Overview

The Shakti wallet library provides access to the Shakti network through a simple C interface, compatible with most programming languages.

Concepts

A *wallet* is just a small string of bytes, generated randomly. These bytes are used to generate all of a user's addresses and keys, so they must be kept secret – store them in secure storage if your platform has that capability, and never put them into a log or send them outside the device for any reason.

A *session* is a connection to a Shakti server, and uses a wallet to access the user's addresses and authorize transfers. Sessions also hold a separate *data cache*, which the library uses to make sessions more efficient and reduce the load on the servers.

A session is represented in the library by a `sessiontoken`, an opaque pointer value returned by the `createSession` function and disposed of by the `freeSession` function. This pointer is passed to most of the library functions.

A *request*, represented by a `reqtoken`, is generated by many of the functions. Requests are handled asynchronously; you generate a request, then later ask the library whether it's complete until you get a positive response. This design allows the library to run sessions in the background, simplifying its use and letting you make additional requests while the first one is being processed.

A *null-terminated string* is a standard string in the C programming language. It consists of a pointer to any number of characters with values between 1 and 255 inclusive, followed by a single character with the value of zero (a "null character"). Many functions return such string values.

Strings in this library will usually consist of nothing but characters with values between 1 and 127 inclusive (the ASCII character set), plus the terminating null character. However, strings that come from outside the library may include non-ASCII characters. To deal with these, all strings must be treated as multi-byte UTF-8 strings. See the Wikipedia UTF-8 article if you're not familiar with the encoding.

JSON refers to *JavaScript Object Notation*, a standardized and language-independent format for data exchange that can be read by both humans and software. It's used by some of the functions. See the Wikipedia article or json.org if you're not familiar with it.

A `NULL`, also known as a *null pointer* and not to be confused with a JSON `null` or the Unicode codepoint zero that ends null-terminated strings, is a pointer with a known value that is not valid for any other use (almost always zero). Functions that return null-terminated strings, `sessiontoken` values, or other pointers will generally return `NULL` if there's an error that prevents them from completing the requested operation.

Although the value of a Shakti account is measured in ShaktiCoin, this library (and the internals of the system) all operate on either "chai" (1/100th of a coin) or "toshi" (1/100,000th of a coin, and the smallest unit of measure in the system). If a denomination isn't clear from the function or variable name, it will be noted in the description.

The Shakti network consists of both the main network (generally referred to as "mainnet") and the testnet. The testnet is generally identical to the main network and has identical capabilities, but the coins on it are not worth anything except for testing compatible software. The Shakti Foundation reserves the right to reset all testnet accounts to zero at any time without warning, or on a regular basis. The testnet may also be rate-limited if warranted.

A *wallet ID* is the first mainnet pocket address that a particular set of wallet bytes produces. It's often used by convention to identify the wallet. You can retrieve it with the call `getAddress(sessiontoken, 0, 0)`, after initializing the library with the wallet bytes.

How to Use It

When your program uses the library for the first time, it will call the `createNewWallet` function to initialize a new wallet, then pass the data that it returns to the `createSession` function. After the initial `createNewWallet` call, that function won't be needed again for the same user.

The library will initiate a connection to a nearby server as soon as you issue the first request that requires one, and maintain it until the connection is broken or you destroy the session handle.

When you issue a request that requires communications with a server, the library will return a `reqtoken` while it talks to the server. You use the `getRequestStatus` and `getRequestResult` functions to ask the library what's going on with it, at your leisure. Then dispose of it with `freeRequest` when you're through with it. You technically don't *have* to call `freeRequest` on request tokens, but if you don't they'll continue to take up memory.

When you're ready to disconnect from the server, call `freeSession` to disconnect and free all resources relating to it (including any `reqtoken` items that you haven't yet freed).

We have provided the source code for a program that uses many of the library's functions, `sxe-wallet`, to demonstrate how they can be implemented.

Advanced Topic: Multithreading

If you're writing a multithreaded program, note that the `sessiontoken` and `reqtoken` items should NOT be considered thread-safe, nor are the stand-alone functions that use a static return buffer at present. You CAN create several sessions if needed, and you can safely access any handle or function from any thread so long as that handle/function isn't used in another thread at the same time.

The API Functions

`calculateTransferFee`

```
const char* calculateTransferFee(const char *recipientReceivesInToshi);
```

Calculates the transfer fee charged by the network for a transfer of the specified size.

The parameter is the amount that the recipient will receive, represented as the string representation of an integer and measured in toshi. The return value is provided in the same format. Note that if the value is too large to be represented internally, or smaller than the minimum transfer value, or can't be interpreted as a numeric value, the function will return `NULL` instead.

See also `separateTransferAndFee`.

`createNewWallet`

```
const char* createNewWallet(const char *reserved,
    const char *passphrase);
```

Creates a new wallet and returns its bytes, encrypted to the provided `passphrase`. On error it will return `NULL`. NOTE: at present, if you create a wallet through this function, the mainnet will not recognize its addresses (though the testnet will). You must go through the Shakti Foundation's systems to get an official wallet to use with the mainnet.

The `reserved` parameter is not presently used and should be `NULL`.

The `passphrase` parameter can be any null-terminated string, or `NULL`. A `NULL` passphrase is treated the same as an empty string, and a default passphrase is used.

The returned value will remain valid until the next call to `createNewWallet`.

Note that each user can have only one wallet. This will eventually be enforced by the network, though it isn't yet.

`createSession`

```
sessiontoken createSession(const char *cacheBytes,
    const char *walletBytes,
    const char *passphrase);
```

Creates a new session, represented by the returned `sessiontoken`. On error, the returned value will be `NULL`; there's no way to retrieve the cause of such an error, but you can generally assume that it means that the passphrase is not valid for the provided `walletBytes` parameter. (See also `verifyWalletBytes`.)

You shouldn't need to create more than one session at a time, since you can issue multiple overlapping requests to a single session.

The `cacheBytes` parameter should be the value returned by `getCacheBytes` from the previous session, or `NULL` or an empty string if this is the first session or there are no cache bytes available for some reason. You can also pass `NULL` or an empty string for this parameter if you want to refresh the cache, which may be desirable if the user significantly changes his location (so that it gets a new set of servers for the new location, rather than using the ones from the previous location).

The `walletBytes` parameter will be either the value returned from the `getWalletBytes` function, or the value returned from `createNewWallet`.

The `passphrase` parameter must be the same as that provided to `getWalletBytes` or `createNewWallet`.

You should call `freeSession` when you're through with the `sessiontoken`, to shut down communications to the server and free all resources associated with the session.

`freeRequest`

```
void freeRequest(reqtoken t);
```

Frees the resources used for a request. Note that all remaining `reqtoken` items are freed by a call to `freeSession`, you don't need to free them separately if you're freeing the session as well.

`freeSession`

```
void freeSession(sessiontoken t);
```

Shuts down any communications with the server and frees the storage backing the provided `sessiontoken`. All pending requests are canceled, though they might continue on the server. Call this when you're finished with a `sessiontoken` to ensure that the operating system frees all resources associated with it. Once you've made this call, you cannot use the `sessiontoken` anymore, or any `reqtoken` items that were generated from it.

`getAddress`

```
const char* getAddress(sessiontoken t,
    unsigned int addressNumber,
    unsigned int network);
```

Returns one of the user's addresses, or `NULL` if there is an error. The returned value will remain valid until the next call to `getAddress` for the given `sessiontoken`.

Every wallet can represent any number of addresses. Users may elect to create addresses beyond the main one at will, for whatever purposes they choose.

The `addressNumber` parameter is an unsigned integer representing the address number to return. Address zero is always the user's main address for the requested network.

The `network` parameter will generally be `0` (zero), to retrieve an address on the main network. It can also be `1` (one) to retrieve an address on the testnet. All other values are presently reserved.

Address zero on the mainnet is termed the *wallet ID*, and is often used by convention as an identifier for the entire wallet.

getBalance

```
reqtoken getBalance(sessiontoken t,
    const char *address);
```

Generates a request for the balance of any address, whether in the user's wallet or not.

After issuing this request, `getRequestStatus` will indicate the progress:

- -101: lost connection
- -1: the request could not be processed.
- 0: the request has not completed.
- 1: the request completed successfully.

Once complete, the balance will be returned (via `getRequestResult`) as a string-formatted integer representing the number of toshi in the account; divide by the toshi-per-coin value to get the value in ShaktiCoin. If the request fails, `getRequestResult` may return an (English) error message indicating the reason.

getBlockByIndex

```
reqtoken getBlockByIndex(sessiontoken t,
    unsigned int zone_network,
    unsigned int index,
    int includeTransactionDetails);
```

Generates a request for a specific block, identified by the network and the block's index number.

The `zone_network` parameter has two uses. The primary use is to indicate the currency network (`SHAKTI_MAINNET` or `SHAKTI_TESTNET`). The secondary use is to indicate a feat ledger zone; this will be zero for normal currency requests, which is what most users of this library will always want, or one of the `shaktiZone` values defined in `walletlib.h` to request information from a feat ledger. A feat ledger request should combine the ledger zone value with the network value using a binary `OR` operation, such as `(ZONE_EUROPE | SHAKTI_TESTNET)`. All values other than those produced by a combination of the `SHAKTI_*`and `ZONE_*` values in `walletlib.h` are presently reserved.

Blocks in the Shakti network start at zero (the "genesis block") and increase monotonically. The `index` parameter refers to this number. Blocks can also be retrieved by their identifiers; see the `getBlockByIdentifier` function. Note that the testnet will be reset regularly; the timestamp on the genesis block will change when that happens.

The `includeTransactionDetails` parameter is non-zero to request the details of all transactions in the block, or zero if you don't need those details.

After issuing this request, `getRequestStatus` will indicate the progress:

- -101: lost connection
- -1: the request could not be processed.
- 0: the request has not completed.
- 1: the request completed successfully.

Once complete, `getRequestResult` will either return `NULL` (if the requested block wasn't found) or the block information as a JSON "object" with the following data:

- `version`: the block's version identifier, presently `blkv1`.
- `network`: the network/currency (presently `testnet` or `mainnet`).
- `identifier`: this block's hash.
- `index`: this block's index.
- `timestamp`: this block's timestamp, in UNIX time (seconds since midnight of January first, 1970, UTC).
- `previousIdentifier`: previous block's hash, because as a blockchain, it's part of the data used for calculating the identifier.
- `nonce`: the nonce string for this block (used to ensure that the block's identifier is unique). Very early testnet blocks may have an empty nonce; nothing else should.
- `transactions`: an array containing the transaction hashes in the block.
- `transactionDetails` (only included if requested): a JSON array of objects containing the transaction data for each of the items in `transactions`. See `getTransaction` for the information included in this.

Note that in the Shakti system, nodes are allowed to have only a recent set of blocks and transactions, to let new nodes start participating in the network without downloading the entire history. As such, some nodes may be unaware of blocks and transactions prior to a particular point in time. We're working on a way to provide such information to users of this library.

getBlockByIdentifier

```
reqtoken getBlockByIdentifier(sessiontoken t,
    const char *identifier,
    int includeTransactionDetails);
```

The same as `getBlockByIndex`, but taking a block identifier instead of a network/currency and index. See that function for details. The identifier denotes both the network involved and the specific block.

getBlockByTime

```
reqtoken getBlockByTime(sessiontoken t,
    unsigned int zone_network,
    unsigned int timestamp,
    int includeTransactionDetails);
```

The same as `getBlockByIndex`, but taking a timestamp instead of an index. The system will return the block on the requested ledger and network that is equal to the requested timestamp, or the first block after that timestamp. If there is no such block, it will return the last block on the requested network.

The `timestamp` must be the number of seconds since midnight of January first, 1970 (the POSIX epoch).

getCacheBytes

```
const char* getCacheBytes(sessiontoken t);
```

Retrieves a null-terminated string of bytes that represent the session's persistent data cache, or `NULL` on any error. This data allows the library to connect to the servers more quickly in the future.

The cache may change during any session, so we recommend storing this data after every session.

getFoundationBytes

```
const char* getFoundationBytes(sessiontoken t);
```

This returns the raw wallet bytes, encrypted to the Shakti Foundation's current public key and an ephemeral private key. This function is intended solely for the Shakti Foundation's use, as the information it provides is useless to anyone who doesn't have access to the Shakti Foundation's private key.

`getLastBlock`

```
reqtoken getLastBlock(sessiontoken t,
    unsigned int network,
    int includeTransactionDetails);
```

The same as `getBlockByIndex`, but only taking a network/currency parameter, and returning the most recent block for that network. See that function for details.

`getRequestResult`

```
const char* getRequestResult(reqtoken t, unsigned int n);
```

This function will return any results from the given request. It will return results for the `n` value given, so long as `n` is less than the return value from `getRequestResultCount` for the same request. It will return `NULL` for any other value.

NOTE: most requests will only provide a single result string at most, unless otherwise noted in their descriptions.

`getRequestResultCount`

```
unsigned int getRequestResultCount(reqtoken t);
```

Returns the number of results available for the request. Most requests will only return zero or one result, but future calls may include more.

`getRequestStatus`

```
int getRequestStatus(reqtoken t);
```

Returns zero (meaning the request is not yet complete), a positive value (meaning the request has completed a step successfully), or a negative value (meaning that the request failed).

The actual values and their meanings depend on the request, but an error code of -101 always means that the library lost its connection to the server so the status of the request is unknown.

NOTE: when calling the `transfer` function (only!), you should get *two different* positive values for each successful transfer. The first will be a `1`, indicating that the transfer looks good to the node that you contacted and its group of validators; the second will be a `2`, indicating that it was included in the blockchain.

`getTransaction`

```
reqtoken getTransaction(sessiontoken t,
    const char *identifier);
```

Generates a request for the details of a specific transaction in the blockchain, identified by the transaction identifier.

After issuing this request, `getRequestStatus` will indicate the progress:

- -101: lost connection
- -53: the requested transaction wasn't found, but the contacted server's blockchain isn't completely up-to-date so this is not definitive.
- -1: the request could not be processed.
- 0: the request has not completed.
- 1: the request completed successfully.

On success, `getRequestResult` will either return `NULL` (if the requested transaction wasn't found in the blockchain) or the transaction information as a JSON "object" with one of the following sets of data, determined by the JSON object's `type` item:

- if `type` is `transfer` (which will usually be the case), then:
  - `version`: the transaction's version identifier, presently `txv1`.
  - `identifier`: the transaction hash.
  - `timestamp`: this transaction's timestamp, in UNIX time (seconds since midnight of January first, 1970, UTC).
  - `sourcePocket`: the source pocket ID.
  - `sourcePublicKey`: the source's base64 public key, needed for validation.
  - `targetPocket`: the target pocket ID.
  - `amountInToshi`: the transfer amount.
  - `feeInToshi`: the total network fee deducted from `amountInToshi` before it's deposited into the target account (zero at the time of this writing, this will be added in the near future).
  - `nonce`: the nonce string used (needed for checking the hash).
  - `transferSignature`: the transfer signature.
  - `verifiers`: an array of strings, indicating the verifying nodes that signed the transaction and their signatures. There is a single string per verifier, with the node's name, a space, the signature in base64 format, another space, and the node's two-character zone identifier.
  - `blockIndex`: the index number of the block that the transaction is part of. NOTE: this will be a JSON `null` if the transaction hasn't been included in a block yet, or on some early testnet transactions where the block information wasn't stored with the transaction.
- if `type` is `admininfo`, then:
  - `version`: the transaction's version identifier, presently `admv1`.
  - `identifier`: the transaction hash.
  - `timestamp`: this transaction's timestamp, in UNIX time (seconds since midnight of January first, 1970, UTC).
  - `nonce`: the nonce string used (needed for checking the hash).
  - `adminSignatureId`: the base64-encoded public key of the signing pocket, needed for validation.
  - `adminSignature`: the signature for the admininfo change record, created by the private key corresponding to `adminSignatureId`.
  - `changes`: one or more admininfo change commands.
  - `blockIndex`: the index number of the block that the transaction is part of. NOTE: this will be a JSON `null` if the transaction hasn't been included in a block yet.

Note that in the Shakti system, nodes are allowed to have only a recent set of blocks and transactions, to let new nodes start participating in the network without downloading the entire history. As such, some nodes may be unaware of blocks and transactions prior to a particular point in time. We're working on a way to provide such information to users of this library.

getWalletBalance

```
reqtoken getWalletBalance(sessiontoken t,
    unsigned int addressNumber,
    unsigned int network);
```

Generates a request for the balance of an address in the user's wallet.

This is a convenience function for requesting the balances of pockets belonging to the wallet that the library was initialized with. It combines the functions of `getAddress` and `getBalance`, so you can make only a single call for each pocket instead of two separate ones.

Refer to the `getAddress` function for details on the `addressNumber` and `network` parameters. Refer to `getBalance` to interpret the returned status.

getWalletBytes

```
const char* getWalletBytes(sessiontoken t,
    const char *passphrase);
```

Retrieves a null-terminated string of bytes that represent this wallet, or `NULL` on error. The returned value will remain valid until `getWalletBytes` is called again on the given `sessiontoken`, or `freeSession` is called on it.

The returned data should be treated as an opaque value; the only guarantee is that it will be a null-terminated string.

The `passphrase` parameter can be any null-terminated string, or `NULL`. A `NULL` passphrase is treated the same as an empty string, and a default passphrase is used.

The user's wallet presently changes only when a new address is created.

separateTransferAndFee

```
const char* separateTransferAndFee(const char *totalInToshi);
```

Splits the provided total into the amount that will be transferred and the network fee for transferring that amount, using the same code that the network does. This function is useful if the user wishes to transfer *all* of the coin in a pocket, or if he wants a specific amount to remain in the pocket.

The parameter must be provided as the string representation of an integer, measured in toshi. The returned value will be the string representations of the amount that the recipient will receive and the fee required, separated by a comma. The function will return `NULL` if there's any problem.

See also `calculateTransferFee`.

submitAdminInfo

```
reqtoken submitAdminInfo(sessiontoken t,
    unsigned int network,
    const char *commandDeclarations);
```

Submits one or more administrative commands for the network. You can only use this if you have a signing key that is recognized by the network (so it's essentially limited to the Shakti Foundation). The parameters are: * `network`: the network (from the `shaktiNetwork` enum in the header file) to submit the administrative commands to. * `commandDeclarations`: the commands, one per line, to submit. Case is significant in the commands. The commands currently recognized are: * `NEWPOCKET <pocketId> [<flag>...]` * `ADDFLAG <pocketId> <flagToAdd> <authorizationCode>` * `REMOVEFLAG <pocketId> <flagToRemove> <authorizationCode>`

submitPoeFeats

```
reqtoken submitPoeFeats(sessiontoken t,
    unsigned int zone_and_network,
    const char *featDeclarations);
```

Submits a set of feat declarations. You can only use this if you have a signing key that is recognized by the network (generally limited to school officials and related authorities). The parameters are: * `zone_and_network`: the ledger zone (from the `shaktiZone` enum in the header file) and network (from the `shaktiNetwork` enum in the header file) to submit the feats for, combined with a binary OR operator. This must be the zone and network that the feats are intended for; this will be checked on the Shakti servers. Example: `(ZONE_MIDDLEEAST | SHAKTI_TESTNET)`. * `featDeclarations`: one or more feat declarations. The format of these has not been finalized at the time of this writing, but will presumably be a set of data in JSON format.

After issuing this request, `getRequestStatus` will indicate the progress. See the `transfer` command for details, though error responses will usually be "reason unknown."

submitTransfer

```
reqtoken submitTransfer(sessiontoken t,
    unsigned int addressNumber,
    const char *toAddress,
    const char *valueInToshi,
    const char *feeInToshi,
    const char *memo);
```

Transfers value from the `addressNumber` (on the network that `toAddress` belongs to) to the `toAddress` provided. The parameters are: * `addressNumber`: the index of the pocket in this wallet that the funds will come from. Refer to the `getAddress` function for details. * `toAddress`: the address of the pocket that the funds will be transferred to. * `valueInToshi`: as the name suggests, this is the amount that the recipient will receive, represented as the string representation of an integer and measured in toshi. * `feeInToshi`: this must be the value returned by the `calculateTransferFee` function, or `NULL` to let the library calculate the value itself. If this parameter is any other value, the transfer will be rejected without being sent to the network and the function will return a `NULL`. * `memo`: this parameter is not used at present; please pass an empty string to it (not `NULL`).

After issuing this request, `getRequestStatus` will indicate the progress:

- -101: lost connection, transfer status unknown
- -52: transfer failed, not acknowledged, you can try again
  - NOTE: this means that the server did not acknowledge receipt of a transaction request within the allotted time. You can resubmit the transfer immediately if desired.
- -51: transfer failed, transient error, you can try again
  - NOTE: this generally means that the transaction looked okay, but a temporary problem with the server nodes prevented it from being approved. We recommend starting a new session before resubmitting the transfer, as resubmitting the transaction to the same node may have the same result for at least a little while.
- -11: transfer failed, fee information is incorrect
- -10: transfer failed, bad or duplicate identifier
- -9: transfer failed, future timestamp
- -8: transfer failed, repeated timestamp
- -7: transfer failed, insufficient funds
- -6: transfer failed, bad transfer signature
- -5: transfer failed, less than minimum
- -4: transfer failed, currency mismatch
- -3: transfer failed, sent to sending address
- -2: transfer failed, sent to faucet
- -1: general failure, transfer status unknown
- 0: transfer still being processed
- 1: transfer approved and should be included in the next block
  - NOTE: for the `transfer` function, this value is NOT the final one. You will always get a second value, either `2` to indicate that the transaction was successfully included in a block, or a negative value to indicate that it failed.
- 2: transfer included in block

`getRequestResult` will return the transaction ID at any point. Note that the transaction ID does not guarantee that the transaction went through, it only identifies the transaction.

Note that the reason given is only the reason that the recipient node has for rejecting the transfer. If the recipient node doesn't see a problem with it, but the rest of the network does, you'll get a "reason unknown" error.

submitMultiTransfer

```
reqtoken submitMultiTransfer(sessiontoken t,
    unsigned int timeoutSeconds,
    unsigned int addressNumber,
    struct multitransfer_data *data,
    unsigned int datacount);
```

Sends a batch of one or more transfers from the `addressNumber` (on the network that each `toAddress` belongs to). The fee for each transfer will be calculated by the library. Batched transfers have a lower priority than ones made with `submitTransfer`, but the servers can handle large numbers of transfers with this one, whereas each server has only a limited queue for `submitTransfer` requests.

The parameters are: * `timeoutSeconds`: this can be any value up to 3600 (which is one hour). A value greater than 3600 will cause the entire batch to be rejected immediately. * `addressNumber`: the index of the pocket in this wallet that the funds will come from. Refer to the `getAddress` function for details. All of the transfers submitted as a single batch will come from the same pocket. * `data`: an array of one or more `multitransfer_data` structures, which contain the null-terminated strings `toAddress`, `valueInToshi`, and `memo`. These must be initialized with the data of the same names given in the `submitTransfer` function description. * `datacount`: the number of `multitransfer_data` items in `data`.

The only limit to the number of transfers you can request in a single call is the number that the server can process before the `timeoutSeconds` pass. This varies unpredictably depending on the server's capacity and current load.

After issuing this request, `getRequestStatus` will indicate the progress. See `submitTransfer` for details; the only difference is that this function will only return one non-zero value (a `1` for success, or a negative for failure), it won't return a second one to indicate that the transactions made it into the blockchain.

The `multitransfer_data` structure contains an `identifier` field as well. It doesn't matter what this is set to when you call the `submitMultiTransfer` function, but when it exits, each `identifier` field will be set to the identifier for that transfer.

To confirm that the transfers succeeded, wait two minutes (120 seconds) or more after the `timeoutSeconds` pass, then call `getTransaction` for each of the identifiers provided. If it returns the transfer transaction, then that transfer was successful; if it does not, then it failed for some reason. Successful transfers may be identifiable earlier, but there's no certain way to tell the difference between a failed transfer and one that simply hasn't completed until two minutes after the `timeoutSeconds` pass.

There is no way to identify why batched transactions failed, but if there is no problem with the data provided and the source pocket has sufficient coin to cover the transfer and associated fee, then it's likely that the server or network could not process all of the transfers within the time you requested, or that there was a transient error.

transfer

```
reqtoken transfer(sessiontoken t,
    unsigned int addressNumber,
    const char *toAddress,
    const char *valueInToshi,
    const char *feeInToshi,
    const char *memo);
```

This name is deprecated in favor of the otherwise-identical `submitTransfer` function.

validateRequestToken

```
int validateRequestToken(reqtoken t);
```

Returns a non-zero value if the token passed to the function is valid, or zero if it isn't.

You shouldn't need call this function. It's only here for the very few cases where the caller can't easily track whether a token is valid or not.

`validateSessionToken`

```
int validateSessionToken(sessiontoken t);
```

Returns a non-zero value if the token passed to the function is valid, or zero if it isn't.

You shouldn't need call this function. It's only here for the very few cases where the caller can't easily track whether a token is valid or not.

`verifyWalletBytes`

```
int verifyWalletBytes(const char *walletBytes,
    const char *passphrase);
```

Returns a non-zero value if the provided `walletBytes` and `passphrase` strings identify a valid set of wallet bytes.

This function generally isn't needed. It's here solely to provide a way to validate a wallet's validity and passphrase without initiating a session, which should rarely be necessary.

`walletLibraryVersion`

```
const char* walletLibraryVersion();
```

Returns the version identifier, which will be a string containing a date in the form `YYYY-MM-DD`, possibly followed by a period and two or more decimal digits identifying the second and subsequent versions made on the same day.

TODO:

I (the library's author) don't like how the system handles error messages, it's not very flexible and it makes complete i18n very difficult. If I can come up with a better solution before the official release, the library's design will change.

The usability of the current `sxe-wallet` example program is very limited as well – there's no way to ask it to generate a new address, among other things. Some of that will require server changes.

There should be an option to provide a callback function for requests as well, so the calling code doesn't have to keep polling for results.