte `02`
   **TrxNew Result**:

| | |
|---|---|
| version | `01 00 00 00` |

| | |
|---|---|
| Inputs number | `02` |

4. Add all necessary Inputs without scriptSig (UTXO). For each Input:

4.1. Add 32-bytes double hash of previos ransaction **TrxPrev** from which IntDS want to redeem an Output.
For example: `ec cf 7e 30 34 18 9b 85 19 85 d8 71 f9 13 84 b8 ee 35 7c d4 7c 30 24 73 6e 56 76 eb 2d eb b3 f2`

***Note**: This value should be computed by IntDS. Some sources presents this value in little-endian (reversed). Should be checked before implementation.*

4.2. Add 4-byte field denoting the Output index IntDS want to redeem from the transaction with the above hash. For example Output number 2 = Output index 1: `01 00 00 00`

4.3. Add one byte for scriptSig length as 0x00 (it will be replaced in the future steps): `00`

4.4. Add a 4-byte field denoting the sequence.
This is currently always set to 0xffffffff: `ff ff ff ff`

4.5. Repeat steps 4.1 – 4.4 for second Input in this example.
For example:
32 bytes hash of previos Trx: `be 66 e1 0d a8 54 e7 ae a9 33 8c 1f 91 cd 48 97 68 d1 d6 d7 18 9f 58 6d 7a 36 13 f2 a2 4d 53 96`
Output index 0: `00 00 00 00`

**<u>Final TrxNew Result for two Inputs</u>**:

| | |
|---|---|
| version | `01 00 00 00` |
| Inputs number | `02` |
| Previos Trx hash for Input0 | `ec cf 7e 30 34 18 9b 85 19 85 d8 71 f9 13 84 b8 ee 35 7c d4 7c 30 24 73 6e 56 76 eb 2d eb b3 f2` |
| Previos Trx Output index | `01 00 00 00` |
| scriptSig length | `00` |
| sequence | `ff ff ff ff` |
| Previos Trx hash for Input1 | `be 66 e1 0d a8 54 e7 ae a9 33 8c 1f 91 cd 48 97 68 d1 d6 d7 18 9f 58 6d 7a 36 13 f2 a2 4d 53 96` |
| Previos Trx Output index | `00 00 00 00` |
| scriptSig length | `00` |
| sequence | `ff ff ff ff` |

5. Add 1, 3, 5 or 9 bytes (depending on integer size) Outputs number. For example 1 Output is 1 byte `01`
6. Add all desired Outputs. For each Output:

6.1. Write an 8-byte field (64 bit integer, little-endian) containing the amount IntDS want to redeem from the specified Output.
For example: 0.999 Btc = 99,900,000 Stoshis$_{10}$ = 5645a60$_{16}$
Add additional zeros to make 8 bytes: 00 00 00 00 05 64 5a 60

Represent this value in a little-endian: `60 5a 64 05 00 00 00 00`

6.2 Make scriptPubKey from recepeint Btc address **1KKKK6N21Xko48zWKuQKXdvSsCf95ibHFa** according to example in the *8.1.2 point*: `c8 e9 09 96 c7 c6 08 0e e0 62 84 60 0c 68 4e d9 04 d1 4c 5c`

6.3 Add length of scriptPubKey 25 bytes = 0x19 : `19`

6.4 Add scriptPubKey: `c8 e9 09 96 c7 c6 08 0e e0 62 84 60 0c 68 4e d9 04 d1 4c 5c`

6.5 Repeat steps 6.1-6.4 for each Output. Nothing should be repeated for current example, because there is only one output.

7. Write 4-byte LockTime field: `00 00 00 00`

**Final TrxNew Result for two Inputs and One Output**:

| | |
|---|---|
| version | 01 00 00 00 |
| Inputs number | 02 |
| Previos Trx hash for Input0 | ec cf 7e 30 34 18 9b 85 19 85 d8 71 f9 13 84 b8 ee 35 7c d4 7c 30 24 73 6e 56 76 eb 2d eb b3 f2 |
| Previos Trx Output index | 01 00 00 00 |
| scriptSig length | 00 |
| sequence | ff ff ff ff |
| Previos Trx hash for Input1 | be 66 e1 0d a8 54 e7 ae a9 33 8c 1f 91 cd 48 97 68 d1 d6 d7 18 9f 58 6d 7a 36 13 f2 a2 4d 53 96 |
| Previos Trx Output index | 00 00 00 00 |
| scriptSig length | 00 |
| sequence | ff ff ff ff |
| Outputs number | 01 |
| Btc value | 60 5a 64 05 00 00 00 00 |
| scriptPubKey length | 19 |
| scriptPubKey | c8 e9 09 96 c7 c6 08 0e e0 62 84 60 0c 68 4e d9 04 d1 4c 5c |
| LockTime | 00 00 00 00 |

8. Copy current transaction **TrxNew** as result of step 7 to make template. **TrxNewTempl** is a copy of **TrxNew.**

9. Sign the transaction. For each Input:

9.1 Create template of transaction for first input “Input0”. Copy **TrxNewTempl** as result of step 8 to make template. **TrxCopy** is a copy of **TrxNewTempl.**

9.2 Create subscript from previos ransaction **TrxPrev**. Subscript is **scriptPubKey** of the Output IntDS wants to redeem Btc.
For example: `76 a9 14 01 09 66 77 60 06 95 3d 55 67 43 9e 5e 39 f8 6a 0d 27 3b ee 88 ac`

9.3 Replace one byte for scriptSig length from step 4.3 with the length of subscript from step 9.2 in **TrxCopy**.
For current example: length is 25 byte = 0x19. Replace `00` by `19`

9.4 Insert subscript from step 9.2 after scriptSig length before sequence field
**TrxCopy Result for Input0**:

| | |
|---|---|
| version | 01 00 00 00 |
| Inputs number | 02 |
| Previos Trx hash for Input0 | ec cf 7e 30 34 18 9b 85 19 85 d8 71 f9 13 84 b8 ee 35 7c d4 7c 30 24 73 6e 56 76 eb 2d eb b3 f2 |
| Previos Trx Output index | 01 00 00 00 |
| scriptSig length | 19 |
| Subscript=scriptPubKey of previos Trx Output | 76 a9 14 01 09 66 77 60 06 95 3d 55 67 43 9e 5e 39 f8 6a 0d 27 3b ee 88 ac |
| sequence | ff ff ff ff |
| Previos Trx hash for Input1 | be 66 e1 0d a8 54 e7 ae a9 33 8c 1f 91 cd 48 97 68 d1 d6 d7 18 9f 58 6d 7a 36 13 f2 a2 4d 53 96 |
| Previos Trx Output index | 00 00 00 00 |
| scriptSig length | 00 |
| sequence | ff ff ff ff |
| Outputs number | 01 |
| Btc value | 60 5a 64 05 00 00 00 00 |
| scriptPubKey length | 19 |
| scriptPubKey | c8 e9 09 96 c7 c6 08 0e e0 62 84 60 0c 68 4e d9 04 d1 4c 5c |
| LockTime | 00 00 00 00 |

9.5 Append 4-byte Sighash type code in little-endian representation in the end of **TrxCopy**. SIGHASH_ALL=0x00000001 type is used as default for normal single-sig transaction (*see Appendix K*). little-endian representation: `01 00 00 00`

**TrxCopy Result for Input0**:

| | |
|---|---|
| version | 01 00 00 00 |
| Inputs number | 02 |
| Previos Trx hash for Input0 | ec cf 7e 30 34 18 9b 85 19 85 d8 71 f9 13 84 b8 ee 35 7c d4 7c 30 24 73 6e 56 76 eb 2d eb b3 f2 |
| Previos Trx Output index | 01 00 00 00 |
| scriptSig length | 19 |
| Subscript=scriptPubKey of previos Trx Output | 76 a9 14 01 09 66 77 60 06 95 3d 55 67 43 9e 5e 39 f8 6a 0d 27 3b ee 88 ac |
| sequence | ff ff ff ff |
| Previos Trx hash for Input1 | be 66 e1 0d a8 54 e7 ae a9 33 8c 1f 91 cd 48 97 68 d1 d6 d7 18 9f 58 6d 7a 36 13 f2 a2 4d 53 96 |
| Previos Trx Output index | 00 00 00 00 |
| scriptSig length | 00 |
| sequence | ff ff ff ff |
| Outputs number | 01 |
| Btc value | 60 5a 64 05 00 00 00 00 |
| scriptPubKey length | 19 |
| scriptPubKey | c8 e9 09 96 c7 c6 08 0e e0 62 84 60 0c 68 4e d9 04 d1 4c 5c |
| LockTime | 00 00 00 00 |

| | |
|---|---|
| Temporarily appended Sighash type | `01 00 00 00` |

9.6 Serialize **TrxCopy.** This serialization is double-SHA256 hash of **TrxCopy.**
**Result:** `aa c3 21 5d c6 c0 ed 93 92 63 9d 79 cc ce 31 d3 2f 74 7c 74 81 26 d1 be 57 c7 d3 7e 94 8d 50 db`
9.7 Create a DER-encoded signature for hash from step 9.6. (*see point 6.3.3*)
9.8 Make a **scriptSig** (*see point 8.1.1*)
For example:
`4930460221009e0339f72c793a89e664a8a932df073962a3f84eda0bd9e02084a6a9567f75aa022100bd9cbaca2e5ec195751efdfac164b76250b1e21302e51ca86dd7ebd7020cdc0601410450863ad64a87ae8a2fe83c1af1a8403cb53f53e486d8511dad8a04887e5b23522cd470243453a299fa9e77237716103abc11a1df38855ed6f2ee187e9c582ba6`

***Note:*** *This scriptSig example is not correspond to hash from step 8.6. This scriptSig can not be used for JUnit test.*

9.9 Verify the signature of this Input by using **scriptSig** from step 9.8 and **scriptPubKey** from step 9.2 (*see point 8.1.3*)
9.10 Go to step 9.11 if execution of scripts validation from step 9.9 return true otherwise repeat steps 9.1-9.9
9.11 Replace one byte for **scriptSig** length from step 4.3 with the length of actual **scriptSig** from step 9.8 in **TrxNew**. For current example: length is 140 bytes = 0x8C. Replace `00` by `8c`
9.12 Insert actual **scriptSig** from step 9.8 after scriptSig length before sequence field in **TrxNew**

**TrxNew Result for signed Input0**:

| | |
|---|---|
| version | `01 00 00 00` |
| Inputs number | `02` |
| Previos Trx hash for Input0 | `ec cf 7e 30 34 18 9b 85 19 85 d8 71 f9 13 84 b8 ee 35 7c d4 7c 30 24 73 6e 56 76 eb 2d eb b3 f2` |
| Previos Trx Output index | `01 00 00 00` |
| scriptSig length | `8c` |
| scriptSig | `49 30 46 02 21 00 9e 03 39 f7 2c 79 3a 89 e6 64 a8 a9 32 df 07 39 62 a3 f8 4e da 0b d9 e0 20 84 a6 a9 56 7f 75 aa 02 21 00 bd 9c ba ca 2e 5e c1 95 75 1e fd fa c1 64 b7 62 50 b1 e2 13 02 e5 1c a8 6d d7 eb d7 02 0c dc 06 01 41 04 50 86 3a d6 4a 87 ae 8a 2f e8 3c 1a f1 a8 40 3c b5 3f 53 e4 86 d8 51 1d ad 8a 04 88 7e 5b 23 52 2c d4 70 24 34 53 a2 99 fa 9e 77 23 77 16 10 3a bc 11 a1 df 38 85 5e d6 f2 ee 18 7e 9c 58 2b a6` |
| sequence | `ff ff ff ff` |
| Previos Trx hash for Input1 | `be 66 e1 0d a8 54 e7 ae a9 33 8c 1f 91 cd 48 97 68 d1 d6 d7 18 9f 58 6d 7a 36 13 f2 a2 4d 53 96` |

| | |
|---|---|
| Previos Trx Output index | 00 00 00 00 |
| scriptSig length | 00 |
| sequence | ff ff ff ff |
| Outputs number | 01 |
| Btc value | 60 5a 64 05 00 00 00 00 |
| scriptPubKey length | 19 |
| scriptPubKey | c8 e9 09 96 c7 c6 08 0e e0 62 84 60 0c 68 4e d9 04 d1 4c 5c |
| LockTime | 00 00 00 00 |

9.13 Repeat steps 9.1-9.12 for each Input. Repeat steps for Input1 in this example.

10 Have a final result of signed transaction

**<u>Final TrxNew Result for two signed Inputs and One Output</u>**:

| | |
|---|---|
| version | 01 00 00 00 |
| Inputs number | 02 |
| Previos Trx hash for Input0 | ec cf 7e 30 34 18 9b 85 19 85 d8 71 f9 13 84 b8 ee 35 7c d4 7c 30 24 73 6e 56 76 eb 2d eb b3 f2 |
| Previos Trx Output index | 01 00 00 00 |
| scriptSig length | 8c |
| scriptSig | 49 30 46 02 21 00 9e 03 39 f7 2c 79 3a 89 e6 64 a8 a9 32 df 07 39 62 a3 f8 4e da 0b d9 e0 20 84 a6 a9 56 7f 75 aa 02 21 00 bd 9c ba ca 2e 5e c1 95 75 1e fd fa c1 64 b7 62 50 b1 e2 13 02 e5 1c a8 6d d7 eb d7 02 0c dc 06 01 41 04 50 86 3a d6 4a 87 ae 8a 2f e8 3c 1a f1 a8 40 3c b5 3f 53 e4 86 d8 51 1d ad 8a 04 88 7e 5b 23 52 2c d4 70 24 34 53 a2 99 fa 9e 77 23 77 16 10 3a bc 11 a1 df 38 85 5e d6 f2 ee 18 7e 9c 58 2b a6 |
| sequence | ff ff ff ff |
| Previos Trx hash for Input1 | be 66 e1 0d a8 54 e7 ae a9 33 8c 1f 91 cd 48 97 68 d1 d6 d7 18 9f 58 6d 7a 36 13 f2 a2 4d 53 96 |
| Previos Trx Output index | 00 00 00 00 |
| scriptSig length | 8c |
| scriptSig | 49 30 46 02 21 00 cf 4d 75 71 dd 47 a4 d4 7f 5c b7 67 d5 4d 67 02 53 0a 35 55 72 6b 27 b6 ac 56 11 7f 5e 78 08 fe 02 21 00 8c bb 42 23 3b b0 4d 7f 28 a7 15 cf 7c 93 8e 23 8a fd e9 02 07 e9 d1 03 dd 90 18 e1 2c b7 18 0e 01 41 04 2d aa 93 31 5e eb be 2c b9 b5 c3 50 5d f4 c6 fb 6c ac a8 b7 56 78 60 98 56 75 50 d4 82 0c 09 db 98 8f e9 99 7d 04 9d 68 72 92 f8 15 cc d6 e7 fb 5c 1b 1a 91 13 79 99 81 8d 17 c7 3d 0f 80 ae f9 |
| sequence | ff ff ff ff |
| Outputs number | 01 |

| | |
|---|---|
| Btc value | 60 5a 64 05 00 00 00 00 |
| scriptPubKey length | 19 |
| scriptPubKey | c8 e9 09 96 c7 c6 08 0e e0 62 84 60 0c 68 4e d9 04 d1 4c 5c |
| LockTime | 00 00 00 00 |

11 Serialize the **TrxNew** into hexadecimal format.
12 Propagate the transaction **TrxNew**.

***Note**: 1. This scenario is based on bitcoin wiki article [2.23]. Other solution with more clear detailed explanation was not found at this moment. The biggest complication is the signature appears in the middle of the transaction, which raises the question of how to sign the transaction before you have the signature. Another complication is transaction with many Inputs. This scenario is clear in case one Input but there is high probability of another sequence of steps in case many Inputs.*

*5. RPC function “sendrawtransaction” can be used instead of steps 11 and 12 (see point 5.7.2)*

## 9.5 Steps to Create Multi-Sig transactions

See point 8.3

This point can be done in the scope of future development. Will need some researching activity.

## 9.6 Ways to Create Contracts

This point can be done in the scope of future development. Will need some researching activity.

A distributed contract is a method of using Bitcoin to form agreements with people via the block chain. Contracts allow you to solve common problems in a way that minimizes trust. Minimal trust often makes things more convenient by allowing human judgements to be taken out of the loop, thus allowing complete automation.

This point can be updated according to scope of future development. Will need some researching activity.

A distributed contract is a method of using Bitcoin to form agreements with people via the block chain. Contracts allow you to solve common problems in a way that minimizes trust. Minimal trust often makes things more convenient by allowing human judgements to be taken out of the loop, thus allowing complete automation.

### 9.6.1 Bitcoin Contract Basics

**Scripts**: Every transaction in Bitcoin has one or more inputs and outputs. Each input/output has a small, pure function associated with it called a script. Scripts can contain signatures over simplified forms of the transaction itself.

**Lock time:** Every transaction can have a **lock time** associated with it. This allows the transaction to be pending: until an agreed-upon future time, specified either as a block index or as a timestamp (the same field is used for both, but values less than 500 million are interpreted as a block index). If a transaction’s lock time has been reached, we say it is final.

- Non-zero nLockTime less than 500 million is interpreted as the block height, meaning the transaction is not included in the blockchain prior to the specified block height.
- Non-zero nLockTime greater than 500 million is interpreted as the Unix Epoch timestamp (seconds since Jan-1-1970) and the transaction is not included in the blockchain prior to the specified time.

**Sequence number:**

Each transaction input has a **sequence number**.

- In a normal transaction that just moves value around, the sequence numbers are all UINT_MAX and the lock time is zero.
- If the lock time has not yet been reached, but all the sequence numbers are UINT_MAX, the transaction is also considered final.
- In order to enforce lock time to a transaction, sequence number should be less than UINT_MAX, else the lock time field will be ignored.
- Sequence numbers can be used to issue new versions of a transaction without invalidating other inputs signatures, e.g., in the case where each input on a transaction comes from a different party, each input may start with a sequence number of zero, and those numbers can be incremented independently.

***Note:*** *UINT_MAX is the maximum value for an object of type unsigned int. Value = 4294967295U. Thus, UINT_MAX is an unsigned int (At least in the [−32767, +32767] range, at least 16 bits in size, but unsigned.)*

These features can be used to achieve the following:

- You send a transaction with a LockTime in the future and a sequence number of 0. The transaction is then not considered by the network to be “final”, and it can’t be included in a block until the specified LockTime is reached.
- If you ever want to lock the transaction permanently, you can set the sequence number to UINT_MAX. Then the transaction is considered to be final, even if LockTime has not been reached.

### 9.6.2 Types of contracts

Contracts can be of varying types depending on how we embed the conditions of the contract within the transaction.

There are two general patterns for safely creating contracts:

1. Transactions are passed around outside of the P2P network, in partially-complete or invalid forms.
2. Two transactions are used: one (the contract) is created and signed but not broadcast right away. Instead, the other transaction (the payment) is broadcast after the contract is agreed to lock in the money, and then the contract is broadcast.

***Note:*** *All bitcoin addresses, private keys, scripts used in examples below are purely for demonstration purpose. These transactions have not been tested on the actual network.*

#### Single signature transaction with nLockTime

***Note:*** *Locktime and nLocktime are synonyms.*

*BIP-0065 (in draft status at the time of writing i.e. August 2015) describes a new opcode (OP_CHECKLOCKTIMEVERIFY) for the Bitcoin scripting system that allows a transaction output to be made unspendable until some point in the future [2.19]. We might need to upgrade this contract functionality once this BIP has been approved.*

This transaction is the simplest form of contract that can be used with single signature and the Locktime feature. The funds are locked up in a BTC address until the time specified in nLockTime field is reached. The recipient can spend funds only after nLockTime has been reached and the transaction has been accepted in the blockchain.

***The responsibility of storing such a transaction and broadcasting it when valid, lies with the sender or the recipient of the transaction.***

**Example**:

In the context of a web wallet, this theme can be used for cold storage functionality.

Alice (the user) opts for locking her funds from a particular wallet (10 BTC) for period of 3 months from the current date.

This can be achieved with following 2 transactions:

Tx1: sends funds from wallet to cold storage

Tx2 (the contract): sends funds from cold storage back to Alice after the Locktime is elapsed

Details of these 2 transactions are as follows:

(Refer section [5.7.2] for detailed descriptions of RPCs used.)

6. Send funds to Cold Storage (simple single signature transaction):
   a) eWallet system creates a dedicated BTC address to receive Alice’s funds that need to be locked up.
   b) Alice creates, signs & broadcasts a transaction (Tx1) that spends all funds from her wallet to the system generated ‘Cold Storage’ BTC address.
      i. Create raw transaction

         Inputs for Tx1 = all UTXO from Alice’s wallet

         Output for Tx1

         - Output amount: Sum of funds in Alice’s wallet.
         - Output address: Cold Storage BTC address.

```
Bitcoind createrawtransaction
‘[{“txid”:”aaa…”,”vout”:1},{“txid”:”bbb…”,”vout”:0}]’ ‘{“csbtcaddr…”:9.8,”cmpfees…”:0.2}’
```

The output will be a raw transaction hex code.

```
0100000001bfd27404f34449a53c6a5d0a4b972342e0193441dbfb66c8b5f7806d3bb3497600000000000ffffffff01
00f2052a010000001976a914249604bc668da89a7d2d494b89fba47f529c52f788ac00000000
```

Note that depending on the eWallet application logic, one of the outputs can include the company fees for this cold storage functionality.

      ii. Sign raw transaction.
         The raw transaction hex code obtained as output from above step will be signed with Alice’s private key. iDaemon will probably be using its own implementation for signing transactions [6.3.2].

         For this example, consider the ***signrawtransaction*** RPC from FOS Daemon.

         ***Note****: The second optional argument (may be null) for* ***signrawtransaction*** *is an array of previous transaction outputs that this transaction depends on but may not yet be in the block chain. We assume that we will be dealing with confirmed transactions only. Hence, this argument can be omitted. However, in case we want to include this argument, we can get the scriptPubKey of the output by using the* ***decoderawtransaction*** *RPC [5.7.2].*

```
bitcoind decoderawtransaction
‘0100000001bfd27404f34449a53c6a5d0a4b972342e0193441dbfb66c8b5f7806d3bb3497600000000000ffffffff0
100f2052a010000001976a914249604bc668da89a7d2d494b89fba47f529c52f788ac00000000’

{

   “txid” : “ef7c0cbf6ba5af68d2ea239bba709b26ff7b0b669839a63bb01c2cb8e8de481e”,
```

```
    “version” : 1,
    “locktime” : 0,
    “vin” : [
        {
            “txid” : “d7c7557e5ca87d439e9ab6eb69a04a9664a0738ff20f6f083c1db2bfd79a8a26”,
            “vout” : 0,
            “scriptSig” : {
                “asm” :
“3045022100ee69171016b7dd218491faf6e13f53d40d64f4b40123a2de52560feb95de63b902206f23a0919471eaa
1e45a0982ed288d374397d30dff541b2dd45a4c3d0041acc001
03a7c1fd1fdec50e1cf3f0cc8cb4378cd8e9a2cee8ca9b3118f3db16cbbcf8f326”,
                “hex” :
“483045022100ee69171016b7dd218491faf6e13f53d40d64f4b40123a2de52560feb95de63b902206f23a0919471e
aa1e45a0982ed288d374397d30dff541b2dd45a4c3d0041acc0012103a7c1fd1fdec50e1cf3f0cc8cb4378cd8e9a2c
ee8ca9b3118f3db16cbbcf8f326”
            },
            “sequence” : 4294967295
        }
    ],
    “vout” : [
        {
            “value” : 5.00,
            “n” : 0,
            “scriptPubKey” : {
                “asm” : “OP_DUP OP_HASH160 56847befbd2360df0e35b4e3b77bae48585ae068
OP_EQUALVERIFY OP_CHECKSIG”,
                “hex” : “76a91456847befbd2360df0e35b4e3b77bae48585ae06888ac”,
                “reqSigs” : 1,
                “type” : “pubkeyhash”,
                “addresses” : [
                    “aaa…”
                ]
            }
        },
        {
            “value” : 5.00,
            “n” : 1,
            “scriptPubKey” : {
```

```
                “asm” : “OP_DUP OP_HASH160 2b14950b8d31620c6cc923c5408a701b1ec0a020
OP_EQUALVERIFY OP_CHECKSIG”,
                “hex” : “76a9142b14950b8d31620c6cc923c5408a701b1ec0a02088ac”,
                “reqSigs” : 1,
                “type” : “pubkeyhash”,
                “addresses” : [
                    “bbb…”
                ]
            }
        }
    ]
}
```

Note that Alice’s private key will be generated from the mnemonic seed according to Type 1 Deterministic approach [6.3.1]. For this step, assume Alice’s private key be “93Fu1spd9rCgBc4RbdkxxGcznA4bnQXM6mebzpYqaFFT2P89Cqi”

```
bitcoind signrawtransaction
‘0100000001bfd27404f34449a53c6a5d0a4b972342e0193441dbfb66c8b5f7806d3bb34976000000000ffffffff0
100f2052a010000001976a914249604bc668da89a7d2d494b89fba47f529c52f788ac00000000’
‘[
{“txid”:”aaa…”,”vout”:1,”scriptPubKey”:”76a9144a06df74729aef1dce5e4641960da3a439d2460b88ac”},
{“txid”:”bbb…”,”vout”:0,”scriptPubKey”:”76a914f88262828f5e64b454249e4c45ddb6071a2ab0a988ac”}
]’
‘[
“93Fu1spd9rCgBc4RbdkxxGcznA4bnQXM6mebzpYqaFFT2P89Cqi”
]’
```

This step will return a raw hex code that can be broadcasted to the network.

```
{
    “hex” :
“01000000011da9283b4ddf8d89eb996988b89ead56cecdc44041ab38bf787f1206cd90b51e000000006a473044022
00ebea9f630f3ee35fa467ffc234592c79538ecd6eb1c9199eb23c4a16a0485a20220172ecaf6975902584987d295b
8dddf8f46ec32ca19122510e22405ba52d1f13201210256d16d76a49e6c8e2edc1c265d600ec1a64a45153d45c29a2
fd0228c24c3a524ffffffff01405dc600000000001976a9140dfc8bafc8419853b34d5e072ad37d1a5159f58488ac0
0000000”,
    “complete” : true
```

```
}
```

iii. Send raw transaction
Use sendrawtransaction RPC to broadcast the hex string obtained above to the network.

```
Bitcoind sendrawtransaction
01000000011da9283b4ddf8d89eb996988b89ead56cecdc44041ab38bf787f1206cd90b51e000000006a4730440220
0ebea9f630f3ee35fa467ffc234592c79538ecd6eb1c9199eb23c4a16a0485a20220172ecaf6975902584987d295b8
dddf8f46ec32ca19122510e22405ba52d1f13201210256d16d76a49e6c8e2edc1c265d600ec1a64a45153d45c29a2f
d0228c24c3a524ffffffff01405dc600000000001976a9140dfc8bafc8419853b34d5e072ad37d1a5159f58488ac00
000000
```

This step will return a transaction id for this transaction.

```
F5a5ce5988cc72b9b90e8d1d6c910cda53c88d2175177357cc2f2cf0899fbaad
```

7. Receive funds from Cold Storage (transaction with nLockTime, sequence number fields set):
   a) eWallet system creates a transaction Tx2 (the contract) that spends Tx1 back to BTC address determined for Alice. This transaction will have the nLockTime field set with value equal to a Unix Epoch timestamp (seconds since 1 Jan 1970) value [3.6]. The sequence number field will be set to 0. Following is the sequence of steps to achieve this:

      i. Create raw transaction to spend Tx1

      Input:

      txid = transaction id of Tx1
      vout = 0

      Output:
      address = Alice’s BTC address
      amount = 9.8 (total BTC that were locked in Step 1)

```
bitcoind createrawtransaction ‘[{“txid”:”
f5a5ce5988cc72b9b90e8d1d6c910cda53c88d2175177357cc2f2cf0899fbaad”,”vout”:0}]’
‘{“mirQLRn6ciqa3WwJSSe7RSJNVfAE9zLkS5”:9.8}’


0100000001bfd27404f34449a53c6a5d0a4b972342e0193441dbfb66c8b5f7806d3bb34976000000000ffffffff01
00f2052a010000001976a914249604bc668da89a7d2d494b89fba47f529c52f788ac00000000
```

Above raw transaction has default sequence number (UINT_MAX) and lock time as 0.

System needs to set lock time to 20 Nov 2015. Also, sequence number needs to be set less than UNIT_MAX.

      ii. Modify the raw transaction above to set: sequence number = 0, lock time = equivalent Unix Epoch timestamp value

Dissecting the raw transaction, we get:

| | |
|---|---|
| 01000000 | version |
| 01 | input count |
| bfd27404f34449a53c6a5d0a4b972342e0193441dbfb66c8b5f7806d3bb34976 | previous output hash |
| 00000000 | previous output index |
| 00 | script length |
| ffffffff | sequence number |
| 01 | output count |
| 00f2052a01000000 | output value |
| 19 | script length |
| 76a914249604bc668da89a7d2d494b89fba47f529c52f788ac | scriptPubKey |
| 00000000 | locktime |

**Setting the sequence number to 0:**

Identify the bytes corresponding to sequence number & set all 4 bytes to 0.

```
0100000001bfd27404f34449a53c6a5d0a4b972342e0193441dbfb66c8b5f7806d3bb3497600000000000ffffffff0100f2
052a010000001976a914249604bc668da89a7d2d494b89fba47f529c52f788ac00000000


0100000001bfd27404f34449a53c6a5d0a4b972342e0193441dbfb66c8b5f7806d3bb34976000000000000000000000100f2
052a010000001976a914249604bc668da89a7d2d494b89fba47f529c52f788ac00000000
```

**Setting the lock time to 20 Nov 2015:**

*(For the sake of this example, an online time converter tool was used to convert datetime into Unix Epoch Timestamp. Depending on the programming language used, respective time functions can be called for this conversion.*

*For example, for C we can use mktime(), for Java we can use java.time package.)*

Convert this date to Unix Epoch Time: 1447981200

Value in Hex: 564E7090

Replace the last 4 bytes of raw transaction with this new value.

```
0100000001bfd27404f34449a53c6a5d0a4b972342e0193441dbfb66c8b5f7806d3bb34976000000000000000000000100f2
052a010000001976a914249604bc668da89a7d2d494b89fba47f529c52f788ac00000000


0100000001bfd27404f34449a53c6a5d0a4b972342e0193441dbfb66c8b5f7806d3bb34976000000000000000000000100f2
052a010000001976a914249604bc668da89a7d2d494b89fba47f529c52f788ac564E7090
```

New raw transaction (Tx2) with updated sequence number and lock time:

```
0100000001bfd27404f34449a53c6a5d0a4b972342e0193441dbfb66c8b5f7806d3bb34976000000000000000000000100f2
052a010000001976a914249604bc668da89a7d2d494b89fba47f529c52f788ac55932D80
```

iii. System signs Tx2 with its own private key.

The raw transaction hex code obtained as output from above step will be signed with Alice's private key.

iDaemon will probably be using its own implementation for signing transactions [6.3.2].

For this example, consider the signrawtransaction RPC from FOS Daemon.

Note that Alice's private key will be generated from the mnemonic seed according to Type 1 Deterministic approach [6.3.1]. Assume the system's private key to be "10De1spd9rCgBc4RbdkxxGcznA4bnQXM6mebzpYqaFFT2P89Cqi"

```
bitcoind signrawtransaction
0100000001bfd27404f34449a53c6a5d0a4b972342e0193441dbfb66c8b5f7806d3bb34976000000000000000000000100f2
052a010000001976a914249604bc668da89a7d2d494b89fba47f529c52f788ac55932D80

'[

{"txid":"
f5a5ce5988cc72b9b90e8d1d6c910cda53c88d2175177357cc2f2cf0899fbaad","vout":0,"scriptPubKey":"76a9144
a06df74729aef1dce5e4641960da3a439d2460b88ac"}

]'

'[

"10De1spd9rCgBc4RbdkxxGcznA4bnQXM6mebzpYqaFFT2P89Cqi"

  ]'
```

This step will return a raw hex code that can be broadcasted to the network.


```
{

    "hex" :
"0100000001bfd27404f34449a53c6a5d0a4b972342e0193441dbfb66c8b5f7806d3bb3497600000000db0048304502210
0bb9ef133361524477c4811b73f7b5093108f646d260dfdd066ea3a06589cf47f02206b91c5bfb091784b2dc62a71477d5
e73a53c3019b6e0b61a4888f24c991e930a014830450221008447 0f4972aab95892e6871168fa0d8456a7e4f55cfc8786a
5ffef289d9d312602206d048d4fa39fd987235ad025c0e2d30ff4d6e7ab60ed5fb899952a3ef888cbf4014752210287f91
69e265380a87cfd717ec543683f572db8b5a6d06231ff59c43429063ae4210343947d178f20b8267488e488442650c27e1
e9956c824077f646d6ce13a285d8452aeffffffff0100f2052a010000001976a914249604bc668da89a7d2d494b89fba47
f529c52f788ac00000000",

    "complete" : true

}
```


b) System permanently deletes the private key used to sign this transaction. This ensures that no one has access to funds in cold storage.
c) The system broadcasts this transaction when the nLockTime time is reached.

```
Bitcoind sendrawtransaction
0100000001bfd27404f34449a53c6a5d0a4b972342e0193441dbfb66c8b5f7806d3bb3497600000000db00483045022100
bb9ef133361524477c4811b73f7b5093108f646d260dfdd066ea3a06589cf47f02206b91c5bfb091784b2dc62a71477d5e
```

```
73a53c3019b6e0b61a4888f24c991e930a0148304502210084470f4972aab95892e6871168fa0d8456a7e4f55cfc8786a5
ffef289d9d312602206d048d4fa39fd987235ad025c0e2d30ff4d6e7ab60ed5fb899952a3ef888cbf4014752210287f916
9e265380a87cfd717ec543683f572db8b5a6d06231ff59c43429063ae4210343947d178f20b8267488e488442650c27e1e
9956c824077f646d6ce13a285d8452aeffffffff0100f2052a010000001976a914249604bc668da89a7d2d494b89fba47f
529c52f788ac00000000

35cdf0594ef0890703a8ede92f6fc80272d0b0b73d19d2a9af80dd17c11e188c
```

***Note:*** *The system needs to monitor when nLockTime will be reached and should broadcast the transaction at the correct time. Broadcasting the transaction earlier than nLockTime might result in dropping of the transaction.*

## 8. of 2 escrow

This escrow transaction doesn't require any 3rd party and utilizes multisig feature of bitcoin. It involves 2 users. This is generally an agreement between 2 parties regarding payment of bitcoins that kicks off ***sometime in future***, depending on ***certain conditions*** being fulfilled.

- The 'future time' part can be implemented by a field called 'locktime' in a bitcoin raw transaction.
- The 'condition' part is a bit tricky. It generally involves consensus of the 2 parties involved and involves some human intervention.

Basic steps involved can be summarized with the following example:

Bob wants to lend Alice 10 BTC but wants to make sure that Alice does not cheat him. Both, Alice & Bob agree that Bob will get the amount back after 1 July 2016. Also, they want the contract to be flexible so that they can change certain aspects (ex. Withdrawing early, extending the 1 July 2016 date etc.)

1. Each party shares their respective public key with each other.
   (Use validateaddress <bitcoinaddress>: Return information about <bitcoinaddress>.)
   *validateaddress RPC shows following information **ONLY** when the bitcoin address belongs to the user and is created using the standard bitcoin client.*

```
Bitcoind validateaddress mpzXCDpitVhGe1WofQXjzC1zgxGA5GCfgD

{

    "isvalid" : true,

    "address" : "mpzXCDpitVhGe1WofQXjzC1zgxGA5GCfgD",

    "ismine" : true,

    "isscript" : false,

    "pubkey" :

    "0287f9169e265380a87cfd717ec543683f572db8b5a6d06231ff59c43429063ae4",

    "iscompressed" : true,

    "account" : ""

}
```

2. Bob creates a P2SH address that requires both parties to sign.
   a) Get public key from Alice (Step 1)

b) Create multisig address that requires both Alice’s & Bob’s signatures.

```
bitcoind addmultisigaddress 2
‘[“0287f9169e265380a87cfd717ec543683f572db8b5a6d06231ff59c43429063ae4”,”0343947d178f20b8267488e4884426
50c27e1e9956c824077f646d6ce13a285d84”]’



3MxKEf2su6FGAUfCEAHreGFQvEYrfYNHvL7
```

3. Bob creates transaction Tx1 by putting 10 BTC into the multisig address

```
bitcoind sendtoaddress 3MxKEf2su6FGAUfCEAHreGFQvEYrfYNHvL7 10.0



7649b33b6d80f7b5c866fbdb413419e04223974b0a5d6a3ca54944f30474d2bf
```

4. Bob communicates the transaction id of Tx1 to Alice

5. Alice can see the P2SH transaction from transaction id provided by Bob

```
bitcoind getrawtransaction 7649b33b6d80f7b5c866fbdb413419e04223974b0a5d6a3ca54944f30474d2bf 1
```


```
{

“hex” :
“01000000013c0c37049cefb7d0754c716c1227e221f1b5cc9fdf7fc8e6aadd5ce6465fad32000000004a493046022100b4133
0548f320fcc282d72462656f80c0da64beb352f7fbbdf55d651674b5846022100cbef624c80302900e6c0e9b4bbb024cd072e5
4d7535c8a79a3ce9b36c304d7cc01ffffffff0100f2052a0100000017a914379ad9b7ba73bdc1e29e286e014d4e2e1f6884e38
700000000”,

“txid” : “7649b33b6d80f7b5c866fbdb413419e04223974b0a5d6a3ca54944f30474d2bf”,

“version” : 1,

“locktime” : 0,

“vin” : [

{

“txid” : “32ad5f46e65cddaae6c87fdf9fccb5f121e227126c714c75d0b7ef9c04370c3c”,

“vout” : 0,

“scriptSig” : {

“asm” :
“3046022100b41330548f320fcc282d72462656f80c0da64beb352f7fbbdf55d651674b5846022100cbef624c80302900e6c0e
9b4bbb024cd072e54d7535c8a79a3ce9b36c304d7cc01”,

“hex” :
“493046022100b41330548f320fcc282d72462656f80c0da64beb352f7fbbdf55d651674b5846022100cbef624c80302900e6c
0e9b4bbb024cd072e54d7535c8a79a3ce9b36c304d7cc01”

},
```

```
"sequence" : 4294967295
}
],
"vout" : [
{
"value" : 10.00000000,
"n" : 0,
"scriptPubKey" : {
"asm" : "OP_HASH160 379ad9b7ba73bdc1e29e286e014d4e2e1f6884e3 OP_EQUAL",
"hex" : "a914379ad9b7ba73bdc1e29e286e014d4e2e1f6884e387",
"reqSigs" : 1,
"type" : "scripthash",
"addresses" : [
"2MxKEf2su6FGAUfCEAHreGFQvEYrfYNHvL7"
]
}
}
]
}
```


Alice is now convinced that Bob has paid 10 BTC in Tx1.

6. Alice creates a transaction Tx2 (the contract). Tx2 spends Tx1 and pays it back to Bob via the address he provided in the first step.
   a) Create raw transaction to spend Tx1

      i.e. Alice creates a transaction that has following input & output:

      Input: vout 0 from Tx1

      Output: Bob's BTC address, 10 BTC

```
bitcoind createrawtransaction
'[{"txid":"7649b33b6d80f7b5c866fbdb413419e04223974b0a5d6a3ca54944f30474d2bf","vout":0}]'
'{"mirQLRn6ciqa3WwJSSe7RSJNVfAE9zLkS5":10}'


0100000001bfd27404f34449a53c6a5d0a4b972342e0193441dbfb66c8b5f7806d3bb349760000000000ffffffff01
00f2052a010000001976a914249604bc668da89a7d2d494b89fba47f529c52f788ac00000000
```

Above raw transaction has default sequence number (UINT_MAX) and lock time as 0.

Alice needs to set lock time to some future date (after 1 July 2016). Hence, sequence number needs to be set less than UNIT_MAX.

b) Modify the raw transaction above to set: sequence number = 0, lock time = 1 July 2016

Dissecting the raw transaction, we get:

| | |
|---|---|
| 01000000 | version |
| 01 | input count |
| bfd27404f34449a53c6a5d0a4b972342e0193441dbfb66c8b5f7806d3bb34976 | previous output hash |
| 00000000 | previous output index |
| 00 | script length |
| ffffffff | sequence number |
| 01 | output count |
| 00f2052a01000000 | output value |
| 19 | script length |
| 76a914249604bc668da89a7d2d494b89fba47f529c52f788ac | scriptPubKey |
| 00000000 | locktime |

**Setting the sequence number to 0:**

Identify the bytes corresponding to sequence number & set all 4 bytes to 0.

```
0100000001bfd27404f34449a53c6a5d0a4b972342e0193441dbfb66c8b5f7806d3bb3497600000000000ffffffff0100f2
052a010000001976a914249604bc668da89a7d2d494b89fba47f529c52f788ac00000000


0100000001bfd27404f34449a53c6a5d0a4b972342e0193441dbfb66c8b5f7806d3bb3497600000000000000000000100f2
052a010000001976a914249604bc668da89a7d2d494b89fba47f529c52f788ac00000000
```

**Setting the lock time to 1 July 2016:**

Convert this date to Unix Epoch Time: 1435708800

Value in Hex: 55932D80

Replace the last 4 bytes of raw transaction with this new value.

```
0100000001bfd27404f34449a53c6a5d0a4b972342e0193441dbfb66c8b5f7806d3bb3497600000000000000000000100f2
052a010000001976a914249604bc668da89a7d2d494b89fba47f529c52f788ac00000000


0100000001bfd27404f34449a53c6a5d0a4b972342e0193441dbfb66c8b5f7806d3bb3497600000000000000000000100f2
052a010000001976a914249604bc668da89a7d2d494b89fba47f529c52f788ac55932D80
```

New raw transaction (Tx2) with updated sequence number and lock time:

```
0100000001bfd27404f34449a53c6a5d0a4b972342e0193441dbfb66c8b5f7806d3bb349760000000000000000000100f2
052a010000001976a914249604bc668da89a7d2d494b89fba47f529c52f788ac55932D80
```

7. Alice signs Tx2

```
bitcoind signrawtransaction
0100000001bfd27404f34449a53c6a5d0a4b972342e0193441dbfb66c8b5f7806d3bb349760000000000000000000100f2
052a010000001976a914249604bc668da89a7d2d494b89fba47f529c52f788ac55932D80

{

    "hex" :
"0100000001bfd27404f34449a53c6a5d0a4b972342e0193441dbfb66c8b5f7806d3bb3497600000000920048304502210
084470f4972aab95892e6871168fa0d8456a7e4f55cfc8786a5ffef289d9d312602206d048d4fa39fd987235ad025c0e2d
30ff4d6e7ab60ed5fb899952a3ef888cbf4014752210287f9169e265380a87cfd717ec543683f572db8b5a6d06231ff59c
43429063ae4210343947d178f20b8267488e488442650c27e1e9956c824077f646d6ce13a285d8452aeffffffff0100f20
52a010000001976a914249604bc668da89a7d2d494b89fba47f529c52f788ac00000000",

    "complete" : false

}
```

8. Finally, the incomplete (half-signed) transaction is sent back to Bob. Bob checks that the contract is as expected – that the coins will eventually come back to him – but, unless things are changed, only after 1 July 2016. Because the sequence number is zero, the contract can be amended in future if both parties agree. The script in the input isn't finished though; there are only zeros where the user's signature should be. He fixes that by signing the contract and putting the new signature in the appropriate spot.

```
Bitcoind signrawtransaction
0100000001bfd27404f34449a53c6a5d0a4b972342e0193441dbfb66c8b5f7806d3bb3497600000000920048304502210084470f4972aab95892e6871168fa0d8456a7e4f55cfc8786a5ffef289d9d312602206d048d4fa39fd987235ad025c0e2d30ff4d6e7
ab60ed5fb899952a3ef888cbf4014752210287f9169e265380a87cfd717ec543683f572db8b5a6d06231ff59c43429063ae421
0343947d178f20b8267488e488442650c27e1e9956c824077f646d6ce13a285d8452aeffffffff0100f2052a010000001976a9
14249604bc668da89a7d2d494b89fba47f529c52f788ac00000000

{

    "hex" :
"0100000001bfd27404f34449a53c6a5d0a4b972342e0193441dbfb66c8b5f7806d3bb3497600000000db00483045022100bb9
ef133361524477c4811b73f7b5093108f646d260dfdd066ea3a06589cf47f02206b91c5bfb091784b2dc62a71477d5e73a53c3
019b6e0b61a4888f24c991e930a01483045022100084470f4972aab95892e6871168fa0d8456a7e4f55cfc8786a5ffef289d9d3
12602206d048d4fa39fd987235ad025c0e2d30ff4d6e7ab60ed5fb899952a3ef888cbf4014752210287f9169e265380a87cfd7
17ec543683f572db8b5a6d06231ff59c43429063ae4210343947d178f20b8267488e488442650c27e1e9956c824077f646d6ce
13a285d8452aeffffffff0100f2052a010000001976a914249604bc668da89a7d2d494b89fba47f529c52f788ac00000000",

    "complete" : true

}
```

9. Bob broadcasts Tx1. Then Tx2

```
bitcoind sendrawtransaction
0100000001bfd27404f34449a53c6a5d0a4b972342e0193441dbfb66c8b5f7806d3bb3497600000000db00483045022100bb9e
f133361524477c4811b73f7b5093108f646d260dfdd066ea3a06589cf47f02206b91c5bfb091784b2dc62a71477d5e73a53c30
19b6e0b61a4888f24c991e930a01483045022100084470f4972aab95892e6871168fa0d8456a7e4f55cfc8786a5ffef289d9d31
```

```
2602206d048d4fa39fd987235ad025c0e2d30ff4d6e7ab60ed5fb899952a3ef888cbf4014752210287f9169e265380a87cfd71
7ec543683f572db8b5a6d06231ff59c43429063ae4210343947d178f20b8267488e488442650c27e1e9956c824077f646d6ce1
3a285d8452aeffffffff0100f2052a010000001976a914249604bc668da89a7d2d494b89fba47f529c52f788ac00000000



35cdf0594ef0890703a8ede92f6fc80272d0b0b73d19d2a9af80dd17c11e188c
```

At this stage, neither Bob nor Alice can spend the 10 BTC independently. After 1 July 2016, the contract will be complete & Bob will get the coins back in his address.

## 9.7 Method to Create an IP Transaction

This point can be done in the scope of future development. Will need some researching activity.

## 9.8 Method to Create a Message Transaction

Message Trx is transaction which is used for sending message via Block Chain. Message can be embedded into Block Chain by using OP_RETURN opcode [2.2]. OP_RETURN outputs are specifically designed to allow you to embed **40 bytes** (320 bit) in a transaction.

Sequence of Steps: First, identify the input that you want to use for this purpose. Remember that all of the amount will go to the miner. Then, identify the scriptPubKey from the raw transaction and replace it to include OP_RETURN & the desired message/metadata.

***Note****: Calculating the length of each field: The length in raw transaction is denoted in hex.*

While creating your own transaction, you need to convert the length in chars to length in bytes. Convert this no. to hex by using decimal to hex converter (1 byte = 2 chars). Count the no. of characters in the fields. Divide by 2. That will give you the no. of bytes. Convert this no. to hex.

**Example:** The scriptPubKey field has the value "76a91401720d2372616d6176fc16cac19378bdcb74b36e88ac"

No. of characters in Decimal: 50

No. of bytes in Decimal: 25 = $1\times16^1+9\times16^0$ (from hex)

Value in hex: 0x19

**Steps:**

**1. Identify the input that needs to be converted to OP_RETURN output.**

Use Message Transaction Management SubSystem (MTrxMSS) to get a list of UTXOs. Identify the ones that can be used to generate the required output.

Output should be equivalent to the "listunspent" RPC output from FOS Daemon.

"listunspent" command  get list of transactions that are unspent & can be used to create new transaction.

Example output from "listunspent" command:

```
[{
    "txid": "99fa789df2a0aef57e705f66f3185f30ba71e544b246661c74c9f6ec22a86546",
    "vout": 1,
    "address": "15VS9KdnP4Zna1MzF8F2jJ6WS1nKGoACqv",
    "account": "",
    "scriptPubKey": "76a91431412386e7fab5d1a5285fd17a6fb113db781eec88ac",
    "amount": 0.00500000,
```

```
    "confirmations": 691
}]
```

### 2. Create a raw transaction

"createrawtransaction" command (*bitcoind command*): use the txid, vout, amount from above step to create a raw transaction.

Command example: createrawtransaction

```
'[{"txid":"99fa789df2a0aef57e705f66f3185f30ba71e544b246661c74c9f6ec22a86546", "vout":1}]'
'{"18eJcmJDXWigB3Bw6drAmCaz6H69F9Mz5":0.0049}'
```

In this case, "18eJcmJDXWigB3Bw6drAmCaz6H69F9Mz5" is the destination address.

(Destination address will be overwritten in further steps.)

Output:

01000000014665a822ecf6c9741c6646b244e571ba305f18f3665f707ef5aea0f29d78fa990100000000ffffffff
01107a07000000000001976a91401720d2372616d6176fc16cac19378bdcb74b36e88ac00000000

**The table below explains the raw transaction parts:**

| | | | | |
|---|---|---|---|---|
| 1 | version | | 4 bytes | 01000000 |
| 2 | input count | | 1 byte | 01 |
| 3 | input | previous output hash (big endian, reversed) | 32 bytes | 4665a822ecf6c9741c6646b244e571ba305f18f3665f707ef5aea0f29d78fa99 |
| 4 | | previous output index | 4 bytes | 01000000 |
| 5 | | script length | 1 byte | 00 |
| 6 | | scriptSig | | |
| 7 | | sequence | 4 bytes | ffffffff |
| 8 | output count | | 1 byte | 01 |
| 9 | output | value | 8 bytes (64 bit, little endian) | 107a070000000000 |
| 10 | | script length | 1 byte | 19 |
| 11 | | scriptPubKey | | 1976a91401720d2372616d6176fc16cac19378bdcb74b36e88ac |
| 12 | block lock time | | 4 bytes | 00000000 |

<u>Explanation:</u>

1. Version: 4 bytes. Transaction data format version
2. No. of inputs: 1 byte. Number of Transaction inputs (incoming trxs)

3. Hash of the transaction from which we want to redeem (reverse order): 32 bytes. One thing to note is that this value is stored as big endian, so, you'll have to reverse the bytes around (Not the digits), so, what normally would be: - 12345678 gets reversed in bytes, so: - 78 = first byte, 56 = second byte, 34 = third byte, 12 = forth byte. Making 0x78563412 from 0x1234678.
4. Output index we want to redeem from the transaction: 4 bytes
5. Length of the scriptSig: 1 byte.
6. Actual scriptSig (equal to length in previous bytes).
7. Default sequence ffffffff: 4 bytes.
8. No. of outputs (outgoing trxs) in the new transaction: 1 byte.
9. Amount to be redeemed (64 bit integer, little-endian): 8 bytes.
10. Length of the scriptPubKey: 1 byte.
11. Actual script (equal to length in previous bytes)
12. Lock time: 4 bytes – 00000000. Block height or timestamp when transaction is final.

### 3. Identify scriptPubKey

Get the scriptPubKey from the raw transaction:

01000000014665a822ecf6c9741c6646b244e571ba305f18f3665f707ef5aea0f29d78fa990100000000ffffffff 01107a0700000000001976a91401720d2372616d6176fc16cac19378bdcb74b36e88ac00000000

scriptPubKey = 1976a91401720d2372616d6176fc16cac19378bdcb74b36e88ac

We need to convert the above scriptPubKey to suit our aim (i.e. embed OP_RETURN message/metadata)

The first byte (19) means the length of it, in this case, 25 bytes long as 0x19 means 25 in decimal (Not including itself, 76a91401720d2372616d6176fc16cac19378bdcb74b36e88ac -> No. of characters = 50 -> No. of bytes = 25).

Hex = 19

Decimal= $1\times16^1+9\times16^0 = 25$

Binary = 11001

Decimal= $1\times2^4+1\times2^3+0\times2^2+0\times2^1+1\times2^0 = 25$

Removing the length, we get:

76a91401720d2372616d6176fc16cac19378bdcb74b36e88ac

First byte denotes the opcode.

0x76 = OP_DUP. But we want OP_RETURN. So, let's create our own signature.

**4. Create our own scriptPubKey for opcode OP_RETURN**

- Message to embed: "Testing OP_RETURN"
- Converting the message to hex, we get: **54657374696e67204f505f52455455524e**

  So basically, we want: an OP_RETURN (0x6a), then the above hex.

- Hence, format of our scriptPubKey will be:

  [OP_RETURN hex][length of message in hex][message in hex]

  - OP_RETURN hex: 0x**6a**
  - the length of our message in hex: 54657374696e67204f505f52455455524e-> No. of characters = 34 -> No. of bytes = 17 = $1\times16^1+1\times16^0$, so, hex 0x**11**
  - and then the message in hex: **54657374696e67204f505f52455455524e**

  Result: **6a1154657374696e67204f505f52455455524e**

- Now we just update the length to being the actual length (in this case **6a1154657374696e67204f505f52455455524e** -> No. of characters = 38 -> No. of bytes = 19 -> 19 in hex = $1\times16^1+3\times16^0$, so hex 0x**13**):-

  Result: **136a1154657374696e67204f505f52455455524e**

- Add it back to the original transaction:-

  0100000014665a822ecf6c9741c6646b244e571ba305f18f3665f707ef5aea0f29d78fa990100000000ffffffff01107a0700000000000**136a1154657374696e67204f505f52455455524e**00000000

  Above steps will create OP_RETURN metadata and embed the hex data in the Block Chain. The transaction will be unspendable. All the amount from the chosen input transaction will go to the miner.

## 9.9 Ways to Create Open Assets Transactions

This point can be done in the scope of future development. Will need some researching activity.

# 10. Intelligent Daemon System Class and Sequence Diagrams

Under construction…

## 10.1 Single-sig Transaction Management SubSystem Diagrams

Under construction…

## 10.2 Accounting Transaction Management SubSystem Diagrams

This point can be done in the scope of future development. Will need some researching activity.

## 10.3 Bank Transaction Management SubSystem Diagrams

This point can be done in the scope of future development. Will need some researching activity.

## 10.4 Exchange Transaction Management SubSystem Diagrams

This point can be done in the scope of future development. Will need some researching activity.

## 10.5 Message Transaction Management SubSystem Diagrams

This point can be done in the scope of future development. Will need some researching activity.

## 10.6 Contracts Management SubSystem Diagrams

This point can be done in the scope of future development. Will need some researching activity.

## 10.7 Monitoring System Diagrams

Under construction …

## 10.8 Diagrams for Wrapper of DmnCC

Under construction …

## 10.9 Shared Libraries Class and Sequence Diagrams

### 10.9.1 Common Ware API

Under construction …

### 10.9.2 4S API

Under construction …

### 10.9.3 ECDSA API

Under construction …

### 10.9.4 Mnemonic Code Generator API

9 Under construction …

# 11.Integration with External Systems

## 11.1 Interfaces

“Intelligent Daemon System” (IntDS) will provide restful interfaces to integrate with other external systems (eWallet web app., BTC Accounting web app., Trading system, other Btc bank systems, etc.).

REST’s client–server separation of concerns simplifies component implementation, reduces the complexity of connector semantics, improves the effectiveness of performance tuning, and increases the scalability of pure server components. Layered system constraints allow intermediaries—proxies, gateways, and firewalls—to be introduced at various points in the communication without changing the interfaces between components, thus allowing them to assist in communication translation or improve performance via large-scale, shared caching. REST enables intermediate processing by constraining messages to be self-descriptive: interaction is stateless between requests, standard methods and media types are used to indicate semantics and exchange information, and responses explicitly indicate cacheability.

The entry point to IntDS is “Transactions Management System” (STrxMSS) therefore all the API provided by STrxMSS are public. The API provided by Daemon Core System is private and used by STrxMSS and Monitoring System only. The description of public API provided by STrxMSS is described in

Intelligent Daemon System Interfaces”

Under construction…

## 11.2 DBs Mapping Recommendations

Under construction…

# Appendix A – Transaction Statuses

| Status ID | Status | Description |
|---|---|---|
| 1 | Confirmed | Transaction was confirmed in the Block Chain. |
| 2 | Pending | Transaction was issued into Block Chain and is awaiting confirmation |
| 3 | In Progress | Transaction was injected into iDaemon system but was not issued into Block Chain |
| 4 | Cancelled | Transaction was cancelled in the iDaemon system. |
| 5 | Rejected | Transaction was rejected in the Block Chain. |
| 6 | Unknown | Status is not recognized by iDaemon system |

# Appendix B – Transaction Types

The value of the column “Type of Script Pairs” is given from Appendix E, column “Type Title”.

| Type | Type of Script Pairs | Description |
|---|---|---|
| Contract | | Transactions which use the decentralized Bitcoin system to enforce financial agreements [2.7].<br>A distributed contract is a method of using Bitcoin to form agreements with people via the block chain. |
| Financial Single-sig Trxs | Standard Transaction to Single-sig Bitcoin address (P2PKH) | Sending some Bitcoins from the Single-sig Btc address to the Single-sig Btc address. All Inputs and Outputs of this Trx should correspond to P2PKH type |
| Financial Multi-sig Trxs | M-of-N Multi-signature Transaction (P2SH) | Complex/Multi-signature transaction is a transaction that has as one of its Inputs a Multi-sig Btc address. Sending some Bitcoins from the Multi-sig Btc Address.<br>Multi-sig addresses are used to make it so multiple keys owned by separate entities are needed to move the bitcoins in an address. |
| IP Trxs | P2PK | Sending bitcoins to an IP address. Sending directly to in PubKey [2.10]. |
| Message Trxs | Provably Unspendable/Prunable Outputs | Transactions which are used for sending some metadata or message |
| Open Assets Trxs | | Open Assets transactions can be used to issue new assets, or transfer ownership of assets [2.6]. |
| Strange | -- | Any Unusual transactions |

# Appendix C – Opcode types

| Type | Description |
|---|---|
| Constants | When talking about scripts, these value-pushing words are usually omitted. |
| Flow control | Conditional flow control opcodes. |
| Stack | Opcodes used to manipulate the stack. |
| Splice | Opcodes used for string splice operations. |
| Bitwise logic | Opcodes used for binary arithmetic and boolean logical operations. |
| Arithmetic | ***Note:*** *Arithmetic inputs are limited to signed 32-bit integers, but may overflow their output.*<br>If any input value for any of these commands is longer than 4 bytes, the script must abort and fail. If any opcode marked as disabled is present in a script – it must also abort and fail. |
| Crypto | Cryptographic and hashing opcodes. |
| Pseudo-words | These words are used internally for assisting with transaction matching. They are invalid if used in actual scripts. |
| Reserved words | Any opcode not assigned is also reserved. Using an unassigned opcode makes the transaction **invalid**. |

# Appendix D – Opcodes [2.2]

| Word | Opcode | Hex | Input | Output | Description |
|---|---|---|---|---|---|
| **Constants:** | | | | | |
| OP_0, OP_FALSE | 0 | 0x00 | Nothing. | (empty value) | An empty array of bytes is pushed onto the stack. (This is not a no-op: an item is added to the stack.) |
| N/A | 1-75 | 0x01-0x4b | (special) | data | The next opcode bytes is data to be pushed onto the stack |
| OP_PUSHDATA1 | 76 | 0x4c | (special) | data | The next byte contains the number of bytes to be pushed onto the stack. |
| OP_PUSHDATA2 | 77 | 0x4d | (special) | data | The next two bytes contain the number of bytes to be pushed onto the stack. |
| OP_PUSHDATA4 | 78 | 0x4e | (special) | data | The next four bytes contain the number of bytes to be pushed onto the stack. |
| OP_1NEGATE | 79 | 0x4f | Nothing. | -1 | The number -1 is pushed onto the stack. |
| OP_1, OP_TRUE | 81 | 0x51 | Nothing. | 1 | The number 1 is pushed onto the stack. |
| OP_2-OP_16 | 82-96 | 0x52-0x60 | Nothing. | 2-16 | The number in the word name (2-16) is pushed onto the stack. |
| **Flow control:** | | | | | |
| OP_NOP | 97 | 0x61 | Nothing | Nothing | Does nothing. |
| OP_IF | 99 | 0x63 | <expression> if [statements] [else [statements]]* endif | | If the top stack value is not 0, the statements are executed. The top stack value is removed. |

| | | | | | |
|---|---|---|---|---|---|
| OP_NOTIF | 100 | 0x64 | <expression> if [statements] [else [statements]]* endif | | If the top stack value is 0, the statements are executed. The top stack value is removed. |
| OP_ELSE | 103 | 0x67 | <expression> if [statements] [else [statements]]* endif | | If the preceding OP_IF or OP_NOTIF or OP_ELSE was not executed then these statements are and if the preceding OP_IF or OP_NOTIF or OP_ELSE was executed then these statements are not. |
| OP_ENDIF | 104 | 0x68 | <expression> if [statements] [else [statements]]* endif | | Ends an if/else block. All blocks must end, or the transaction is invalid. An OP_ENDIF without OP_IF earlier is also invalid. |
| OP_VERIFY | 105 | 0x69 | True / false | Nothing / False | Marks transaction as invalid if top stack value is not true. |
| OP_RETURN | 106 | 0x6a | Nothing | Nothing | Marks transaction as invalid. |
| **Stack:** | | | | | |
| OP_TOALTSTACK | 701 | 0x6b | x1 | (alt)x1 | Puts the input onto the top of the alt stack. Removes it from the main stack. |
| OP_FROMALTSTACK | 108 | 0x6c | (alt)x1 | x1 | Puts the input onto the top of the main stack. Removes it from the alt stack. |
| OP_IFDUP | 115 | 0x73 | x | x / x x | If the top stack value is not 0, duplicate it. |
| OP_DEPTH | 116 | 0x74 | Nothing | <Stack size> | Puts the number of stack items onto the stack. |
| OP_DROP | 117 | 0x75 | x | Nothing | Removes the top stack item. |
| OP_DUP | 118 | 0x76 | x | x x | Duplicates the top stack item. |
| OP_NIP | 119 | 0x77 | x1 x2 | x2 | Removes the second-to-top stack item. |

| | | | | | |
|---|---|---|---|---|---|
| OP_OVER | 120 | 0x78 | x1 x2 | x1 x2 x1 | Copies the second-to-top stack item to the top. |
| OP_PICK | 121 | 0x79 | xn … x2 x1 x0 <n> | xn … x2 x1 x0 xn | The item n back in the stack is copied to the top. |
| OP_ROLL | 122 | 0x7a | xn … x2 x1 x0 <n> | … x2 x1 x0 xn | The item n back in the stack is moved to the top. |
| OP_ROT | 123 | 0x7b | x1 x2 x3 | x2 x3 x1 | The top three items on the stack are rotated to the left. |
| OP_SWAP | 124 | 0x7c | x1 x2 | x2 x1 | The top two items on the stack are swapped. |
| OP_TUCK | 125 | 0x7d | x1 x2 | x2 x1 x2 | The item at the top of the stack is copied and inserted before the second-to-top item. |
| OP_2DROP | 109 | 0x6d | x1 x2 | Nothing | Removes the top two stack items. |
| OP_2DUP | 110 | 0x6e | x1 x2 | x1 x2 x1 x2 | Duplicates the top two stack items. |
| OP_3DUP | 111 | 0x6f | x1 x2 x3 | x1 x2 x3 x1 x2 x3 | Duplicates the top three stack items. |
| OP_2OVER | 112 | 0x70 | x1 x2 x3 x4 | x1 x2 x3 x4 x1 x2 | Copies the pair of items two spaces back in the stack to the front. |
| OP_2ROT | 113 | 0x71 | x1 x2 x3 x4 x5 x6 | x3 x4 x5 x6 x1 x2 | The fifth and sixth items back are moved to the top of the stack. |
| OP_2SWAP | 114 | 0x72 | x1 x2 x3 x4 | x3 x4 x1 x2 | Swaps the top two pairs of items. |
| **Splice:** | | | | | |

| | | | | | |
|---|---|---|---|---|---|
| OP_CAT | 126 | 0x7e | x1 x2 | out | Concatenates two strings. Disabled. |
| OP_SUBSTR | 127 | 0x7f | in begin size | out | Returns a section of a string. Disabled. |
| OP_LEFT | 128 | 0x80 | in size | out | Keeps only characters left of the specified point in a string. Disabled. |
| OP_RIGHT | 129 | 0x81 | in size | out | Keeps only characters right of the specified point in a string. Disabled. |
| OP_SIZE | 130 | 0x82 | in | in size | Pushes the string length of the top element of the stack (without popping it). |
| **Bitwise logic:** | | | | | |
| OP_INVERT | 131 | 0x83 | in | out | Flips all of the bits in the input. Disabled. |
| OP_AND | 132 | 0x84 | x1 x2 | out | Boolean and between each bit in the inputs. Disabled. |
| OP_OR | 133 | 0x85 | x1 x2 | out | Boolean or between each bit in the inputs. Disabled. |
| OP_XOR | 134 | 0x86 | x1 x2 | out | Boolean exclusive or between each bit in the inputs. Disabled. |
| OP_EQUAL | 135 | 0x87 | x1 x2 | True / false | Returns 1 if the inputs are exactly equal, 0 otherwise. |
| OP_EQUALVERIFY | 136 | 0x88 | x1 x2 | True / false | Same as OP_EQUAL, but runs OP_VERIFY afterward. |
| **Arithmetic:** | | | | | |
| OP_1ADD | 139 | 0x8b | in | out | 1 is added to the input. |
| OP_1SUB | 140 | 0x8c | in | out | 1 is subtracted from the input. |

| | | | | | |
|---|---|---|---|---|---|
| OP_2MUL | 141 | 0x8d | in | out | The input is multiplied by 2. Disabled. |
| OP_2DIV | 142 | 0x8e | in | out | The input is divided by 2. Disabled. |
| OP_NEGATE | 143 | 0x8f | in | out | The sign of the input is flipped. |
| OP_ABS | 144 | 0x90 | in | out | The input is made positive. |
| OP_NOT | 145 | 0x91 | in | out | If the input is 0 or 1, it is flipped. Otherwise the output will be 0. |
| OP_0NOTEQUAL | 146 | 0x92 | in | out | Returns 0 if the input is 0. 1 otherwise. |
| OP_ADD | 147 | 0x93 | a b | out | a is added to b. |
| OP_SUB | 148 | 0x94 | a b | out | b is subtracted from a. |
| OP_MUL | 149 | 0x95 | a b | out | a is multiplied by b. disabled. |
| OP_DIV | 150 | 0x96 | a b | out | a is divided by b. disabled. |
| OP_MOD | 151 | 0x97 | a b | out | Returns the remainder after dividing a by b. disabled. |
| OP_LSHIFT | 152 | 0x98 | a b | out | Shifts a left b bits, preserving sign. Disabled. |
| OP_RSHIFT | 153 | 0x99 | a b | out | Shifts a right b bits, preserving sign. Disabled. |
| OP_BOOLAND | 154 | 0x9a | a b | out | If both a and b are not 0, the output is 1. Otherwise 0. |
| OP_BOOLOR | 155 | 0x9b | a b | out | If a or b is not 0, the output is 1. Otherwise 0. |
| OP_NUMEQUAL | 156 | 0x9c | a b | out | Returns 1 if the numbers are equal, 0 otherwise. |
| OP_NUMEQUALVERIFY | 157 | 0x9d | a b | out | Same as OP_NUMEQUAL, but runs OP_VERIFY afterward. |

| | | | | | |
|---|---|---|---|---|---|
| OP_NUMNOTEQUAL | 158 | 0x9e | a b | out | Returns 1 if the numbers are not equal, 0 otherwise. |
| OP_LESSTHAN | 159 | 0x9f | a b | out | Returns 1 if a is less than b, 0 otherwise. |
| OP_GREATERTHAN | 160 | 0xa0 | a b | out | Returns 1 if a is greater than b, 0 otherwise. |
| OP_LESSTHANOREQUAL | 161 | 0xa1 | a b | out | Returns 1 if a is less than or equal to b, 0 otherwise. |
| OP_GREATERTHANOREQUAL | 162 | 0xa2 | a b | out | Returns 1 if a is greater than or equal to b, 0 otherwise. |
| OP_MIN | 163 | 0xa3 | a b | out | Returns the smaller of a and b. |
| OP_MAX | 164 | 0xa4 | a b | out | Returns the larger of a and b. |
| OP_WITHIN | 165 | 0xa5 | x min max | out | Returns 1 if x is within the specified range (left-inclusive), 0 otherwise. |
| **Crypto:** | | | | | |
| OP_RIPEMD160 | 166 | 0xa6 | in | hash | The input is hashed using RIPEMD-160. |
| OP_SHA1 | 167 | 0xa7 | in | hash | The input is hashed using SHA-1. |
| OP_SHA256 | 168 | 0xa8 | in | hash | The input is hashed using SHA-256. |
| OP_HASH160 | 169 | 0xa9 | in | hash | The input is hashed twice: first with SHA-256 and then with RIPEMD-160. |
| OP_HASH256 | 170 | 0xaa | in | hash | The input is hashed two times with SHA-256. |
| OP_CODESEPARATOR | 171 | 0xab | Nothing | Nothing | All of the signature checking words will only match |

| | | | | | |
|---|---|---|---|---|---|
| | | | | | signatures to the data after the most recently-executed OP_CODESEPARATOR. |
| OP_CHECKSIG | 172 | 0xac | sig pubkey | True / false | The entire transaction's outputs, inputs, and script (from the most recently-executed OP_CODESEPARATOR to the end) are hashed. The signature used by OP_CHECKSIG must be a valid signature for this hash and public key. If it is, 1 is returned, 0 otherwise. |
| OP_CHECKSIGVERIFY | 173 | 0xad | sig pubkey | True / false | Same as OP_CHECKSIG, but OP_VERIFY is executed afterward. |
| OP_CHECKMULTISIG | 174 | 0xae | x sig1 sig2 ... <number of signatures> pub1 pub2 <number of public keys> | True / False | Compares the first signature against each public key until it finds an ECDSA match. Starting with the subsequent public key, it compares the second signature against each remaining public key until it finds an ECDSA match. The process is repeated until all signatures have been checked or not enough public keys remain to produce a successful result. All signatures need to match a public key. Because public keys are not checked again if they fail any signature comparison, signatures must be placed in the scriptSig using the same order as their corresponding public keys were placed in the scriptPubKey or redeemScript. If all signatures are valid, 1 is returned, 0 otherwise. Due to a bug, one |

| | | | | | |
|---|---|---|---|---|---|
| | | | | | extra unused value is removed from the stack. |
| OP_CHECKMULTISIGVERIFY | 175 | 0xaf | x sig1 sig2 ... <number of signatures> pub1 pub2 ... <number of public keys> | True / False | Same as OP_CHECKMULTISIG, but OP_VERIFY is executed afterward. |
| **Pseudo-words:** | | | | | |
| OP_PUBKEYHASH | 253 | 0xfd | -- | -- | Represents a public key hashed with OP_HASH160. |
| OP_PUBKEY | 254 | 0xfe | -- | -- | Represents a public key compatible with OP_CHECKSIG. |
| OP_INVALIDOPCODE | 255 | 0xff | -- | -- | Matches any opcode that is not yet assigned. |
| **Reserved words:** | | | | | |
| OP_RESERVED | 80 | 0x50 | -- | -- | When used... Transaction is invalid unless 324ubscript in an unexecuted OP_IF branch |
| OP_VER | 98 | 0x62 | -- | -- | When used... Transaction is invalid unless 324ubscript in an unexecuted OP_IF branch |
| OP_VERIF | 101 | 0x65 | -- | -- | When used... Transaction is invalid even when 324ubscript in an unexecuted OP_IF branch |
| OP_VERNOTIF | 102 | 0x66 | -- | -- | When used... Transaction is invalid even when 324ubscript in an unexecuted OP_IF branch |
| OP_RESERVED1 | 137 | 0x89 | -- | -- | When used... Transaction is invalid unless 324ubscript in an unexecuted OP_IF branch |

| | | | | | |
|---|---|---|---|---|---|
| OP_RESERVED2 | 138 | 0x8a | -- | -- | When used... Transaction is invalid unless 325ubscript in an unexecuted OP_IF branch |
| OP_NOP1-OP_NOP10 | 176-185 | 0xb0-0xb9 | -- | -- | When used... The word is ignored. Does not mark transaction as invalid. |

# Appendix E – Types of Script Pairs

| ID | Script Type Title | Script Type | Output's Script Title | Output's Script formula (send) | Input's Script Title | Input's Script formula (claim) | Description |
|---|---|---|---|---|---|---|---|
| 1 | Pay-to-Public-Key-Hash | P2PKH | scriptPubKey | OP_DUP OP_HASH160 [pubKeyHash] OP_EQUALVERIFY OP_CHECKSIG | scriptSig | [sig][pubKey] | Scripts for Standard Transaction sending money to a Single-sig Bitcoin address and claiming money sent in this way.<br>A Bitcoin address is only a hash, so the sender can't provide a full public key in scriptPubKey. When redeeming coins that have been sent to a Bitcoin address, the recipient provides both the signature and the public key. The script verifies that the provided public key does hash to the hash in scriptPubKey, and then it also checks the signature against the public key. |
| 2 | Pay-to-Public-Key (Obsolete) | P2PK | | [pubKey] OP_CHECKSIG | | [sig] | Now most often seen in coinbase transactions.<br>Standard script assigning newly generated coins to a Bitcoin address and claiming these coins.<br>This is also used for transactions to an IP address. |
| 3 | Data Output (Provably Unsp | OP_RETURN | | OP_RETURN {zero or more ops as metadata, | | --- | OP_RETURN immediately marks the script as invalid, guaranteeing that no scriptSig exists that could possibly spend that output. Thus the output can be immediately pruned |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | endable/ Prunable Outputs) | | | message etc.} | | | from the UTXO set even if it has not been spent. |
| 4 | Pay-to-Script-Hash | P2SH | | OP_HASH160 [hashOfScript] OP_EQUAL<br>***Note:*** *[hashOfScript] is 20-byte-hash-value* | | [signatures as required by script][serialized script] | Scripts for M-of-N Multi-signature Transaction. Standard script sending money to a script instead of a Bitcoin address (P2SH, BIP 16). The script must be one of the other standard output scripts.<br>The scriptPubKey in the funding transaction is script which ensures that the script supplied in the redeeming transaction hashes to the script used to create the address. In the scriptSig, 'signatures' refers to any script which is sufficient to satisfy the following serialized script. |
| | | | | OP_SMALLINT1 [pubKey][pubKey][pubKey] OP_SMALLINT2 OP_CHECKMULTISIG | | OP_0 [sig][sig][sig] | Standard script requiring multiple signatures to claim coins (BIP 11). |
| 5 | | | | [message] OP_DROP [pubKey] OP_CHECKSIG | | [sig] | Sample non-standard transaction including a message. |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 6 | Trans action puzzl e | | | OP_HASH 256 6fe28c0ab 6f1b372c1 a6a246ae 63f74f931 e8365e15 a089c68d 61900000 00000 OP_EQUA L | | --- | Transaction a4bfa8ab6435ae5f25dae9d89e 4eb67dfa94283ca751f393c1ddc 5a837bbc31b is an interesting puzzle. To spend the transaction you need to come up with some data such that hashing the data twice results in the given hash.<br>This transaction was successfully spent by 09f691b2263260e71f363d1db5 1ff3100d285956a40cc0e4f8c8c 2c4a80559b1. The required data happened to be the Genesis block, and the given hash was the genesis block hash. Note that while transactions like this are fun, they are not secure, because they do not contain any signatures and thus any transaction attempting to spend them can be replaced with a different transaction sending the funds somewhere else. |

# Appendix F – Script Parameters Names

| Parameter Name | Description |
|---|---|
| [pubKeyHash] | A Part of Btc address: RIPEMD160(SHA256(PubKey)) |
| [pubKey] | Public Key |
| [script] | Script |
| [scriptHash] | Script hash |
| [redeemScript] | 20-byte hash of redeem script |
| [sig] | Signature |
| [message] | String of message |
| [hashOfScript] | Hash of Script |
| [data] | Any data |
| PUSHDATA | The next byte contains the number of bytes to be pushed onto the stack. |
| 6fe28c0ab6f1b372c1a6a246ae63f74f931e8365e15a089c68d61900000000 | Transaction puzzle data. Hash |

# Appendix G – Value Conversion

| Value | Conversion |
| --- | --- |
| big-endian convention | Stores data **big-end first**. When looking at multiple bytes, the first byte (lowest address) is the **biggest**.<br><br>The resulting sequence *q* is converted to an integer value using the big-endian convention: If input bits are called *b_0* (leftmost) to *b_(qLen-1)* (rightmost), then the resulting value is<br>$b\_0 * 2^{(qLen-1)} + b\_1 * 2^{(qLen-2)} + \ldots + b\_(qLen-1) * 2^0$<br>where *qLen* is the binary length of *q*<br><br>Example: Decimal: 1025<br>16 bit representation in memory: Hex: 0x0401, Binary: 00000100 00000001<br>32 bit representation in memory: Hex: 0x00000401,<br>Binary: 00000000 00000000 00000100 00000001 |
| little-endian convention | Stores data **little-end first**. When looking at multiple bytes, the first byte is **smallest**.<br><br>The resulting sequence *q* is converted to an integer value using the little-endian convention: If input bits are called *b_0* (leftmost) to *b_(qLen-1)* (rightmost), then the resulting value is<br>$b\_0 * 2^0 + b\_1 * 2^1 + \ldots b\_(qLen-2) * 2^{(qLen-2)} + b\_(qLen-1) * 2^{(qLen-1)}$<br>where *qLen* is the binary length of *q*<br><br>Example: Decimal: 1025<br>16 bit representation in memory: Hex: 0x0104, Binary: 00000001 00000100<br>32 bit representation in memory: Hex: 0x01040000,<br>Binary: 00000001 00000100 00000000 00000000 |
| reversed bytes operation | You’ll have to reverse the bytes around (Not the digits).<br><br>Example:<br><br>12345678 gets reversed in bytes, so: - 78 = first byte 56 = second byte 34 = third byte 12 = forth byte<br><br>Making 0x78563412 from 0x1234678 |

<table>
<tr><td></td><td>Note that the bytes representing the entire number are swapped. Also note that only the bytes are reversed and the bits within the byte are NOT reversed.</td></tr>
<tr><td>1 Btc</td><td>$10^8$ = 100,000,000 Satoshi</td></tr>
<tr><td>1 byte</td><td>A sequence pf 8 bits (or 2 chars in the byte string). A bit has two values: on or off, 1 or 0. The “leftmost” bit in a byte is the biggest.<br>Example: the binary sequence 00001001 is the decimal number 9.<br>00001001 = ($2^3 + 2^0$ = 8 + 1 = 9). Bits are numbered from right-to-left.<br>Bit 0 is the rightmost and the smallest; bit 7 is leftmost and largest in this example.</td></tr>
<tr><td>1 Satoshi</td><td>1E-8 = $10^{-8}$</td></tr>
<tr><td>VarInt</td><td>Integer can be encoded depending on the represented value to save space. Variable length integers always precede an array/vector of a type of data that may vary in length. Longer numbers are encoded in little endian.
<table>
<tr><td>Value in hex</td><td>Value in dec</td><td>Storage length in bytes</td><td>Format</td></tr>
<tr><td>< 0xFD</td><td>< 253</td><td>1</td><td>uint8_t</td></tr>
<tr><td><= 0xFFFF</td><td><= 65535</td><td>3</td><td>0xFD followed by the length as uint16_t</td></tr>
<tr><td><= 0xFFFF FFFF</td><td><= 4294967295</td><td>5</td><td>0xFE followed by the length as uint32_t</td></tr>
<tr><td>-</td><td>-</td><td>9</td><td>0xFF followed by the length as uint64_t</td></tr>
</table>

```
// testValue
unsigned long long testValue = 0xFFFFFFFFFFFFFFFF; // 18446744073709551615
// 1 byte -> [0-255] or [0x00-0xFF]
uint8_t        number8 = testValue; // 255
unsigned char  numberChar = testValue; // 255
// 2 bytes -> [0-65535] or [0x0000-0xFFFF]
uint16_t       number16 = testValue; // 65535
unsigned short numberShort = testValue; // 65535
// 4 bytes -> [0-4294967295] or [0x00000000-0xFFFFFFFF]
uint32_t     number32 = testValue; // 4294967295
unsigned int numberInt = testValue; // 4294967295
```
</td></tr>
</table>

| | |
|---|---|
| | ```<br>// 8 bytes -> [0-18446744073709551615] or [0x0000000000000000-0xFFFFFFFFFFFFFFFF]<br>uint64_t      number64 = testValue; // 18446744073709551615<br>unsigned long long numberLongLong = testValue; // 18446744073709551615<br>``` |

# Appendix H – Binary <–> Decimal Conversions

**Binary to Decimal:**

| Binary base 2 | Decimal base 10 | Formula |
|---|---|---|
| 0 | 0 | $0_2 = 0\cdot2^0 = 0_{10}$ |
| 1 | 1 | $1_2 = 1\cdot2^0 = 1_{10}$ |
| 10 | 2 | $10_2 = 1\cdot2^1+0\cdot2^0 = 2_{10}$ |
| 11 | 3 | $11_2 = 1\cdot2^1+1\cdot2^0 = 3_{10}$ |
| 100 | 4 | $100_2 = 1\cdot2^2 +0\cdot2^1+0\cdot2^0 = 4_{10}$ |
| 101 | 5 | $101_2 = 1\cdot2^2 +0\cdot2^1+1\cdot2^0 = 5_{10}$ |
| 110 | 6 | $110_2 = 1\cdot2^2 +1\cdot2^1+0\cdot2^0 = 6_{10}$ |
| 111 | 7 | $111_2 = 1\cdot2^2 +1\cdot2^1+1\cdot2^0 = 7_{10}$ |
| 1000 | 8 | $1000_2 = 1\cdot2^3 +0\cdot2^2 +0\cdot2^1+0\cdot2^0 = 8_{10}$ |
| 1001 | 9 | $1001_2 = 1\cdot2^3 +0\cdot2^2 +0\cdot2^1+1\cdot2^0 = 9_{10}$ |
| 1010 | 10 | $1010_2 = 1\cdot2^3 +0\cdot2^2 +1\cdot2^1+0\cdot2^0 = 10_{10}$ |
| 1011 | 11 | $1011_2 = 1\cdot2^3 +0\cdot2^2 +1\cdot2^1+1\cdot2^0 = 11_{10}$ |
| 1100 | 12 | $1100_2 = 1\cdot2^3 +1\cdot2^2 +0\cdot2^1+0\cdot2^0 = 12_{10}$ |
| 1101 | 13 | $1101_2 = 1\cdot2^3 +1\cdot2^2 +0\cdot2^1+1\cdot2^0 = 13_{10}$ |
| 1110 | 14 | $1110_2 = 1\cdot2^3 +1\cdot2^2 +1\cdot2^1+0\cdot2^0 = 14_{10}$ |
| 1111 | 15 | $1110_2 = 1\cdot2^3 +1\cdot2^2 +1\cdot2^1+1\cdot2^0 = 15_{10}$ |
| 10000 | 16 | $10000_2 = 1\cdot2^4 +0\cdot2^3 + 0\cdot2^2 +0\cdot2^1+0\cdot2^0 = 16_{10}$ |
| 100000 | 32 | $100000_2 = 1\cdot2^5 +0\cdot2^4 +0\cdot2^3 + 0\cdot2^2 +0\cdot2^1+0\cdot2^0 = 32_{10}$ |
| 1000000 | 64 | $1000000_2 = 1\cdot2^6 +0\cdot2^5 +0\cdot2^4 +0\cdot2^3 + 0\cdot2^2 +0\cdot2^1+0\cdot2^0 = 64_{10}$ |
| 10000000 | 128 | $10000000_2 = 1\cdot2^7 +0\cdot2^6 +0\cdot2^5 +0\cdot2^4 +0\cdot2^3 + 0\cdot2^2 +0\cdot2^1+0\cdot2^0 = 128_{10}$ |
| 100000000 | 256 | $100000000_2 = 1\cdot2^8+0\cdot2^7+0\cdot2^6+0\cdot2^5+0\cdot2^4+0\cdot2^3+ 0\cdot2^2+0\cdot2^1+0\cdot2^0 =256_{10}$ |

The decimal number is equal to the sum of powers of 2 of the binary number’s ‘1’ digits place.

Example: 1110012 = $1\cdot2^5+1\cdot2^4+1\cdot2^3+0\cdot2^2+0\cdot2^1+1\cdot2^0 = 57_{10}$

**Decimal to Binary:**

“Divide by 2” algorithm is used to convert integer values into binary numbers. “Divide by 2” algorithm starts a conversion with an integer greater than 0. A simple iteration then continually divides the decimal number by 2 and keeps track of the remainder. The first division by 2 gives information as to whether the value is even or odd. An even value will have a remainder of 0. It will have the digit 0 in the ones place. An odd value will have a remainder of 1 and will have the digit 1 in the ones place. The binary number is a sequence of digits, where the first computed remainder be the last digit in the sequence.

Example:

322/2 = 116 + 1 -> rem 1

116/2 = 58 + 0 -> rem 0

58/2 = 29 + 0 -> rem 0

29/2 = 14 + 1 -> rem 1

14/2 = 7 + 0 -> rem 0

7/2 = 3 + 1 -> rem 1

3/2 = 1 + 1 -> rem 1

1/2 = 0 + 1 -> rem 1

Result: $11101001_2 = 322_{10}$

# Appendix I – Hex <–> Decimal Conversions

**Hex to Decimal:**

| Hex base 16 | Decimal base 10 |
|---|---|
| 0 | 0 |
| 1 | 1 |
| 2 | 2 |
| 3 | 3 |
| 4 | 4 |
| 5 | 5 |
| 6 | 6 |
| 7 | 7 |
| 8 | 8 |
| 9 | 9 |
| A | 10 |
| B | 11 |
| C | 12 |
| D | 13 |
| E | 14 |
| F | 15 |
| 10 | 16 |
| 20 | 32 |
| 30 | 48 |
| 40 | 64 |
| 50 | 80 |
| 60 | 96 |
| 70 | 112 |
| 80 | 128 |
| 90 | 144 |
| A0 | 160 |
| B0 | 176 |
| C0 | 192 |

| | |
|---|---|
| D0 | 208 |
| E0 | 224 |
| F0 | 240 |
| 100 | 256 |
| 200 | 512 |
| 300 | 768 |
| 400 | 1024 |

A regular decimal number is the sum of the digits multiplied with $10^n$.

**Example #1**

137 in base 10 is equal to each digit multiplied with its corresponding $10^n$:

$137_{10} = 1\times10^2+3\times10^1+7\times10^0 = 100+30+7$

Hex numbers are read the same way, but each digit counts $16^n$ instead of $10^n$. Multiply each digit of the hex number with its corresponding $16^n$.

**Example #2**

3B in base 16 is equal to each digit multiplied with its corresponding $16^n$:

$3B_{16} = 3\times16^1+11\times16^0 = 48+11 = 59$

**Example #3**

E7A9 in base 16 is equal to each digit multiplied with its corresponding $16^n$:

$E7A9_{16} = 14\times16^3+7\times16^2+10\times16^1+9\times16^0 = 57344+1792+160+9 = 59305$

**Decimal to Hex:**

For decimal number X:

9. Get the highest power of 16 that is less than the decimal number X:

   $16^n < x, (n=1,2,3,\ldots)$

10. The high hex digit is equal to the integer if the decimal number X divided by the highest power of 16 that is smaller than X:

    $d_n = \text{int}(x / 16^n)$

3. Calculate the difference $\Delta$ of the number X and the hex digit dn times the power of 16, $16^n$:

   $\Delta = x - d_n \times 16^n$

4. Repeat step #1 with the difference result until the result is 0.

**Example:** Convert X=603 to hex:

Step 1: $n=2$, $16^2=256 < 603$

$n=3,\ 16^3=4096 > 603$

So $n = 2$

Step 2: $d_2 = \text{int}(603 / 16^2) = 2$

Step 3: $\Delta = 603 - 2\times16^2 = 91$

Repeat step #1 with the difference result until the result is 0.

$N = 1,\ x = \Delta = 91$

$d_1 = \text{int}(91 / 16^1) = 5$

$\Delta = 91 - 5\times16^1 = 11$

$n = 0,\ x = \Delta = 11$

$d_0 = \text{int}(11 / 16^0) = 11_{10} = B_{16}$

$\Delta = 11 - 11\times16^0 = 0$

$(d_2d_1d_0) = 25B$

Result:

$x = 603_{10} = 25B_{16}$

# Appendix J – Common prefixes for version bytes

| Type | Version prefix (hex) | Base58 result prefix |
|---|---|---|
| Bitcoin Address | 0x00 | 1 |
| Pay-to-Script-Hash Address | 0x05 | 3 |
| Bitcoin Testnet Address | 0x6F | m or n |
| Private Key WIF | 0x80 | 5, K or L |
| BIP38 Encrypted Private Key | 0x0142 | 6P |
| BIP32 Extended Public Key | 0x0488B21E | xpub |

# Appendix K – Sighash Type codes [2.22]

| Sighash Type | Value | Description |
|---|---|---|
| SIGHASH_ALL | 0x00000001 | This type is default. Type signs all the inputs and outputs, protecting everything except the signature scripts against modification. |
| SIGHASH_NONE | 0x00000002 | Type signs all of the Inputs but none of the Outputs, allowing anyone to change where the satoshis are going unless other signatures using other signature hash flags protect the outputs. |
| SIGHASH_SINGLE | 0x00000003 | Type code the only Output signed is the one corresponding to this Input (the Output with the same output index number as this Input), ensuring nobody can change your part of the transaction but allowing other signers to change their part of the transaction. The corresponding Output must exist or the value “1” will be signed, breaking the security scheme. This Input, as well as other Inputs, are included in the signature. The sequence numbers of other Inputs are not included in the signature, and can be updated. |
| SIGHASH_ANYONECANPAY | 0x00000080 | 1. The txCopy input vector is resized to a length of one.<br>2. The 339ubscript (lead in by its length as a var-integer encoded!) is set as the first and only member of this vector.<br>Think of this as “Let other people add inputs to this transaction, I don’t care where the rest of the bitcoins come from.” |

# Appendix L – IntDS Error Codes

See SubSystem Abbreviations in the start of document: “Acronyms and Abbreviations of the Current Document”.

| ID | Error Code | SubSystem Abbreviation | Error Description |
|---|---|---|---|
| 1 | Balance calculation error | STrxMSS | STrxMSS error in the process of Wallet balance calculation. |
| 2 | Wallet creation error | STrxMSS | STrxMSS error in the process of new Wallet creation. |
| 3 | Wallet was not found | STrxMSS | STrxMSS can not find Wallet or error was generated in this process. |
| 4 | Wallet signature validation error | STrxMSS | STrxMSS error in the process of Wallet signature validation. |
| 5 | Status was not found | STrxMSS | STrxMSS can not find transaction status or error was generated in this process. |
| 6 | Transaction creation error | STrxMSS | STrxMSS error in the process of new transaction creation. |
| 7 | Transaction send error | STrxMSS | STrxMSS error in the sending of transaction to blockchain |
| 8 | Error of Temp Transaction deleting | STrxMSS | STrxMSS error in the deleting of temporary transaction data |
| 9 | Error of Transferring Funds | STrxMSS | STrxMSS error in the process of transferring Wallet dependencies to another Wallet |
| 10 | STrxMSS error | STrxMSS | STrxMSS error |
| 11 | Trx data was not found | STrxMSS | STrxMSS can not find transaction data or error was generated in this process. |
| 12 | Inbound Trx was not found for Btc address | STrxMSS | STrxMSS can not find Inbound transaction for given Btc address or error was generated in this process. |

| | | | |
|---|---|---|---|
| 13 | Error in the creation of Btc address | STrxMSS | STrxMSS error in the process of new Btc address creation. |
| 14 | Error of Locking Wallet | STrxMSS | STrxMSS error in the process of locking Wallet. |
| 15 | Data of system error was not found | STrxMSS | STrxMSS can not find error data or error was generated in this process. |
| 16 | Rejection message was not found | STrxMSS | STrxMSS can not find rejection message or error was generated in this process. |
| 17 | Btc address is invalid | STrxMSS | STrxMSS Btc address validation result: Btc address is invalid |
| 18 | Btc funds is not enough | STrxMSS | STrxMSS result of transaction creation: Btc funds is not enough in the current wallet to create transaction |
| 19 | createSingleSigTrx fnc validation error | STrxMSS | STrxMSS error in the process of "createSingleSigTrx" function data validation. |
| 20 | deleteTempTrx fnc validation error | STrxMSS | STrxMSS error in the process of "deleteTempTrx" function data validation. |
| 21 | sendSingleSigTrx fnc validation error | STrxMSS | STrxMSS error in the process of "sendSingleSigTrx" function data validation. |
| 22 | Wallet data validation error | STrxMSS | STrxMSS error in the process of validating wallet data. |
| 23 | Transaction data validation error | STrxMSS | STrxMSS error in the process of validating transaction data. |
| 24 | Transaction record error | STrxMSS | STrxMSS error in the process of Trx record creation |
| 25 | UTXOs selection problem | STrxMSS | STrxMSS cannot select UTXOs for given Wallet. |
| 26 | Error in the creation of Private Key | STrxMSS | STrxMSS error in the process of new Private Key creation. |

| | | | |
|---|---|---|---|
| 27 | Error in the creation of Public Key | STrxMSS | STrxMSS error in the process of new Public Key creation. |
| 28 | Mnemonic seed restoring error | STrxMSS | STrxMSS error in the process of Mnemonic seed restoring |
| 29 | Wallet record error | STrxMSS | STrxMSS error during creation of wallet record |
| 30 | Mnemonic code generation error | STrxMSS | STrxMSS error during generating the user part of wallet private key |
| 31 | MNM record error | STrxMSS | STrxMSS error during creation of MNM record |

# Appendix M – Blockchain Rejection Messages

This table keeps data which should be captured by BTC_REJECTION_MSG table from “shared_data” DB. There are 4 categories at this moment [2.25]:

- Block
- Common
- Transaction
- Version

| ID | Message Code | Category | Message Description |
|---|---|---|---|
| 1 | 10 | Block | Block is invalid for some reason (invalid proof-of-work, invalid signature, etc) |
| 2 | 11 | Block | Block's version is no longer supported |
| 3 | 43 | Block | Inconsistent with a compiled-in checkpoint |
| 4 | 01 | Common | Message could not be decoded |
| 5 | 10 | Transaction | Transaction is invalid for some reason (invalid signature, output value greater than input, etc.) |
| 6 | 12 | Transaction | An input is already spent |
| 7 | 40 | Transaction | Not mined/relayed because it is "non-standard" (type or version unknown by the server) |
| 8 | 41 | Transaction | One or more output amounts are below the 'dust' threshold |
| 9 | 42 | Transaction | Transaction does not have enough fee/priority to be relayed or mined |
| 10 | 11 | Version | Client is an obsolete, unsupported version |
| 11 | 12 | Version | Duplicate version message received |

# Glossary

| Definition | Description |
|---|---|
| Affine coordinates | In mathematics: An **Affine coordinate** system is a coordinate system on an *Affine Space* where each coordinate is an *Affine Map* to the *Number Line*. [3.10]<br>- An **Affine Space** is a geometric structure that generalizes certain properties of parallel lines in Euclidean space.<br>- An **Affine Map** is a function between *Affine Spaces* which preserves points, straight lines and planes.<br>- A **Number Line** is a picture of a straight line on which every point is assumed to correspond to a real number and every real number to a point |
| BIP | A Bitcoin Improvement Proposal and is one of the mechanisms used by the Bitcoin "core developers" to improve Bitcoin [2.8], [2.9]. |
| Bitcoin address | A 160-bit hash of the ECDSA public key (public portion of a public/private **ECDSA** key pair) |
| Block | Data is permanently recorded in the Bitcoin network through files called **Blocks**. A **Block** is a record of some or all of the most recent Bitcoin transactions that have not yet been recorded in any prior blocks. New blocks are added to the end of the record (known in Bitcoin as the **Block Chain**), and once written, are never changed or removed. Each block memorializes what took place immediately before it was created [2.5].<br>Every block contains a hash of the previous block. This has the effect of creating a chain of blocks from the genesis block to the current block. Each block is guaranteed to come after the previous block chronologically because the previous block's hash would otherwise not be known. Each block is also computationally impractical to modify once it has been in the chain for a while because every block after it would also have to be regenerated. Each block has a size limit of 1,000,000 bytes. |
| Block Chain | A transaction database shared by all nodes participating in a system based on the Bitcoin protocol [2.13]. A chain is valid if all |

| | |
|---|---|
| | of the blocks and transactions within it are valid, and only if it starts with the genesis block. For any block on the chain, there is only one path to the genesis block [2.3]. |
| Coinbase Transaction | A special kind of transaction, has no **Inputs**. It is created by miners, and there is one **Coinbase** transaction per **Block**. Because each block comes with a reward of newly created Bitcoins (e.g. 50 BTC for the first 210,000 blocks), the first transaction of a block is, with few exceptions, the transaction that grants those coins to their recipient (the Miner). |
| DPA attack | Differential Power Analysis attack is a type of Power consumption attack. DPA is SPA plus statistical analysis such as data dependencies on power consumption to crack the system |
| Dust | A transaction output is considered dust when the cost of spending it is close to its value.<br>Precisely, Bitcoin Core defines dust to be an output whose fees exceed 1/3 of its value. This computes to everything smaller than 546 satoshis being considered dust by Bitcoin Core. |
| Genesis Block | The first block of a block chain [2.4]. Modern versions of Bitcoin assign it block number 0, though older versions gave it number 1. |
| Inputs | Records which reference the funds from other previous transactions. |
| Octet | Sequences of eight bits. The first (leftmost) bit within an octet has numerical value 128, while the last (rightmost) has numerical value 1. 8 bits = 1 byte = 2 chars in the byte string |
| Outputs | Records which determine the new owner of the transferred Bitcoins, and which will be referenced as **Inputs** in future transactions as those funds are respent. |
| Mnemonic Code | *Generally*: Mnemonics aim to translate information into a form that the brain can retain better than its original form.<br>*In the scope of this document*: A mnemonic code or mnemonic sentence is a group of easy to remember words. |
| NAF or wNAF | An binary signed-digit representation known as **w-ary Non-Adjacent Form** of the number. It’s a unique integer representation |

| | |
|---|---|
| | For ***2****NAF*, $w = 2$<br><br>*i*-bit integer $d = (d_{i-1}, d_{i-2} . . . d_0), d_i \in \{1, -1, 0\}$ |
| Public Key | A number that corresponds to a private key, but does not need to be kept secret. A public key can be calculated from a private key, but not vice versa. A public key can be used to determine if a signature is genuine (in other words, produced with the proper key) without requiring the private key to be divulged.<br><br>In Bitcoin, public key are either compressed or uncompressed. Compressed public keys are 33 bytes, consisting of a prefix either 0x02 or 0x03, and a 256-bit integer called **X**. The older uncompressed keys are 65 bytes, consisting of constant prefix (0x04), followed by two 256-bit integers called **X** and **Y** (2 * 32 bytes). The prefix of a compressed key allows for the **Y** value to be derived from the **X** value. |
| 1 Satoshi | All values in the Bitcoin network are integers in Satoshis (1E-8 BTC) so technically all the numbers would be multiplied by 1E8.<br><br>1 BTC = 100,000,000 Satoshi. |
| scriptSig | Contains a signature and a public key. |
| SPA attack | Simple Power Annalysis attack is a type of Power consumption attack. SPA simply interprets power consumption into visual representation during the operation of a device or system, and such information may leak important information about the system. |
| Transaction | A Cryptographically signed section of data that is broadcast to the network and collected into blocks. It typically references previous transactions and reassign ownership of Bitcoins from them to one or more new bitcoin addresses. So, Transactions have **Inputs** and **Outputs**. It is not encrypted, so it is possible to browse and view any transaction to ever be collected into a block [2.1]. |
| | |

# Project Authorisation

**Project Identification** Project “Intelligent Daemon System”

**Assigned Priority** High

**Commencement Date:** ______/______/______.

## Executive Group

______________________________

**Chief Executive Officer**

**Date:** _____/_____/_____.

## Development Group

______________________________

**Program Manager**

**Date:** _____/_____/_____.