**EPSMI Documentation**

**(Version 0.0.4** – 02.01.2021**)**

**Proprietary Notes**

EPSMI is a product of MHM Software, license to be determined.

**Table of Contents**

[1 Overview 3](#_Toc60823861)

[1.1 What is present in EPS and missing in EPSMI 3](#_Toc60823862)

[1.2 Features of EPSMI going beyond EPS 3](#_Toc60823863)

[2 Installation and Setup 3](#_Toc60823864)

[2.1 Local Bitcoin Node 3](#_Toc60823865)

[2.2 Electrum Wallet 3](#_Toc60823866)

[2.3 EPSMI 3](#_Toc60823867)

[2.3.1 Download binary and run 3](#_Toc60823868)

[2.3.2 Build your own and run 3](#_Toc60823869)

[2.3.3 EPSMI Configuration 3](#_Toc60823870)

[2.3.4 Preparing keystore file for SSL 4](#_Toc60823871)

[3 API 4](#_Toc60823872)

[3.1 Miscellaneous Queries by Electrum Wallet 4](#_Toc60823873)

[3.1.1 blockchain.scripthash.get\_balance 5](#_Toc60823874)

[3.2 Block-related Queries by Electrum Wallet 6](#_Toc60823875)

[3.2.1 blockchain.block.header 6](#_Toc60823876)

[3.2.2 blockchain.block.get\_header 7](#_Toc60823877)

[3.2.3 blockchain.block.headers 7](#_Toc60823878)

[3.2.4 blockchain.block.get\_chunk 7](#_Toc60823879)

[3.3 Transaction-related Queries by Electrum Wallet 8](#_Toc60823880)

[3.3.1 blockchain.transaction.get 8](#_Toc60823881)

[3.3.2 blockchain.transaction.id\_from\_pos 9](#_Toc60823882)

[3.3.3 blockchain.transaction.get\_merkle 9](#_Toc60823883)

[3.3.4 blockchain.transaction.broadcast 9](#_Toc60823884)

[3.4 Subscription Queries 10](#_Toc60823885)

[3.4.1 blockchain.scripthash.subscribe 11](#_Toc60823886)

[3.4.2 blockchain.headers.subscribe 11](#_Toc60823887)

[4 Testing 12](#_Toc60823888)

[4.1 Run tests via SBT 12](#_Toc60823889)

[4.2 Manual Acceptance Testing in Testnet 12](#_Toc60823890)

[5 Full Configuration Documentation 13](#_Toc60823891)

[6 Modules, Internals 14](#_Toc60823892)

[6.1 Transaction Monitor 14](#_Toc60823893)

[6.1.1 buildAddressHistory (External) 14](#_Toc60823894)

[6.1.2 checkForUpdatedTxs (External) 14](#_Toc60823895)

[6.1.3 getAddressBalance (External) 14](#_Toc60823896)

[6.1.4 checkForNewTxs 15](#_Toc60823897)

[6.1.5 getInputAndOutputScriptpubkeys 15](#_Toc60823898)

[6.1.6 generateNewHistoryElement 15](#_Toc60823899)

[6.1.7 checkConfirmations 15](#_Toc60823900)

[6.1.8 checkForReorganizations 15](#_Toc60823901)

[7 Issues 15](#_Toc60823902)

[8 Future extensions 15](#_Toc60823903)

# Overview

EPSMI is the Scala implementation of the original Electrum Personal Server written in Python by Chris Belcher.

Intention of EPSMI is to provide alternative implementation, which could be more accessible and readable to Scala and Java developers, as well as subjectively more accessible for people who find Scala structure and type system more readable than Python.

Another intention is to ease the maintenance burden, again, given Scala-inclined developers are in charge, and last but not least, to ease the effort of adding new features.

## What is present in EPS and missing in EPSMI

* TOR
* Rescan is a command line parameter in EPS, while configuration setting in EPSMI

## Features of EPSMI going beyond EPS

* TBD ☺

# Acknowledgements

# Installation and Setup

## Local Bitcoin Node

## Electrum Wallet

## EPSMI

### Download binary and run

Check integrity and fingerprint after download.

### Build your own and run

### EPSMI Configuration

### Preparing keystore file for SSL

# API

## Miscellaneous Queries by Electrum Wallet

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Name | Description | Input Parameters | Output Parameters | Errors and Exceptions |
| server.ping |  | None | None | TODO |
| server.banner | Provides information about server, blockchain and Bitcoin network that is visible in the console tab of Electrum app. | None | String with banner text. | Calls networkInfo and blockchainInfo – may throw exception (TODO – see how these exceptions are handled). |
| server  .donation\_address | Provides donation address which will be used by Electrum app in the Help/Donate to server function | None | String with Bitcoin address that can be used as Electrum payment destination. | - |
| mempool  .get\_fee\_histogram | The histogramisan array of [fee, vsize] pairs, where vsizen is the cumulative virtual size of mempool transactions with a fee rate in the interval [feen-1, feen], and feen-1 > feen.  (Deprecated? – check it out, this API is resource intensive and is currently switched off – TODO – turn it on and test**)** | None | Array[Array[Int]] – array of pairs of integers for fee and vsize |  |
| server.version | Note that returned server version must be 1.4, changing it has serious implications as some logic triggers on the client side, change it from 1.4 only if you know what you are doing | Two strings representing Electrum client version, currently ignored | Array of strings, consisting of server name and version |  |
| blockchain.estimatefee |  | waitBlocks – integer indicating urgency for the fee requested – in how many blocks confirmation is expected (must be in range 1-1008 | BigDecimal – fee rate | Error when waitBlocks parameter is not in the range 1-1008 |
| blockchain.relayfee |  | none | BigDecimal - minimum relay fee for transactions in BTC/kB, transactions with fee less than this won’t be broadcast by the node (currently it is 1000 satoshis) |  |
| blockchain.scripthash  .get\_balance | Confirmed and unconfirmed balances of a scripthash | String containing scripthash as hex, e.g.: 5be022609383  d23e2d545b3  b359446466c  269686c1e697  b60355424ed3  0490d2 | {  "confirmed": "1.03873966",  unconfirmed": "0.236844"  } | **TODO – futurize it** |
| blockchain.scripthash  .get\_history |  | String containing scripthash as hex, e.g.: 5be022609383  d23e2d545b3  b359446466c  269686c1e697  b60355424ed3  0490d2 | Array of HistoryItem, containing height, tx\_hash and fee (HistoryItem represents transaction in a scripthash history |  |

### blockchain.scripthash.get\_balance

For a given scripthash sh, we retrieve history elements from transaction monitor state transaction history. For each history element, we obtain txhash, from txhash we obtain decoded raw transaction (via 2 calls to bitcoin rpc). In a raw transaction we have access to vout, we filter vouts whose scriptpubkey is our sh, and collect amounts and confirmation counts. We also collect from vin utxos txhashes and delete them if thay happen to already be in our collection – this is to avoid counting output of one transaction is consumed by another for the same scripthash. At the end we sum separately amounts with zero confirmations and with 1 or more confirmations.

## Block-related Queries by Electrum Wallet

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Name | Description | Input Parameters | Output Parameters | Errors and Exceptions |
| blockchain.block  .header | Hex representation of block’s header information | Block height (integer) | Block header hex (string) |  |
| blockchain.block  .get\_header | Dictionary representation of block’s header information | Block height (integer) | Header result dictionary HeaderResult |  |
| blockchain.block  .headers | Concatenated hex representation of multiple block’s header information | Block start height and count | Result dictionary BlockHeadersResult |  |
| blockchain.block  .get\_chunk | Concatenated hex representation of multiple block’s header information | Desired 2016 blocks chunk number – first 2016 blocks when 0, second 2016 blocks when 1, etc. | Concatenated block header hexes for all blocks in a chunk (of size 2016 or less) |  |

### blockchain.block.header

Block height is converted to block hash via RPC getBlockHash, then block header structure is obtained via RPC getBlockHeader. Information from the header structure is packed according to the format:

<i32s32sIII

Which translates to:

little endian int | byte[32] | byte[32] | unsigned int | unsigned int | unsigned int

The following information is inserted:

blockHeader.version

blockHeader.previousblockhash

blockHeader.merkleroot

blockHeader.time

blockHeader.bits

blockHeader.nonce

Returned string length is 160 which is hex representation of the 80 bytes filled out as above (4+32+32+4+4+4).

The information is the same as bitcoin-cli getblockheader with verbose = false, e.g.:

***bitcoin-cli getblockheader 00000000c937983704a73af28acdec37b049d214adbda81d7e2a3dd146f6ed09 false***

Note that you need block hash as input for the above call, which you can obtain from block height via a call to getblockhash, e.g.:

***bitcoin-cli getblockhash 600000***

### blockchain.block.get\_header

Similar to blockchain.block.header, only information is returned not as array of bytes in hex, but rather as a dictionary with the following fields:

* Block height
* Previous block hash
* Timestamp
* Merkle root
* Version
* Nonce
* Bits

The information is the same as bitcoin-cli getblockheader with verbose = true, e.g.:

***bitcoin-cli getblockheader 00000000c937983704a73af28acdec37b049d214adbda81d7e2a3dd146f6ed09 true***

### blockchain.block.headers

Similar to blockchain.block.header, only returned hex contains concatenated byte arrays of many blocks. Returned BlockHeadersResult structure contains the following fields:

* Hex – hex string being a result of concatenation of 80-byte arrays (160 character hex strings), each in format as in the result of blockchain.block.header
* Count – effective count, maybe smaller than given count if highest block has been reached
* Max – maximum number of blocks that can be processed, it is a constant set to 2016

Note that implementation utilizes the fact that get RPC getblockheader returns hashes to the previous and next blocks. Hence, we only need to call RPC getblockhash, converting block height to block hash, once, and for subsequent blocks we can only call RPC getblockheader, always using the nextblockhash field from GetBlockHeaderResult. Note that GetBlockHeaderResult is a structure returned by RPC getblockheader.

### blockchain.block.get\_chunk

Similar to blockchain.block.headers, only does not return a structure but rather a single concatenated string. Input parameter is a chunk number, so it is 0, result will contain headers for the first 2016 blocks. If it is 1, result will contain headers for the second 2016 blocks, etc. If chunk exceeds the highest block, fewer than 2016 headers will be returned.

Result is a hex encoded string whose length is a multiple of 160, typically the length will be 2016\*160 (322560) or less if the chunk borders with the highest block.

## Transaction-related Queries by Electrum Wallet

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Name | Description | Input Parameters | Output Parameters | Errors and Exceptions |
| blockchain.transaction  .get | Provides hex representation of transaction data | Transaction id (string) | Hex representation of transaction (string) |  |
| blockchain.transaction  .id\_from\_pos | Provides n-th transaction id in a given block | Block height, position within block (pos, integer), include merkle information (merkle, boolean) | If merkle is false, returns pos-th transaction id within block of given height  If merkle is true, returns a struct with 2 fields: tx\_hash with pos-th transaction id and merkle with electrum merkle proof (list of strings) |  |
| blockchain.transaction  .get\_merkle | Provides merkle proof for a given transaction presence in the block | Transaction id (txid, string), block height (integer) | Struct with 3 fields: block\_height, pos (position within block) and merkle with electrum merkle proof (list of strings) |  |
| blockchain.transaction  .broadcast | Broadcasts transaction to the Bitcoin network | Hex-encoded raw bytes of the transaction to be broadcast (string) | Transaction id | Error if transaction is invalid or broadcasting is not allowed by configuration or the bitcoin node |

### blockchain.transaction.get

Accepts transaction id and returns transaction in hexadecimal form.

Calls RPC gettransaction which provides in-wallet transaction, e.g.:

***bitcoin-cli gettransaction "22667c482f0f69daefabdf0969be53b8d539e1d2abbfc1c7a193ae38ec0d3e31"***

Among fields returned by this call is a field hex, which contains raw data for the transaction.

If transaction is not in-wallet, the API uses another RPC, getrawtransaction, e.g.:

***bitcoin-cli getrawtransaction "b850bd9f727888019ddd5481124b83c17b9dd263fe4c7c007a0a6c0f4c0f1573"***

The second call returns only raw data for the transaction.

### blockchain.transaction.id\_from\_pos

Third parameter merkle decides if merkle proof information is required.

If merkle is false, the call boils down to calling RPC getblockhash to obtain blockhash from block height, then calling RPC getblock to obtain GetBlockResult, from which pos-th transaction id is taken from the field tx.

If merke is true, additional call is being made to gettxoutproof, which provides MerkleBlock, which is then converted to Electrum merkle proof format, which is ElectrumMerkleProof. The conversion disassembles raw data provided by gettxoutproof.

RPC gettxoutproof returns a serialized, hex-encoded proof that a given transaction is included in the block.

### blockchain.transaction.get\_merkle

Similar to blockchain.transaction.id\_from\_pos with merkle=true, yet here we only know transaction id, and we want to know the block and position within block and merkle proof of transaction presence in the block.

Given txid we call RPC getrawtransaction, from which we obtain block hash, which is fed to RPC getblockheader to obtain block header. This is needed to learn the block containing our transaction. Then we call RPC gettxoutproof to obtain merkle proof, and we deserialize it into Electrum merkle proof format, in the same way as in blockchain.transaction.id\_from\_pos.

### blockchain.transaction.broadcast

Submits serialized, hex-encoded raw transaction to Bitcoin network, returns transaction id or error if:

* transaction is invalid
* broadcast-method is not set to own-node in configuration
* RPC getnetworkinfo returns localrelay set to false (meaning broadcasting is disabled)

RPC testmempoolaccept is used to obtain TestMempoolAcceptResult, containing either the transaction id if field allowed set to true, or rejectReason (string) when allowed is set to false. The reject reason, if it is there, is passed back as error information.

testmempoolaccept checks if the transaction would be accepted by the mempool. In addition to consensus or policy rules, it will also reject transaction with fee rate higher than specific value, EPSMI hardcodes this value to 0.1 BTC (TODO move it to configuration).

sendrawtransaction is used for broadcasting transaction, once it passes the testpoolaccept test. Transaction will be sent to all peers.

getnetworkinfo is used to obtain flag localrelay, which is set to true if the node is able and willing to send transactions, and also receive transactions from peers to be sent. Original information for this flag: ***true if transaction relay is requested from peers*** is confusing and means rather ***true if we do not want to send or receive transactions***, as explained in: ***https://github.com/bitcoin/bitcoin/pull/8049.***

## Subscription Queries

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Name | Description | Input Parameters | Output Parameters | Errors and Exceptions |
| blockchain.scripthash  .subscribe | Turns on subscription for a given scripthash, will cause notifications to be sent back to the caller whenever scripthash state changes | scripthash (string) | status of the scripthash (history hash) – see below,  causes subsequent notifications on the same socket, whenever status of the scripthash changes | Error if scripthash is not found in transaction monitor’ address history |
| blockchain.headers  .subscribe | Turns on new block subscription, will case notifications to be sent back to the caller whenever there is a new block | None | returns struct with two fields: height (current top block height) and hex (current top block header hex),  causes subsequent notifications on the same socket, whenever there is a new block |  |
| server.peers  .subscribe | It is not a subscription, simply returns a list of peer servers | None | Returns an empty array of strings, as there are no peers in EPSMI deployment, in which EPSMI is a sole server obtaining all network information from the local bitcoin node via RPC |  |

### blockchain.scripthash.subscribe

Causes lasting effect of sending notification whenever status of a given scripthash changes. Immediate return carries current status.

By status of a scripthash we mean either null (if no transactions are present to the scripthash) or a sha256 of an ordered list of strings in a format <txid>:height. The order is by increasing height and position within a block, if more than one transaction is in the same block. Such list is concatenated with a list of transactions not confirmed yet, with height 0 or -1. The latter if at least one of the inputs of a transaction is unconfirmed.

Sha256 of the above list is dubbed status of a scripthash, such status will change whenever there is new transaction to the scripthash, or whenever some attribute of any of the transactions in the list changes, like number of confirmations.

Note that notifications are sent back on the same socket. In order to learn about changes in transaction list for a given scripthash, periodical checks are performed, triggered by a heartbeat. The check prepares a list of updated transactions and sends this list via onUpdtatedScripthashes function call (name is implementation detail, but it conveys the idea). For every updated scripthash the following notification is sent:

{

"jsonrpc": "2.0",

"method": "blockchain.scripthash.subscribe",

"params": ["<scripthash>", "<status (history hash)>"]

}

The above notification is pushed into the socket every time a heartbeat-drive check detects change in scripthash history.

### blockchain.headers.subscribe

Causes lasting effect of sending notification whenever there is a new block on the Bitcoin blockchain. Immediate return carries current top block height and current top block header hash.

Heartbeat triggers two following RPC calls: getbestblockhash and getblockheader. getbestblockhash provides best blockhash, which is fed to getblockheader, which returns GetBlockHeaderResult, which is serialized into header hex (serialized raw bytes with header information in the format <i32s32sIII).

For every new block detected, the following notification is sent:

{

"jsonrpc": "2.0",

"method": "blockchain.headers.subscribe",

"params": [{"hex": "<top block header hex>", "height": <top block height>}]

}

# Testing

## Run tests via SBT

***sbt test -Djavax.net.ssl.keyStore=/Users/miloszm/proj/epsmi/rpcserver2.jks -Djavax.net.ssl.keyStorePassword=123456***

Note – requires running a fully synchronized Bitcoin node, and a local Electrum wallet.

TODO – make sure tests run with any wallet content

## Manual Acceptance Testing in Testnet

Switch local Bitcoin node to Testnet. Make sure it is fully synchronized.

Switching is done in bitcoin.conf, change testnet=0 to testnet=1

You can open Bitcoin Core configuration file by going to Bitcoin Core/Preferences, and push the button Open Configuration File.

Switch EPSMI to Testnet.

You can do it by changing testnet=false to testnet=true in EPSMI configuration.conf file.

Start Electrum wallet in Testnet mode.

This can be done, for example, by starting Electrum via the following command:

***/Applications/Electrum.app/Contents/MacOS $./run\_electrum --testnet***

Make sure your wallet is only using EPSMI as the Electrum server.

Go to Testnet faucet and send some bitcoin to your wallet.

***https://bitcoinfaucet.uo1.net/send.php***

Make sure new bitcoin appears correctly in your wallet, and confirmation process is shown as expected.

Send some bitcoin from your wallet back to the faucet (or any other destination) and make sure your wallet balances change as expected, see the confirmation process being show as expected.

# Full Configuration Documentation

|  |  |  |
| --- | --- | --- |
| subscription.link | "-na-" | Used for "Manage Subscription" link in the digest. |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

Note

As example, current production settings are as follows:

|  |  |
| --- | --- |
| Entry Name | Entry Value |
| cerebro.activity.sso.url | http://sappc2ci.swissre.com:8080/sap/zcrm\_sso?crmparam= |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

# Modules, Internals

## Transaction Monitor

Transaction monitor is a set of pure functions operating on transaction monitor state. Note that transaction monitor state is immutable, methods (or functions) of transaction monitor always create and return new state based on the old state.

The external methods are:

* buildAddressHistory
* checkForUpdatedTxs
* getAddressBalance

### buildAddressHistory (External)

Called by main at initialization time, with a list of spks (script public keys, i.e. addresses) and wallets.

### checkForUpdatedTxs (External)

Called by code triggered by heartbeat – returns new transaction monitor state and a list of updated addresses (scripthashes).

### getAddressBalance (External)

Called by the API blockchain.scripthash.get\_balance, returns current confirmed and unconfirmed balances based on the history of scripthash transactions as per transaction monitor state.

To most important internal methods belong:

* checkForNewTxs
* getInputAndOutputScriptpubkeys
* generateNewHistoryElement
* checkConfirmations
* checkForReorganizations

### checkForNewTxs (Internal)

Produces new transaction monitor state to reflect detect new transactions. Calls RPC listtransactions, which lists current wallet transactions. listtransactions accepts argument count, which limits number of transactions to list. checkForNewTxs calls listtransactions in a loop increasing count until if finds last known transaction, known in a sense that it is present in the transaction monitor state.

The following comment from original EPS (authored by Chris Belcher) explains how listtransactions works:

## skip and count parameters take most-recent txes first  
## so skip=0 count=1 will return the most recent tx  
## and skip=0 count=3 will return the 3 most recent txes  
## but the actual list returned has the REVERSED order  
## skip=0 count=3 will return a list with the most recent tx LAST

Hence, listtransactions is used to detect new transactions. If we remember last known transaction and find it in a list returned by list transactions, we can be sure that all transactions in the list **after** our found transaction are new. We need to list enough transactions, so that we can find our known one. We start with 2, then 4, and keep doubling the count until we either find known transaction or reach 256. If we don’t have known transaction yet or we don’t see known transaction even after we read 256 transactions, that we assume all transactions are new. If we find known transaction for, say, count=x, but number of read transactions is x, we may not have listed all new transactions, so we continue iterating until, ideally, known is found and number of read transactions is not equal count.

Once we found new transactions, we need to create new transaction monitor state which will contain new transactions. We use word update state in code, although the state is never updated in place, as it is immutable. Old state is copied into new state, and only to this state new transactions are added. State update uses two methods that are explained in dedicated sections:

* getInputAndOutputScriptpubkeys
* generateNewHistoryElement

### getInputAndOutputScriptpubkeys (Internal)

Based on transaction id, returns two lists: a list of input spks, a list of output spks, and a decoded transaction.

By spk we mean here script public key, which is a form of bitcoin address. Getting a list of output spks is easy as they are contained in the transaction’s vout list, getting a list of input spks requires getting corresponding transactions via RPC calls to gettransaction, decoding them via RPC decoderawtransaction and reading their corresponding vouts. This complexity is caused by the fact that transaction does not contain input spks, but only input transaction ids.

### generateNewHistoryElement (Internal)

Converts the following data:

* number of confirmations
* transaction id
* block hash
* decoded transaction structure

into a history element containing:

* transaction id
* block height
* fee

As transaction id is give, only new values are block height and fee. If number of confirmations is not equal zero, we can read height via RPC call getblockheader (since we have block hash). If number of confirmations is zero, we need to read all unspent input UTXOs. If any of the unconfirmed, we enter height -1 in history element, otherwise we enter 0. We also calculate fee, but only in case number of confirmations is 0.

TODO: explain why we only set fee when number of confirmations is zero, is HistoryElement.fee used at all? We pass it to blockchain.scripthash.get\_history but not sure it is using it.

### checkConfirmations (Internal)

### checkForReorganizations (Internal)

## Modifications in bitcoin-s

* Changes for heartbeat in a thread that is waiting on the socket, heartbeat callback to allow hooking up periodic checks
* changes allowing direct in-to-socket-channel response

# Issues

* Lack of TOR support
* Better logging, logging categories of output to separate files
* get\_fee\_hitstogram has not been tested when enabled
* all TODOs in code

# Future extensions

* Lightning network support
* JMX or some other kind of UI
* Functionality around labels

# Private (Remove when open sourcing)

## Check Lock Time Verify

December 2015, BIP-65

CHECKLOCKTIMEVERIFY (CLTV)

CLTV is per output lock, it is not a transaction lock

Restricts UTXO so that it can be spent in the future transaction with nLocktime set to a greater or equal value.

In order to lock an output with CLTV, you insert it into the redeem script of the output in the transaction that creates the output.

<now + say 3 months>

CHECKLOCKTIMEVERIFY

DROP

DUP

HASH160

<Recipient’s Public Key Hash>

EQUALVERIFY

CHECKSIG

When recipient tries to spend this UTXO, she constructs a transaction that references the UTXO as an input. She uses her signature and public key in the unlocking script of that input and sets the transaction nLocktime to be equal or greater to the timelock in CTLV above.

Recipient then broadcasts the transaction on the bitcoin network.

The transaction is evaluated as follows:

CLTV halts and marks the transaction invalid if:

* Stack is empty
* Top stack item is < 0
* Top stack item is height and nLocktime is timestamp or vice versa
* Top stack item > nLocktime
* nSequence is 0xffffffff

Otherwise CLTV acts as NOP.

## Functionality around CLTV

When EPSMI accepts transaction from Electrum to be broadcast, it needs to modify this transaction so that its output (one or more) is time locked.

In another scenario, EPSMI modifies nLocktime of unlocking transaction, so that it is possible to unlock the funds.

<https://freedomnode.com/blog/easy-method-to-pass-your-bitcoins-to-your-family-members-when-you-die/>

coinb.in web-based wallet has a feature to time lock funds

<https://github.com/OutCast3k/coinbin/>

JMX would have to have a button:

* Hold broadcast

And a text:

* Transaction held for modification and future broadcasting

Then there are other buttons:

* Set nLocktime in held transaction (param is txid)
* Add CLTV guard to script in held transaction(param is locktime, txid)

First one is needed when unlocking the UTXO, second one is needed when locking.

Modified transaction will be shown on the screen, with some hints about the lock time or unlock time.

Yet another button will broadcast the above transaction.

This code needs to be extremely well tested as any mistake will cost money.

Suggestion is not to lock more than 5000 dollars in one UTXO in one transaction.

There should be some built-in check to limit the amount to be locked, to, say, 0.1BTC.