Freimarkets: extending bitcoin protocol with user-specified bearer instruments, peer-to-peer exchange, off-chain accounting, auctions, derivatives and transitive transactions

Version v0.0.1

Mark Friedenbach, Jorge Timón

August 24, 2013

Abstract

This proposal adds primitives to bitcoin necessary for implementing non-currency financial constructs, such as dividend-yielding bonds, asset ownership tokens, credit relationships, a variety of forms of smart contracts, and distributed marketplaces for exchanging all of the above. Private accounting servers provide a mechanism to support unlimited volume of off-chain transactions while being able to interact with in-chain assets through atomic cross-chain trade and an integrated peer-to-peer market.

Keywords: bitcoin, freicoin, user-specified bearer instruments, off-chain accounting, atomic trades, auctions, derivatives, transitive transactions

Contents

1 Introduction

Herein we propose a new transaction format, `nVersion=3` transactions, which enables hierarchies of independently verified sub-transactions, additional validation scripts and introspective opcodes, strict currency controls, as well as relaxation of the rules regarding coin generation via coinbase transactions for the purpose of supporting user-defined assets on the block-chain. We also introduce the concept of private centralized accounting servers to perform transactions of off-chain assets that cam interact with each other as well as with in-chain assets. Combined with suitable extensions to the peer-to-peer, JSON-RPC, RESTful, and wallet interfaces, these protocol changes complete bitcoin's repertoire of low-level constructs, allowing the emulation of a wide variety of financial instruments.

Together this enables the following sorts of applications:

- Issuing new assets by means of asset definition transactions (coinbase transactions other than the usual first transaction of a block). Such assets are allowed to specify their own interest/demurrage rate, unit granularity, display scale, and a hash field referencing an external resource, possibly a legal or Ricardian contract that the chain itself does not validate.
- Issuing unique and indivisible tokens that are transferred in sets instead of numeric amount, and allow fast look ups on their current ownership to enhance smart property use cases and manage some permissions of the regular custom assets.
- Atomic exchange of assets of differing types through inclusion of inputs and outputs of both types in a single transaction.
- Signing orders (partial-transactions giving up one asset in exchange for another) that are binding but not completed until they get into the chain as part of a balanced transaction, and have attached expiration dates or can be explicitly cancelled by double-spending the signed inputs.
- Executing an arbitrary number of these orders atomically by creating a complete valid transaction where the orders are included as nested *sub-transactions*, thereby executing an atomic trade without requiring each of the parties to be online or in direct communication with each other. Composing orders from separate markets into an atomic trade with intermediate assets enables payments based on transitive trust relationships.
- Destruction of coins, tokens, or assets when no longer needed by a special class of non-spendable, prunable output script.
- Restricting the conditions by which a transaction or sub-transaction may be selected for inclusion by specifying *validation scripts*, which are run when the enclosing block is validated. Introspection of the block chain from within the bitcoin scripting environment is enabled by the introduction of new opcodes.
- Running accounting servers as private chains with centralized rather than distributed consensus, in which off-chain assets can be issued, transferred and traded in the same way they are in the public chain, with the private block chain providing an audit log.
- Execute an arbitrary number of trades from different accounting servers and/or the public

chain in an atomic transaction, using either the public chain or an agreed upon timestamping service for the commit phase.

- Public chains or private accounting servers configured to "observe" other chains to enable much faster but secure cross-chain trade, compared with the existing slow, multi-phase protocols involving revelation of hashed secrets. This requires the ability to extract proofs from the observed chain in order to validate conditional transactions.
- Restrict the usage of a custom asset by assigning to it rotatable signing keys which must sign all transactions involving the restricted assets prior to inclusion (support for KYC regulatory compliance).

In the remainder of this document we first overview the major features of this proposal, then detail the precise modifications and formal specifications before covering a handful of applications.

2 Major features

Here we describe in non-normative prose the proposed changes. We assume technical familiarity with the bitcoin protocol and Freicoin's various extensions thereof.

2.1 Indivisible, unique tokens

Indivisible, uniquely identifiable asset tokens are useful for applications like physical ownership - keys to a smart car, numbered seats or membership tokens. Since a token is contained within no more than one unspent transaction output at any given time, it's not necessary to trace the ownership back to a "genesis transaction" as one would need to do with a colored coins approach [1]. This allows smart property clients to have smaller requirements when authenticated unspent transaction output index checkpoints are included in each block [2].

2.2 User-issued assets

Divisible currency and/or tokens representing user-issued assets may be minted in special coinbase transactions separate from the usual first transaction of a block (where freicoins are currently, and continue to be minted). Coins created in such generating transactions are not freicoins, but rather user-issued asset shares which represent fungible ownership of the underlying asset type, or asset tokens identified by per-asset unique bitstrings. Such coins or tokens can be included in transactions containing regular Freicoin currency, which in this document is sometimes called the host currency or fee currency.

The creator of the new asset can define an interest/demurrage rate. The quantity issued may be fixed or he may define a list of issuance tokens that permit their owners issue new units of the asset being defined.

The creator of the asset definition transaction may also specify a list of authorizer tokens. The signature of an authorizer is required every time a transaction involves inputs or outputs of that asset. This allows issuers/gateways to manage closed list of "authorized accounts" of registered users if regulatory restrictions of their jurisdiction requires them to do so [3] or if they desire whitelisting of participants (for example, local currencies or restricted stock sales). It also allows issuers to charge fees when the assets are traded or moved.

[1] Colored coins approach to custom assets in the chain is to define a genesis transaction that identifies the asset and trace the funds in that transaction outside of the chain to treat them differently as they represent more than regular bitcoins. This approach has several limitations when compared to this protocol extension. There's a discussion group on colored coins development here: https://groups.google.com/forum/#!forum/bitcoinx

[2] Having a fast access UTXO tree indexed in each block would enhance light clients security and it is also important for scalability, something important for this proposal since it enables new uses and a bigger volume is to be expected. Mark Friedenbach's work on these improvements is documented here: http://utxo.tumblr.com/

[3] Issuers of currencies convertible to fiat may have to comply with know your customer regulations in their jurisdiction for anti-money laundering enforcement. For example, U.S. dollar gateways based in the USA need to comply with FinCEN's normative.

Using unique tokens to manage new issuance and authorizers allows the creator to follow his own key cycling policy or security protocols. By utilizing multisig or multiple signatures, it is possible for transactions to remain valid even across one or more key rotations.

These various properties of the asset, its interest/demurrage rate, unit granularity and display scale, and listings of issuer and authorizer tokens are set in the coinbase string of the asset definition transaction.

2.3 Partial transactions

This proposal extends the transaction format with an optionally empty nested level of sub-transactions. Sub-transactions differ from regular, block-level transactions in that their inputs and outputs are not required to balance and they have associated with them a quantity and granularity allowing for fractional redemption.

Since validation of sub-transactions occurs separately from each other and the higher-level enclosing transaction, pre-signed, unbalanced transactions are able to act as offers on a distributed exchange: market participants sign offers adding coins of one asset type in exchange for an output of another type. These signed offers are broadcast through a side-channel and aggregated by miners. When a cross-over is detected (a bid higher than an ask), the miner combines the two pre-signed offers and claims the difference as a fee.

Other use cases are enabled. For example, when the underlying assets represent lines of credit, the exchange mechanism allows payments based on transitive trust relationships, in the style of the original Ripplepay application by Ryan Fugger.

2.4 Private ledgers

Private accounting servers, or "accountants" use a variant of the Freicoin/Freimarkets code base that is stripped of the distributed consensus proof-of-work mechanism. Accountants are responsible for eliminating double-spending, reserving balances for pending transfers, and authorizing transactions, sometimes conditionally on external events. Accountants are able to prevent transactions from going through if the owner has already obligated funds elsewhere, by keeping track of the available balance (actual balance minus funds in various stages of commit). Accountants use various distributed consensus mechanisms for coordinating the transaction commitment with other private accounting servers or public block chains.

The level of privacy may vary from one server to another. Server operators are allowed freedom in choosing which parts of the block chain audit log to publish, with a sensible default being the block headers and coinbase transactions, allowing for validation of authenticated inclusion and index proofs used to notify users of their wallet balance, history and current activity, but not revealing other user's balances or transaction history.

By using newly added extrospective opcodes to construct scripts dependent on external chains, it is possible for private transactions to be conditional on public Freicoin blockchain data or other private accounting servers.

Note that the opposite relation cannot apply at this time. Public chains could support transactions conditional to data on other chains to enhance cross-chain trade, but then the observing chain's validation becomes dependent on the observed chain validation. This approach to cross-chain has been described several times [4], and would be trivial to implement with this protocol extension.

[4] At least these two threads describe this cross-chain trade scheme:
https://bitcointalk.org/index.php?topic=31643.0
https://bitcointalk.org/index.php?topic=91843.0

3 Proposed changes

3.1 Precision, rounding, and limits

All internal computation of accounting quantities are performed using arbitrary precision fractions, or an equivalent mathematical system which does not suffer from loss of precision or over/underflow.

3.1.1 `MAX_MONEY` / MoneyRange limitation

The maximum numerical value allowed for any output or stored intermediary value of any asset type is 2^{53} - 1 kria, or 9.007199254740991 x 10^{384}. This is about 10% less than the maximum value representable in the `decimal64` type. A transaction which violates this constraint is invalid.

3.1.2 IEEE 754-2008 decimal floating point

Output amounts for `nVersion=3` transactions are positive, real decimal floating point values using a stricter subset of the binary integer decimal encoding specified by IEEE 754-2008. Infinities and not-a-numbers are not allowed, and the normal (lowest exponent) representation must be used. For `nVersion=1` and `nVersion=2` transactions, the `int64 nValue` field is interpreted according to the following equation:

```
nValue :: int64
dValue :: decimal64
dValue = nValue * 10^369
```

That is to say, an old-style minimum representable positive value of 1 kria (0.00000001 freicoins) would be encoded as a new-style `decimal64` value of 10^{369}. Since the smallest representable positive `decimal64` value is 10^{-398}, that gives an expressive range of approximately 768 orders of magnitude in the exponent, plus sixteen digits of precision. While not technically providing infinite divisibility, this leaves plenty of room at the bottom.

3.1.3 A note on units

Throughout this document a couple of differing units are used for describing financial quantities on the Freicoin block chain. This unfortunate and confusing situation arises from the history of representing bitcoin/freicoin amounts both in user interface and serialization formats.

When talking about the host currency we speak of freicoins, with 1 freicoin (1 frc) traditionally being specified with 8 decimal places of precision. In old-style transactions the smallest non-zero representable unit of freicoins is 1 kria, with 100,000,000 kria = 1 frc. New-style transactions encode freicoin quantities as `decimal64` values, with 1 kria = 10^{369}, and the smallest non-zero representable unit being 10^{-398} or 10^{-775} frc. In either case, it is expected that the user interface will be configured to show units of freicoins, or the multiplier of the underlying asset.

3.2 Indivisible tokens

New-style outputs contain the `decimal64` continuous value combined with a possibly empty list of bitstrings. These bitstrings are indivisible, unique outputs. Any output token must be found in an input of the enclosing transaction, and tokens cannot be shared among two outputs of the same transaction.

The asset definition transaction, or any transaction with one or more of an asset's issuers as signatory is allowed to violate the constraint that continuous outputs are less than or equal to inputs, and that output unique tokens are a subset of inputs for that asset.

3.3 Asset tags

New-style outputs are tagged with a 160 bits identifying the asset from which the output is drawn. This tag is the 20-byte serialized hash (`ripemd160 . sha256`) of the asset definition transaction. For outputs of the host currency Freicoin, the similarly-calculated 20-byte hash of the genesis block is used instead.

3.4 Granular outputs

The granularity option of the asset definition determines the minimum increment which may be used to transfer an otherwise continuous value. It is represented as a positive `decimal64` value. If left unspecified, an asset is limited at this time to the minimum encodable positive `decimal64` value (10^{-398}), but with further subdivision allowed if future extensions enable it. The host currency Freicoin is defined to be maximally divisible in this way. For assets with non-zero interest/demurrage, granularity checks are made at the reference-height of the transaction.

3.5 Granular redemption

In general, outputs are considered spent only when the full amount has been claimed. If a later transaction claims less than the full amount, that amount is subtracted from the remaining balance.

A transaction may claim less than the full amount by utilizing a granular offer. The signed offer contains a 64-bit integer field `nGranularity` which specifies the number of equal-sized units the offer is split into, and any transaction making use of the offer may choose the number of units to claim, so long as there remains sufficient output remaining.

In order to implement this functionality, the set of unspent transaction outputs must include a field recording the amount remaining (or equivalently, the amount spent so far).

Fractional redemption of outputs containing unique tokens is not allowed.

3.6 Validation scripts

New-style transactions have a validation script, split into the two fields `scriptValidPubKey` and `scriptValidSig`, which when combined and executed must run to completion without abnormal termination, and return a non-zero value on the stack for a transaction to be valid.

While performing signature operations in any other script, the `scriptValidSig` is set to the empty script before performing hash serialization and the `scriptValidPubKey` is stripped of any code prior to (and including) the last `DELEGATION_SEPARATOR`, if one exists.

As a special case, an empty `scriptValidPubKey` and `scriptValidSig` automatically passes, and for an old-style `nVersion=1` or `nVersion=2` transaction, the empty script is the value of these fields.

3.7 Authorizing signatories

New-style transactions have a sorted list of `<assetid:token, scriptSig>` signatories. The `assetid` is the 20-byte asset tag, with a token bitstring taking the remaining bytes. The `scriptPubKey` is retrieved from the current unspent transaction output containing the identified authorizing token.

3.8 New scripting opcodes

Several new scripting language opcodes are added by this proposal. Their behavior are detailed here.

3.8.1 `BLOCK_HEIGHT` and `BLOCK_TIME`

These opcodes push the height of the block containing the current frame, or its `nTime` value onto the stack.

3.8.2 `DELEGATION_SEPARATOR`

The `DELEGATION_SEPARATOR` opcode is a `NOP` during execution, but does affect signature hash operations. During such serialization for any script *except* the one being executed, all code prior to and including the `DELEGATION_SEPARATOR` is omitted.

3.8.3 `QUANTITY`

The new `QUANTITY` opcode pushes the `nQuantity` value of the current frame onto the stack, or 1 if the current frame is a block-level transaction.

3.8.4 Extrospection opcodes

The following opcodes assume the maintenance of a discrete set of observed chains by each chain. If a public chain observes another public chain, it's validation and security become completely dependent on the observed chain, and any reorg on the later can trigger another reorg on the former.

Even assuming that a public chain only observes its own chain, the opcodes may require full nodes to have more data than it's currently on the utxo set, opening the door to new DoS attacks vectors.

For these reasons the opcodes are only recommended to be used in private chains, and even in those cases configure them with caution, potentially limiting more strictly the standard behavior described here. For example, in Freicoin their behavior is modified as described in section 3.8.4.4.

3.8.4.1 `OUTPUT_SPENT`

`<tx_id> <output number> <height> <chain-id> OUTPUT_SPENT`

Throws an error (abnormally terminating script execution) if the following condition is true:

- The chain identified by <chain-id> (the hash of the chain's genesis block) is not part of the set of chains observed by the chain for which the script is being validated.

Returns 0 if the following condition is true:

- The output identified by $<tx_{id}>$:<output number> still exists in the UTXO of <chain-id> at block height <height>.

Returns 1 otherwise.

3.8.4.2 `OUTPUT_SPENT_IN`

`<tx_id> <output number> OUTPUT_SPENT_IN`

It is almost equivalent to the following script:

`<tx_id> <output number> BLOCK_HEIGHT FRC_CHAIN_ID OUTPUT_SPENT`

The difference being that during execution the unspent transaction output set is the result of applying all transactions in the block chain prior to the one being validated, including transactions in the current block which precede the transaction being validated, but excluding transactions which come later.

3.8.4.3 `OUTPUT_EXISTS`

```
<refheight> <amount> <tokens> <scriptPubKey>
<chain-id> <asset-id> <from> <to> OUTPUT_EXISTS
```

Throws an error (abnormally terminating script execution) if any of the following conditions are true:

- The chain identified by <chain-id> (the hash of the chain's genesis block) is not part of the set of chains observed by the chain for which the script is being validated.
- <tokens> is not a serialized, sorted, non-repeating but possibly empty list of bitstrings.

Returns 0 if the following condition is true:

- There's no unspent output in <chain-id> from block height <from> to block height <to> (both included) of the specified asset and contract script, with an output amount greater than or equal to <amount> at reference-height <refheight>, and a set of output tokens which are a superset of <tokens>.

Returns 1 otherwise.

3.8.4.3.1 `OUTPUT_EXISTS_IN`

```
<refheight> <amount> <tokens> <scriptPubKey> <asset-id> OUTPUT_EXISTS_IN
```

It is almost equivalent to the following script:

```
<refheight> <amount> <tokens> <scriptPubKey>
FRC_CHAIN_ID <asset-id> BLOCK_HEIGHT BLOCK_HEIGHT OUTPUT_EXISTS
```

Again this will only make nodes look for outputs that are in the utxo at the moment of validation (see `OUTPUT_SPENT_IN` above for a more detailed explanation).

3.8.4.4 Freicoin's treatment of extrospection opcodes

In Freicoin, the only observed chain is Freicoin itself. The depth of the introspection is restricted too. So only the more limited `OUTPUT_SPENT_IN` and `OUTPUT_EXISTS_IN` opcodes are available. Any use of the generic ones will result in abnormal termination of the script.

3.9 Transaction expiration

`nExpireTime` works in a very similar way than `nLockTime`, mandating in this case a maximum time (also specified in either unix time or block height), after which the transaction cannot be accepted into a block.

See nLockTime in Bitcoin's protocol specification for more details.

4 Formal specification

The formal specifications assume familiarity with both the bitcoin protocol and various extensions to it, as well as modifications made by Freicoin developers. This document makes reference to but does not specify these extensions and modifications in detail.

4.1 nVersion=3 transactions

This specification defines a new standard bitcoin transaction type, `nVersion=3` transactions (`nVersion=2` being Freicoin's reference-height transactions, which this specification extends). `nVersion=3` transactions differ syntactically from `nVersion=2` transactions in the following ways:

- A possibly-empty sub-transaction list precedes the input list.
- Outputs are prefixed with an asset identifier tag, a 20-byte serialized hash (`ripemd160 . sha256`) of the coinbase transaction from which the output's coins are derived. Each output contains coins and/or tokens from a single asset/currency. For the host currency Freicoin, the similarly-calculated 20-byte hash of the entire chain's genesis block is used instead; within an asset definition transaction, the asset being defined is identified with the 0 hash.
- Outputs are suffixed with an optionally empty, sorted list of unique token bitstrings.
- An optionally empty sorted-list mapping of `<assetid:token, scriptSig>` signatories is added immediately following `vout`.
- A new script field, split into two fields `scriptValidPubKey` and `scriptValidSig`, is added following the signatories' list.
- A new 32-bit block-time field, `nExpireTime`, is added immediately following `nLockTime`.

The following modifications are made to the validation rules for nVersion=3 transactions:

1. If a sub-transaction list is present, each nested sub-transaction must independently validate, according to the rules for sub-transaction validation.
2. Sub-transaction aggregate input and output balances are calculated at the sub-transaction's reference-height, and then time-adjusted to the enclosing transaction's reference-height, before being summed together as contributors to that transaction's aggregate balance.
3. The asset tag of each output must reference an asset that still has unspent, unpruned transaction outputs. (Coins or tokens may be destroyed by sending them to the category of prunable, unspendable `scriptPubKey` prefixed by `OP_RETURN`, and if all unspent outputs of an asset are so constructed, the asset itself is considered destroyed.)
4. For a block-level transaction, each asset/currency must independently balance (input coin >= output coin, input tokens equal to or a superset of output tokens; the difference if any left as a fee to the miner). A transaction which has a signature from a token in the asset's issuers list is exempted from this requirement for that particular asset, as are asset definition transactions for asset being defined.

5. Each signature in the signatories mapping must reference an existing token, execute and run to completion using that token's `scriptPubKey` without abnormal termination (with the other signatories removed during signature operations) or else the transaction does not validate. A script that does not finish execution with a non-zero value on the top of the stack is not a valid signature, but otherwise does not stop transaction validation.

   For example, if an authorizer's signature is required and only one such signature is present and it terminates with zero on the top of the stack, then the transaction does not validate (error: missing authorizer signature). But if there are two such authorizer signatures, and at least one of them passes then the transaction may still validate.

6. For each asset used in the transaction, if that asset has a non-empty list of authorizers, at least one such signature must be present in the signatories mapping.

7. The `scriptValidPubKey` and `scriptValidSig` of the block-level transaction and each nested sub-transaction at any depth, when separately combined and executed must run to completion without abnormal termination, and return a non-zero value on the stack for a transaction to be valid.

   As a special case, if both `scriptValidPubKey` and `scriptValidSig` are empty, the check is skipped for that script.

8. The current time or block height must be less than or equal to the transaction's `nExpireTime`, where the single field can be interpreted as either a block number or UNIX timestamp in the same manner as `nLockTime`.

9. For the purposes of enumeration and indexing, the inputs and the outputs of the block-level transaction are counted first, followed its sub-transactions in order. This corresponds to a depth-first, pre-order traversal of the sub-transaction tree.

10. If the transaction is a coinbase but not the first transaction of a block, then extra validation rules for asset definition transactions apply.

4.2 Hierarchical sub-transactions

Any nVersion=3 transaction includes an optionally empty nested level of sub-transactions, serialized in-between the `nVersion` and `vin` fields. Sub-transactions differ syntactically from regular transactions in the following ways:

- Sub-transactions are prefixed by a `VARINT` value, `nQuantity`, which is required to lie within the semi-closed interval `(0, nGranularity]`.
- Sub-transactions are suffixed with a `VARINT` value, `nGranularity`, which is required to be non-zero.

Sub-transactions are otherwise similar to regular block-level bitcoin transactions, but with additional verification rules:

1. Null (coinbase) sub-transaction inputs are not allowed.
2. Inputs and outputs do not need to balance (aggregate input may exceed output for any asset).

3. The reference-height of a sub-transaction must be less than or equal to its enclosing transaction's (and greater than or equal to each of its inputs and sub-transactions).

4. During script execution, the current frame is the sub-transaction. This means that input or output indices are relative to the sub-transaction, and signature operations evaluate the hash of the sub-transaction only.

5. When performing signature operations within the frame of the sub-transaction, `nGranularity` is included in the hash serialization whereas `nQuantity` is not.

4.3 Asset definition transactions

The coinbase transaction creating an asset is the asset definition genesis transaction. Such a transaction has a single nullary input (thereby marking it as a coinbase), and zero or more ordinary inputs containing freicoins or other asset tokens of any type, typically used to supply a fee [5]. The output vector must include outputs of the newly defined asset (marked by an all-zero asset tag), or else the asset is immediately considered destroyed.

Here are the ways in which an asset definition transactions differ from ordinary transaction types:

- Asset definition transactions must not be the first transaction of a block, which is reserved for the Freicoin miner coinbase.
- As with the Freicoin miner coinbase, the first input of the block-level asset definition coinbase transaction must be nullary (`0` *txid*, `INT_MAX` *n*-index).
- Unlike the Freicoin miner coinbase, the asset definition coinbase string (the `scriptSig` of the nullary input) is allowed to have a length within the closed interval `[0, 65535]`. However the string must be script-parseable and meet other criteria specified below.
- The coinbase string contains the asset's interest/demurrage rate, unit granularity, display scale and external contract hash. These values are `decimal64`, `decimal64`, signed integer and a 20-byte serialized hash (`ripemd160 . sha256`) respectively.
- Other inputs besides the nullary input are allowed.
- An asset definition generating transaction may not hash (ripemd160 . sha256) to any extant asset tag unless all asset tokens for the previously defined asset have been destroyed by spending to a provably unspendable, prunable output (`scriptPubKey` prefixed with `OP_RETURN`).
- The 0-hash asset tag refers to the asset being defined, within the context of the asset definition transaction only.
- The transaction does not require its own issuer or authorizer signatures (the issuer and authorizer lists of the asset being defined take effect *after* the asset definition transaction).

[5] This is in contrast to regular coinbase transactions which do not currently allow extra inputs.

5 Example use cases with scripts

Here we expand the set of possible Bitcoin contracts [6] by using the protocol extensions.

5.1 General notation conventions

The following examples will use a summarized notation for transactions that doesn't represent accurately the actual serialized format and may lead to confusion. For example, the transaction:

```
input: 100 FRC
output: 100 FRC to Alice
```

Means that the payer (a general term for the user building the final transaction, regardless of it being an actual payment or a trade execution) adds signed inputs totaling at least 100 FRC and inserts a change address reclaiming the remainder minus fees. For the most part, change addresses, miner fees, and reference-height details are elided from the examples in order to keep the presentation clear.

`alice1`, `alice2`, `alice3`, etc. are all addresses or script hashes (p2sh) that Alice controls. `pubA`, `pubB`, `pubC`, etc. are custom assets issued in the public chain. `privA`, `privB`, `privC` are all private assets managed outside the public chain, if nothing is said, it is assume that they're issued in different accounting servers. If it needs to be clarified, a label for the chain or server will be prefixed as follows: `FRC_CHAIN_ID:FRC`, `FRC_CHAIN_ID:pubA`, `chainB:pubB`, `accountantC:privC`, `accountantD:privD`, etc.

5.2 Basic uses

5.2.1 Peer-to-peer exchange

Sub-transactions enable the creation of partially valid transactions that act like open binding orders that wait outside of the chain. For example, considering this offer1 created by Alice:

```
input: 50 pubA
output: 100 pubB to alice1
granularity: 10
```

The price is here 2 pubB for each pubA, and the offer can be divided in smaller pieces of 10 pubB for each 5 pubA, as specified by granularity 10.

While the 100 pubA remain in the UTXO set, anyone can use this sub-transaction in a full valid transaction. Bob buys 10 of those pubA by broadcasting this transaction:

```
sub-txns: <quantity=2, offer1>
input: 20 pubB
output: 10 pubA to bob1
```

[6] There are several contracts use cases already described in the bitcoin wiki: https://en.bitcoin.it/wiki/Contracts

Since the `nQuantity` specified for offer1 is 2, he has to put in 20 pubB and he can claim up to 10 pubA.

Although the sub-transaction has appeared in the chain already, the 50 pubA referenced in the offer hasn't been fully spent yet, 40 pubA remain in the offer. Carol could take 20 more pubA with this transaction:

```
sub-txns: <4, offer1>
input: 40.1 pubB, 0.1 FRC
output: 19.99 pubA to carol1
```

Since Carol has payed more than needed and claimed less than she could, the miner gets the 0.1 pubB, 0.1 FRC and 0.01 pubA as fee.

Finally, Alice decides to cancel the offer by just spending the remaining 10 pubA which is left in the partially-spent transaction output:

```
input: tx_id:output where the 50 pubA were originally contained
output: 10 pubA to alice2
```

Since this clears out the remaining balance of the output, it is removed from the set of unspent transaction outputs, and any further attempt to use offer1 will invalidated.

Although in this examples the payers (Bob and Carol) use the orders directly and actively, miners can act as exchange engines by pairing matching crossover orders as described in section 5.3.3.

5.2.2 Transitive trust relationships

By issuing assets representing IOU debts and signing outstanding offers representing lines of credit, standard marketplace mechanisms can be used to execute payments through networks of transitive trust relationships. These payments look like the marketplace transactions involving 3 or more asset types.

Alice, Bob, and Carol issue public assets pubA, pubB, and pubC representing bitcoin IOUs. For simplicity we use public assets and bitcoins over freicoins to avoid complicating the example with cross-chain trade and demurrage.

5.2.3 Baskets currencies

A *basket currency* can be issued and fully managed within the block chain. The basket manager issues asset value and then offers it in bidirectional exchange for multiple other assets at a fixed rate.

5.3 Auctions

5.3.1 English Auction

In the English auction, the owner of an asset declares his intent to sell by auction, and starts collecting bids like the following examples:

```
input: 100 FRC
output: 1 item to bid1
```

```
input: 110 FRC
output: 1 item to bid2
```

When the auction is ended, the seller selects the highest bid and composes a complete transaction:

```
sub-txns: <bid2>
input: 1 item
output: 110 FRC to seller1
```

Since this is a higher-level transaction, the signature of the seller covers the included highest bid sub-transaction, so it is not possible for another bid to be substituted for the winner.

5.3.2 Dutch auction

A Dutch auction is basically the same as an English auction, but with the roles of the buyer and seller reversed in the protocol. The seller suggests a price by constructing a signed offer like the following:

```
input: 1 item
output: 120 FRC to offer1
```

The seller then broadcasts this offer and waits some period of time to see if anyone takes it. If not, the price is lowered and a new offer broadcast:

```
input: 1 item
output: 110 FRC to offer2
```

The seller knows an offer has been accepted and the auction closed when he detects a transaction of the following form on the network:

```
sub-txns: <offer2>
input: 110 FRC
output: 1 item to buyer1
```

The first buyer to get a combined transaction on the chain using one of the seller's offers wins the auction.

5.3.3 Double auction (market/exchange)

This is a generalization of the multi-item English auction, which is basically a regular market with the miners handling order execution. For any asset pairing, an out-of-chain mechanism exists for building, sharing, and collecting signed offers.

Alice offers to buy 100 pubB at a price of 0.500 pubA for each pubB, in units of 10 pubB at a time:

```
input: 50 pubA
output: 100 pubB to bid1
```

```
granularity: 10
```

Bob independently offers to sell 20 pubB for 9.5 pubA, a price of 0.475 pubA for each pubB, in units of 5 pubB at a time:

```
input: 20 pubB
output: 9.5 pubA to ask1
granularity: 4
```

So long as the bid price is greater than the ask price, as is the case here, it is possible for anyone to combine these two offers together to yield a composite market transaction:

```
sub-txns: <quantity=2, bid>
          <quantity=4, ask>
fee: 0.5 pubA to miner
```

The use of granularity and quantity allow fractional parts of each offer to be claimed.

Note that although the crossover spread could be claimed as an output, anyone else could take the bids and construct their own matching transaction and claim the fee for their own. We assume that miners will know how to do this, and one way or another the crossover spread will ultimately be claimed by them. Market clearing becomes a profitable source of revenue in addition to intentional transaction fees.

5.4 Options

Options[7] are financial instruments typically used to hedge. Here we describe how to implement the most basic types using the protocol extensions.

5.4.1 Call

The in a long call the buyer pays premium P for the right to buy up to Q pubA in exchange of pubB at price X before expiry Exp.

The seller signs the following transaction tx1:

```
input: Q pubA
output: Q pubA to script1
```

script1:

```
DUP HASH160 <seller1 pkh> EQUALVERIFY CHECKSIGVERIFY
HASH160 <seller's secret hash> EQUALVERIFY
IF ( BLOCK_HEIGHT < Exp )
    DUP HASH160 <buyer1 pkh> EQUALVERIFY CHECKSIGVERIFY
ELSE
    DUP HASH160 <seller2 pkh> EQUALVERIFY CHECKSIGVERIFY
```

[7] https://en.wikipedia.org/wiki/Option_(finance)#The_basic_trades_of_traded_stock_options.28*American_style*.29

option sub-transaction sub-tx1:

```
input: script1
output: Q * X pubB to <seller3 pkh>
granularity: N
expiry: Exp
```

The seller signs sub-tx1 with seller1. The buyer only lacks the seller's secret to be able to exercise the sub-transaction. So he pays the premium conditionally to the secret being revelead before Exp2 with the following sub-transaction sub-tx2:

```
input: P pubB
output: -
granularity: 1
expiry: Exp2
scriptValidPubKey:
    HASH160 <seller's secret hash> EQUALVERIFY
```

Finally, the seller completes the buyer's transaction to receive the premium by revealing the secret, allowing the buyer to use the option transaction:

```
sub-txs: <1, sub-tx2>
input: -
output: P pubB to <seller4 pkh>
scriptValidSig: <seller's full secret>
```

Now the payer could at any moment before Exp complete the option sub-transaction with:

```
sub-txs: <n, sub-tx1>
input: Q pubB * X
output: Q pubA to <buyer2 pkh>
scriptValidSig: <buyer1 sig> <buyer1 pk> <seller's full secret>
```

The `nQuantity` n must be lower than the `nGranularity` N. After Exp, the seller can double spend script1 by signing any valid transaction with seller1 and seller2.

The seller in the long call example is taking the short call position.

5.4.2 Put

In the long call example the asset being traded was pubA and pubB was the base currency. If the premium is paid in pubA instead of pubB you can consider that pubB is the asset being traded and pubA is the base currency.

The right to "buy pubA for pubB" is equivalent to "sell pubB for pubA". So in the previous example with the premium being paid in pubA, the long call buyer would be the long put seller, and the long call seller is the long put buyer.

The buyer of a long put is taking the short put position.

5.5 Gateways and Bridges

Gateways are similar to basket currencies: an issuer creates an asset and then distributes it when funds are received out-of-protocol. This could be in the form of a fiat wire transfer, physical deposit of precious metals, or a cross-chain transaction (atomically swapping bitcoin for freicoin, for example). Assets are redeemed by a similar process in reverse.

5.6 Off-chain transactions

For ultimate privacy and scalability, off-chain accounting services are preferred. This proposal provides the missing pieces necessary for accounting servers to implement their own private block chains with a secure audit log and without the expensive distributed consensus mechanism, allowing opt-in global consensus only when it is necessary for "cross-chain" (multi-server, or public/private) trade.

To support global consensus mechanisms, a new suite of extrospective opcodes are added, allowing transactions to contain cross-chain conditional dependencies.

5.6.1 Private buy with public funds

Seller constructs private order (200 privB for 100 pubA):

```
input: 200 privB
output: -
granularity: 4
validation scriptPubKey:
    DELEGATION_SEPARATOR DUP HASH160 <accountantB_pkh> EQUALVERIFY CHECKSIGVERIFY
    FROMALTSTACK(refheight) DUP 8000 EQUALVERIFY
    FROMALTSTACK(amount) DUP 25 DIV QUANTITY EQUALVERIFY
    FROMALTSTACK(tokens) DUP 0 EQUALVERIFY
    seller1 FRC_CHAIN_ID pubA FROMALTSTACK(from) FROMALTSTACK(to) OUTPUT_EXISTS
```

. . . and signs the partial transaction.

The validation script contains `DELEGATION_SEPARATOR`, which is a NOP as far as the script interpreter is concerned, but marks the part of the validation script that needs to be signed by the accountant but not the owners of the inputs in the transaction or sub-transaction, the rest

Note that there's some data being fetched from the stack. That data must be set by accountantB or the script will return false if it's not in the stack. Whoever appears in `CHECKSIGVERIFY` (in this case accountantB) must sign the full transaction with the complete validation script, including what's before `DELEGATION_SEPARATOR`.

The payer (who just wants 50 privB) completes the private transaction with:

```
input: -
output: 50 privB to buyer1
```

The buyer also creates the public transaction:

```
input: 50 pubA
output: 50 pubA to seller1
expiry: 10000
refheight: 8000
```

. . . but doesn't sign it. It sends both complete but not signed transactions to the accountant who reads them and completes the private validation scriptPubKey with:

```
10000 TOALTSTACK(to)
0     TOALTSTACK(from)
0     TOALTSTACK(tokens)
50    TOALTSTACK(amount)
8000  TOALTSTACK(refheight)
```

Finally accountantB signs it all and fills the sub-tx validation scriptSig with:

```
<accountantB_sig> <accountantB_pk>
```

Before validation the `scriptValidSig` and `scriptValidPubKey` are combined together to yield:

```
<accountantB_sig> <accountantB_pk>
10000 TOALTSTACK(to)
0     TOALTSTACK(from)
0     TOALTSTACK(tokens)
50    TOALTSTACK(amount)
8000  TOALTSTACK(refheight)
DELEGATION_SEPARATOR
DUP HASH160 <accountantB_pkh> EQUALVERIFY CHECKSIGVERIFY
FROMALTSTACK(refheight) DUP 8000 EQUALVERIFY
FROMALTSTACK(amount) DUP 25 DIV QUANTITY EQUALVERIFY
FROMALTSTACK(tokens) DUP 0 EQUALVERIFY
seller1 FRC_CHAIN_ID pubA FROMALTSTACK(from) FROMALTSTACK(to) OUTPUT_EXISTS
```

Now if buyer1 signs the public transaction and it gets into the FRC chain before height 10000, the private transaction will be valid. Until that happens or height 10000 is reached the transaction is considered to be in process and after height 10000 without appearance of the public one, the private transaction is invalid.

5.6.2 Buying public assets with private assets

The seller constructs the public order:

```
input: 100 pubB
output: 100 accountantA:privA to seller1
validation scriptPubKey:
    DELEGATION_SEPARATOR
    DUP HASH160 <accountantA_pkh> EQUALVERIFY CHECKSIGVERIFY
```

. . . and signs the partial transaction.

The payer (who just wants 50 pubB) completes the public transaction with:

```
input: -
output: 50 pubB to buyer1
expiry: 10000
```

The buyer also creates the private transaction:

```
input: 50 privA
output: 50 privA to seller1
validation scriptPubKey:
    0 50 0 buyer1 0 10000 pubB FRC_CHAIN_ID OUTPUT_EXISTS
```

The buyer signs the private transaction and sends it with the public one to accountantA. The public transaction only lacks accountantA's signature to be valid. If the public transaction gets into the chain before 10000 the private one is also valid, otherwise is rolled back.

5.6.3 Hybrid Transitive transaction

pubA -> pubB -> privC -> privD -> pubE -> userA

So the payer (userA) will pay pubA and receive pubE in exchange. PrivCs and privDs are managed by accountants accC and accD respectively.

Opened offers:

1) Fully public:

```
input: 100 pubB
output: 100 pubA to userB
```

2) Private for Public:

```
input: 100 privC
output: [100 FRC:pubB to userC]
validation scriptPubKey:
    DELEGATION_SEPARATOR
    DUP HASH160 <accountantC_pkh> EQUALVERIFY CHECKSIGVERIFY
    FROMALTSTACK(refheight) DUP 0 EQUALVERIFY
    FROMALTSTACK(amount) DUP 100 EQUALVERIFY
    FROMALTSTACK(tokens) DUP 0 EQUALVERIFY
    userC FRC_CHAIN_ID pubB FROMALTSTACK(from) FROMALTSTACK(to) OUTPUT_EXISTS
```

3) Private for private:

```
input: 100 privD
output: [100 accC:privC to userD]
validation scriptPubKey:
    DELEGATION_SEPARATOR
```

```
    DUP HASH160 <accountantC_pkh> EQUALVERIFY CHECKSIGVERIFY
    DUP HASH160 <accountantD_pkh> EQUALVERIFY CHECKSIGVERIFY
```

4) Public for private:

```
input: 100 pubE
output: [100 accD:privD to userE]
validation scriptPubKey:
    DUP HASH160 <accountantD_pkh> EQUALVERIFY CHECKSIGVERIFY
```

The payer (userA) who wants to buy 50 pubE for 50 pubA builds the public transaction (pub_{tx}) using offers 1 and 4:

```
input:  50 pubA
output: 50 pubB to userC
        50 pubE to userA
expiry: 10000
```

Since 50 pubB from offer 1 are used to pay C, 50 pubA must go to userB, and those are funded by userA in the inputs so sub-tx 1 is complete and valid. But offer 4 still requires accD to sign the full transaction. UserA still hasn't provided the scriptSig to access those 50 pubA in the inputs neither.

Two private transactions need to be created:

Using offer 2, the payer also builds transaction $priv_{tx1}$:

```
input: -
output: 50 privC to userD
```

The validation scriptPubKey for 2 must be completed pushing 50 as the amount and 10000 as the expiry into the stack. The validity of offer 2 and thus this whole transaction still depends on accC's signature.

The other private transaction ($priv_{tx2}$) is built using offer 3:

```
input: -
output: 50 privD to userE
validation scriptPubKey:
    0 50 0 userC 0 10000 pubB FRC_CHAIN_ID OUTPUT_EXISTS
```

Offer 3 doesn't require any completion for its validation scriptPubKey, but the corresponding scriptSig requires the signatures of both accC and accD.

Now that all transactions are complete, it's time to sign.

First accC signs $priv_{tx1}$ and shares with userA and accD. This is secure because $priv_{tx1}$ still depends 50 pubB being sent to userC.

UserD is secure because $priv_{tx2}$ in which he gives privD will only be valid if $priv_{tx1}$ is valid too, that is, if 50 pubB are sent to userC before expiry as the validation scriptPubKey of $priv_{tx2}$ requires. So accC and accD can sign offer 3 in any order to make $priv_{tx2}$ almost valid.

Now accD signs pub_{tx} to make offer 4 valid.

Only userA's signature for the 50 pubA input is missing. The payer (userA) signs the full transaction and broadcasts. If it gets into the block before expiry, all transactions are valid, otherwise all of them are invalid.

At any point, accC, accD or even userA right before the end could stop signing and forwarding the transactions, but that would only cause all transaction to expire.