Minima Transaction Tutorial

v0.83

Contents:

Introduction

In this document I will go through and explain how to create some basic transactions that use custom scripts. This will show how to use the Minima commands to construct, sign and post transactions. To test scripts and play with different transactions or constructions it is recommended that you use a private chain. Then none of your funds are at risk!

First create a new folder and copy the minima.jar into that folder. We will run everything from there.

From the command line cd into that folder.

To start a private chain..

```
java -jar minima.jar -data minidata1 -test -nop2p -genesis
```

These parameters mean :

-data : the data folder where all the files for this minima node are stored. If you want to specify a folder not in this directory you must use the complete path. If unspecified the default data directory is ~/.minima

-test : this uses test settings which have a faster blocktime and only keep a shorter chain in memory. If you are compiling Minima yourself you can edit these to your needs.

-nop2p : do not try and connect to other nodes in the p2p network

-genesis : delete all old data and create a new genesis block

Once you start Minima in this way you will have your own private chain, you will be sent the genesis Minima to spend, and you can enter commands directly into the console. Check your balance with ‘balance’.

If you need to quit Minima and wish to restart it without deleting old data use :

```
java -jar minima.jar -data minidata1 -test -nop2p
```

Transaction Basics

At its core Minima keeps track of Coins. That's all it does. Not users, not balances, Coins. A coin says - you can spend me if you can unlock the script conditions, and when you spend me, you can send the funds in part or in full to other coins. This is a transaction. A transaction reallocates value from some coins to other coins. I think of a coin as some unit of Smart Value controlled by Smart Contracts.

A transaction in Minima is a set of input coins, a set of output coins and a state variable list from 0-255. An analogue logic gate - Smart Circuitry. Each coin has an amount, address, tokenid and coinid. The address is the hash of a script. You send funds to an address that can be spent - in full - if the script returns TRUE when run. A ‘contract’ is the script that locks the funds in a coin and is in fairness interchangeable with the word 'script'. A transaction can be signed by 1 or more public keys - and Signatures can even be added as state variables if you want oracle style contracts. Minima script is case sensitive.

Users only keep track of coins that are relevant to them, and provide an MMR (Merkle Mountain Range) proof of their validity when they spend them. Each coin can, and by default does, store the state variables along with its details in an MMR database. These can be referenced in scripts using the PREVSTATE command allowing for a basic memory mechanic. A coin remembers the state of the previous transaction it was in. This way sequences, counters and far more interesting scripts can be achieved.

A transaction can 'burn' Minima, when the sum of the outputs is less than the sum of the inputs, and this is how the ordering of transactions in the mempool is achieved. This is how precedence is given to transactions. You burn more, you are added sooner, as the coins that are left in the system will be proportionally more. That's the incentive to add them first. Unlike other chains - these burnt coins are NOT given to the User who finds the block - which happens randomly when they are mining TxPoW transactions. This is important as it removes the incentive to mine for profit and ergo - centralise consensus.

Pruning

To ensure all users on the Minima network can run a Complete node, both validating all transactions and helping in the construction of the chain, all users only keep track of coins and tokens that are relevant to them. All the coin data is stored in an MMR database, a type of hash tree, and when a User wishes to spend a coin they add a proof of its existence to the transaction. This means when using multiple User coins in a single transaction each user may not have all the coin data required.

To assist in this Minima provides useful functions. **coinimport** and **coinexport** allow a User to share coin proofs with other users (this does not allow the spending of a coin - just the knowledge of its existence), and **tokens action:import/export** allows the sharing of token data.

If the coin / token data is recent, in the last 24 hours, then each User will already have all the required data, so this will only apply for older coins.

Let's say you have a coin with coinid :

0x99ADECBF2579801921017F81F523D396AA2ED125BF8DE7244045F86C7F7B56E0

To export this coin so another node can import and now know about this coin.

```
coinexport
coinid:0x99ADECBF2579801921017F81F523D396AA2ED125BF8DE7244045F86C7F7B56E0
```

The command response is the coin data required to be imported by another user.

Then the other node can import this proof by using,

```
coinimport
data:0x0000002099ADECBF2579801921017F81F523D396AA2ED125BF8DE7244045F86C7F7B5
6E000000020E5A95D8CDDA0A66A9EB5180479254B23FFBA427EF20603A83F66F95179D8257
D00010A00000001000100010000000010A00000200D3000100000020130000100
```

A similar set of functions can be run to export and then import token details.

To export a token: **tokens tokenid: action:export**

To import a token: **tokens tokenid: action:import data:**

Layer 1 - On Chain

Basic signed contract

The most common contract in Minima is one that returns TRUE if signed by a single public key. When you use the ‘send’ or ‘getaddress’ functionality this is the address that is automatically created either to receive the funds or the change address. You can view your current default addresses with ‘keys’ and your custom scripts with ‘scripts’. If you need help with the parameters - use ‘help’

A default address in Minima :

```
RETURN SIGNEDBY(0xFFEE67F7C..)
```

Time lock contract

A time lock contract is a script that can only be spent after a certain amount of time - in this case a certain block - has passed.

So the script for that would be :

```
RETURN (SIGNEDBY(0xFFEE67F7C..) AND @BLOCK GT 100)
```

This script will return TRUE if the transaction is signed by the correct public key and the blocktime is over 100. The @ symbol represents ‘Global’ variables that are set automatically and represent variables related to the transaction itself, the coin, and the current state of the blockchain, that you can then access in your scripts,

Global Variables :

@BLOCK - the current block number
@BLOCKMILLI - the current block time in milliseconds since Jan 1 1970
@CREATED - the block this coin was created in
@COINAGE - the difference in @BLOCK and @CREATED.
@INPUT - the input number in the transaction of this coin
@COINID - the coinid
@AMOUNT - the amount
@ADDRESS - the address
@TOKENID - the tokenid
@SCRIPT - the script of this coin
@TOTIN - the total number of input coins
@TOTOUT - the total number of output coins

You can test your scripts without sending them on chain using the ‘runscript‘ function. This allows you to set all the variables and globals as you see fit.

```
runscript script:"RETURN SIGNEDBY(0xFF) AND @BLOCK GT 100" globals:{"@BLOCK":"101"}
signatures:["0xFF"]
```

You can use lowercase when entering the scripts and it will be ‘cleaned’ for you into the correct format.

Now let's attempt another very useful contract, one that requires 2 signatures from different parties to be valid.

MultiSig Contract

You have funds, and you want them to be spent ONLY if **Multiple** parties agree.

Let's go through the full steps required to run a simple 2 of 2 multisig on chain via the Minima commands.

You would run 2 versions of Minima, to simulate both users, still on a private network.

As before - Start node 1 (if not already started)

```
java -jar minima.jar -data minidata1 -test -nop2p -genesis
```

Now - Start node 2

```
java -jar minima.jar -data minidata2 -test -nop2p -clean -port 10001 -connect 127.0.0.1:9001
```

-clean wipes previous data so you don't have to use it if you want to keep your old files

The second node needs a separate data folder, a different port, and is told to connect to version 1 on the default port 9001.

Now you have 2 versions of Minima running on a private test network. You can send minima back and forth to each other.

Next both users will need a public key.. Run this on both. The output will show you a public key

Node 1:

```
getaddress
```

On my setup the Public key (use your output here):
0x1539C2B974C1589C6AB3C734AA41D8E7D999759EFE057B047B200E836BA5268A

Node 2:

```
getaddress
```

On my setup the Public key :
0xAD25E1E40605A68AFE357ECF83E51FE27EC10013851AE95889A00C695D5B9402

Let's talk about **getaddress** vs **newaddress**. Minima creates 64 default keys for you to use as change addresses or to accept money or create tokens etc etc.. to keep the number of proof checks to a minimum (these are checked in every block to see if any coins are relevant to you). This also means that backup and restore operations know which

addresses to look out for easily - so even if you do a new transaction your previous restore file will know the keys of the change address you use. Unless you really need a SINGLE address that is not used for anything else - use getaddress. To create a brand new address, that will not be used for anything else.. then use newaddress.

Now that you have both the public keys.. You can create the multisig contract. You must add the script to BOTH nodes, so they know how to spend the coin.

```
RETURN
SIGNEDBY(0x1539C2B974C1589C6AB3C734AA41D8E7D999759EFE057B047B200E836BA5268A) AND
SIGNEDBY(0xAD25E1E40605A68AFE357ECF83E51FE27EC10013851AE95889A00C695D5B9402)
```

In Minima when you send funds to an address the script is not revealed. It is only added to the transaction when you wish to spend the coin, as an input.

So on both you need to run

```
newscript track:true script:"RETURN
SIGNEDBY(0x1539C2B974C1589C6AB3C734AA41D8E7D999759EFE057B047B200E836BA5268A) AND
SIGNEDBY(0xAD25E1E40605A68AFE357ECF83E51FE27EC10013851AE95889A00C695D5B9402)"
```

This will tell you the address.. In this case :
0x7C6EB00C850E4E95743C6D6A181489D1215F53D39AE9702C42069C9F09DF378C

You will note ‘track’. This tells your node to track all instances of this address. The default is true. Sometimes you will add scripts you need to spend but you do not wish to track all occurrences. (An exchange contract for instance)

Minima will automatically track any coin that has an address you track, or an address you track or public key you own in the state variables.

We now have both nodes able to understand the script. Let's send some funds to it.

On Node 1 - the genesis node..

```
send amount:10
address:0x7C6EB00C850E4E95743C6D6A181489D1215F53D39AE9702C42069C9F09DF378C
```

Now both nodes should show a new coin!

When you try ‘balance’ you will see the confirmed coins and sendable coins. Confirmed are the coins you are tracking.. Sendable are the coins with simple addresses you can use with the send function. It takes 3 blocks for a coin to go from unconfirmed to confirmed.

Now lets construct a transaction that uses this input.

```
txncreate id:multisig
```

This creates a custom transaction.

Now we need to find the coin to add as input

```
coins relevant:true
```

You could just use 'coins' here on its own and it defaults to 'coins relevant:true'.

This will display all the coins you are tracking. You can search for any coin using this function but we are only interested in our coins for now.

Copy the coinid.. For the coin with the address and amount 10. Then add it to your transaction. This will be the same on both nodes.

```
txninput id:multisig
coinid:0x9EAD12B53C8B595BFAFA636BC844AB51E3BF3B4B463DFF6D983FA236B3AEB49F
```

Your coinid will be different. You should now see a transaction with 1 input. You can check the txn with :

```
txncheck id:multisig
```

And list them with

```
txnlist
```

Now - let's add an output

1 to be sent to 0xFF (or whatever - you could create a newaddress on one of your nodes) and 9 to be sent back to the multisig as change. Remember that all transactions spend all the inputs so change outputs are required.

```
txnoutput id:multisig address:0xFF amount:1
```

```
txnoutput id:multisig
address:0x7C6EB00C850E4E95743C6D6A181489D1215F53D39AE9702C42069C9F09DF378C
 amount:9
```

Now - we have an almost complete transaction.. BUT it needs both nodes to sign it.

So - on this node.. Where you are constructing the transaction..

```
keys
```

This will show all your public keys. Pick the Public Key you used to make the multisig and run :

```
txnsign id:multisig
publickey:0x1539C2B974C1589C6AB3C734AA41D8E7D999759EFE057B047B200E836BA5268A
```

You now have a partially signed transaction. You need to export it, to a file, and send the data to the other node to sign..

```
txnexport id:multisig file:mymultisig.txn
```

Now on the _other_ node.. You can change the id if you wish..

```
txnimport file:mymultisig.txn
```

Now you should have the transaction on the other node.. Sign it with the other key..

```
txnsign id:multisig
publickey:0xAD25E1E40605A68AFE357ECF83E51FE27EC10013851AE95889A00C695D5B9402
```

And NOW - you should have a correctly signed transaction.. by both parties.. Just needs the correct MMR proofs and scripts.. Which is done with..

```
txnbasics id:multisig
```

The reason this is not done immediately when you add an input.. Is that you may be posting this transaction weeks or months after it is created and will need up to date proofs

And now - we can attempt to publish the transaction

```
txnpost id:multisig
```

IF everything has gone to plan - you will post that transaction and you will see the coins appear on both sides!

More Complex MultiSig

The above example has 2 signatures and 2 parties. But you can use a more complex script to give more power to the users.

You could use 2 keys for one party, and 1 key for the second. 1 of the keys is cold and buried in the ground. 1 of the keys is hot - on your computer. And the second party is a co-signer. Something like :

```
IF SIGNEDBY ( 0xColdKey) THEN
  RETURN TRUE
ELSEIF SIGNEDBY( 0xHotKey ) AND SIGNEDBY( 0x2ndKey) THEN
  RETURN TRUE
ENDIF
```

Why? This contract says : If you sign with the cold key (the ROOT key) you can spend me. Otherwise both you and the co-signer must sign. You could make a bank or trusted person a cosigner, and so to access the funds a hacker would need to hack BOTH of you. Just one of you is not enough. Much more secure. The cold key is NEVER on a connected computer unless one of you is hacked, and then you use that to move the funds - in an emergency.

M of N MultiSig

An M of N signature is a threshold signature. It requires a certain number of people in a pool to sign. Minima has a function just for that. So if you need 2 of 3 to sign.. You can use..

```
RETURN MULTISIG ( 2 0xFirstKey 0x2ndKey 0x3rdKey )
```

In fact our first 2 of 2 multisig could have just been

```
RETURN MULTISIG ( 2 0xFirstKey 0x2ndKey )
```

SlowCash

This simple contract allows you to send funds to a contract that only allows a certain amount to be withdrawn over a period of time. There are more complex variants.

```
IF @COINAGE LT 10000 THEN RETURN FALSE ENDIF

ASSERT SIGNEDBY ( 0xOwnerPublicKey ) AND VERIFYOUT(@INPUT @ADDRESS
@AMOUNT*0.9 @TOKENID TRUE)
```

This allows the owner to withdraw 10% every 10000 blocks, forever. It never stops.

Hashed Time Lock Contract

A VERY powerful variant on the time lock contract and the backbone and basis of the lightning network. This is a contract that has both a time lock and a hash lock. This means you can spend it at a certain time IF you know a secret. The preimage of a hash..

Let's say you own the coin. And your public key is 0xFF.

The other party you are sending the coin to is 0xEE

```
IF @BLOCK GT 1000 AND SIGNEDBY(0xFF) THEN
   RETURN TRUE
ENDIF

RETURN ( SIGNEDBY(0xEE) AND KECCAK(STATE(1)) EQ 0x546FCD56E.. )
```

What this does :

You can cancel the contract after block 1000 (you could use @COINAGE) - but until then only the 0xEE user can claim the funds if they know the secret. They need to know the secret to claim before that.

Also - you will note we have used a state variable!.. This is where the value of the preimage is put.

You could use this to perform an atomic cross chain exchange, The same contract would be written by the other party, using the same hash, on a different chain. Then you collect on the other chain, since you know the secret, show the preimage and they can then collect on Minima. Minima also has SHA2 and SHA3 as hash functions for this very purpose.

You can use the 'hash' function to hash the data.

```
hash data:0xFFEEDD
```

And use that in your scripts..

One thing of note here..

If you were to use a String for the preimage, so :

```
hash data:"this is my secret"
```

You would get
0x1D63D6377EA45E6A5F410FCCED3066B80CA9FB391C346E74948FBB27C7617908
as the hash of the data - the byte representation of the string. Also - this is the KECCAK hash, and you would probably want to use SHA2 or SHA3 for cross chain antics as that is what BTC and ETH support.

Now to add this as a state variable you would need to enclose it in square brackets.

So..

```
runscript script:"LET preimage=STATE(0) RETURN KECCAK(preimage) EQ
0x1D63D6377EA45E6A5F410FCCED3066B80CA9FB391C346E74948FBB27C7617908"
state:{"0":"[this is my secret]"}
```

This is because Strings in KISSVM are enclosed in square brackets.. Not quotes - ".

Another already widely used case is in the Lightning Network on Bitcoin. HTLC contracts work just as well off chain as on chain.. You can set up cascading HTLC contracts that allow for payments to hop through multiple parties.

For instance - imagine I have a channel with Bob and Bob has one with Alice. I want to pay Alice. I create an HTLC payment to Bob locked for 12 hours if he knows a secret that only I know. He then creates the same HTLC payment to Alice, locked for 6 hours. I then reveal the secret - give it to Alice, and she collects the payment from Bob. Bob now also knows the secret and collects his payment from me. This can be used in larger groups with more hops, so instead of only being able to pay people you know.. You can pay people who know people who know people you know. A much larger set - in fact at 6 hops you should theoretically be able to pay anyone in the network.

Exchange Contract

Lets create a simple Layer 1 exchange contract for Minima and some tokens. Minima allows you to create tokens very easily - and tokens can have their own script. When sending a token the address script AND token script must return TRUE.

For now let's first create a simple token

```
tokencreate name:mytoken amount:1000
```

This will create a token with a globally unique tokenid.

Use balance and you will see it there.

Now create yourself a newaddress, we will use the public key and address

Use your key from before (or create a new one).. what does this script do..

```
IF SIGNEDBY(0xTheOwnerPublicKey) THEN RETURN TRUE ENDIF

RETURN VERIFYOUT( @INPUT 0xYouAddress 10 0x00 TRUE )
```

This contract says, the owner can cancel at any time, by signing with their key OR you can spend it if you send 10 Minima (tokenid 0x00) to 0xYourAddress at the opposite output in the transaction.

ASSERT will do nothing if the expression is TRUE but RETURN FALSE if the expression is FALSE.

VERIFYOUT checks that an output exists of a certain address , amount and tokenid - at a certain position, and if you are keeping the state. By using @INPUT you know the opposite output must be the desired one and multiple exchange contracts can take place in a single transaction.

You would need to add this as a script as before. But it's not so useful as it is specific to you..

A better contract would be..

```
IF SIGNEDBY( PREVSTATE(0) ) THEN RETURN TRUE ENDIF

ASSERT VERIFYOUT( @INPUT PREVSTATE(1) PREVSTATE(2) PREVSTATE(3) TRUE )

RETURN TRUE
```

This contract is generic, the values would be entered as state variables in the initial transaction, you can set state variables in 'send', when you send some tokens/ Minima to the address, and it allows you to specify how much of what token you want sent to a specific address.

You can find it's address with

```
scripts action:clean script:"IF SIGNEDBY( PREVSTATE(0) THEN RETURN TRUE ENDIF ASSERT VERIFYOUT( @INPUT PREVSTATE(1) PREVSTATE(2) PREVSTATE(3) TRUE ) RETURN TRUE"
```

In this case this is :
0xBB696A834B6FD91F62A28C9BDCF3754C77F03AFFFEE4EBA3A7485FDC8FD2F3C9

You can easily send funds with the state variables using the send function..

```
send amount:1
address:0x39AC9C96DBC9E4A108E6EBB795003A53F11258BC47EECF0C03275C500CB8DDA1
state:{"0":"0xOwnerKey","1":"0xOwnerAddress","2":"0xDesiredAmount","3":"0xDesiredToken"}
```

'Clean' takes your script and formats it correctly for use on Minima. You can use lowercase and spaces and these will all be removed / fixed.

FlashCash

Flash loans - where you borrow a coin for a single transaction - allow for many arbitrage possibilities.. If you want to allow people to use your coins.. As a flash loan.. How about sending your funds to this address :

```
IF SIGNEDBY( PREVSTATE(1) ) THEN RETURN TRUE ENDIF

ASSERT SAMESTATE ( 1 1 )

RETURN VERIFYOUT( @INPUT @ADDRESS @AMOUNT*1.01 @TOKENID TRUE )
```

Again this is a generic contract, so users can spot them on chain - it doesn't have to be of course you could specify your public key - and have a unique address - but defeats the point as you want users to know this address.

When sending funds to this contract, place your public key as state variable 1..

This says :

The owner - PREVSTATE(1) - can cancel at any time..

Or you can spend this coin if you send the same Tokens / Minima to the same address with 1% more.. (AND make sure state 1 == prevstate 1)

Simple.

You could even specify the 1% increase as a state variable (@AMOUNT*PREVSTATE(2)), to make it 0.1% or whatever you would like. Coupled with the exchange contracts.. Users can borrow coins when they see an opportunity and use them to construct a transaction, as long as they give you back all your coins + 1%, in the same transaction.

MultiSig MultiCoin

Lets now try a more complex but more powerful exchange contract.

2 users are negotiating the price of a certain amount of tokens or an NFT. They do this over a chat app (MaxChat) and send offers backwards and forwards to each other.. All off chain. When they finally agree on a price.. what they would like is to construct a transaction that takes the Tokens and Minima as inputs (one party does one of each) and then sends those 'coins' to the correct recipient.

All of this can be checked and validated by each user before either signs the transaction so that the swap itself is atomic, trustless and secure. Either they both get what they want or it doesn't happen. These types of offers could be collected together in a pool, or shared with other users, to allow for a completely decentralised exchange dynamic.

Let's go through the steps to construct this transaction.

I have 2 instances of Minima running. I have created an NFT token on one of them.

```
tokencreate name:mynft amount:1 decimals:0
```

Now the other user wishes to buy this off me for 10 Minima.

To remove the need to worry about change the other user creates a 10 Minima coin.. By using getaddress and then sending 10 Minima to his own address. (Normally you wouldn't do this and just send the change back to yourself when constructing the transaction)

```
send amount:10
address:0x0AC281E79A096F046A1FAEF17D268BCF6D5DA604F533F3C64FE84F079C775FCE
```

So we now have 2 coins, 1 on each Instance of Minima, and we want to create a transaction that spends them both and sends them to the other party..

The user who owns the NFT starts by creating a transaction

```
txncreate id:swap
```

He then gets one of his addresses and adds an output paying him 10 Minima.

```
txnoutput id:swap
address:Mx1W5E6AESNJG687D8GZC2JF9DZVVA3WE3TUW9PGWRRRCM32YV1BKMPRR2VY
amount:10 tokenid:0x00
```

Note that I use the Mx.. style address!.. This is a special address format (converted before used - you can't use them in Scripts) that takes the original 0x.. HEX address, hashes it, adds the first 4 bytes of the hash as a checksum to the end and then converts the whole thing to Base32. This way you have basic error checking so you can't input an invalid address and the size remains less than before.

He then finds his NFT ('coins' is the same as 'coins relevant:true')

```
coins
```

And adds the coin as an input.. Note the use of scriptmmr !

Note: This next part may require you to export & import the coin proof. Read the Pruning section to learn more.

```
txninput id:swap
coinid:0x5869DDC397979D9529AD92658C0FACC183D94F3E23F7E8028E02B183DE2FAB8B
scriptmmr:true
```

This adds the coin and the script and MMR proof. But it is not yet signed.. Since the transaction is not yet complete.

Export the transaction and let the other user import it. We have not signed it so the transaction is still small.. We can just copy paste the data without going via a file..

```
txnexport id:swap
```

Now the other user imports it.. Again just using the data.. This works well over RPC but the command line has a limit - which is why you would use files for larger signed transactions.

```
txnimport
data:0x0000004737761700001010000000205869DDC397979D9529AD92658C0FACC183D94F3
E23F7E8028E02B183DE2FAB8B000000204BECB914994CE42E94209D0DD4EECAA19262DE6
B502EF9FC6347C416BC19EC4F2C01010000002002C30D6CD58E0230481C0A5075E40A837E
DEA7331F7423BA4AE807315623188401000100000000010800000020BCF0001000100000020586
9DDC397979D9529AD92658C0FACC183D94F3E23F7E8028E02B183DE2FAB8B0000000B5245
5455524E205452554500012C2C0101000000107B226E616D65223A226D796E6674227D00020B
CF00010100000020 8DA0C0DEC62C8DAA07703706A214E2418C86070EF166F8EBDBB202D4D
DB71A8900000020322B8CA772F3819076A21860A6F4B71FFA87270FBE9273096F7B65862AFC
2BA500010A0000000100010001000000000100000001000001000000010000000001000001000
0101000000205869DDC397979D9529AD92658C0FACC183D94F3E23F7E8028E02B183DE2FAB
8B000000204BECB914994CE42E94209D0DD4EECAA19262DE6B502EF9FC6347C416BC19EC
4F2C01010000002002C30D6CD58E0230481C0A5075E40A837EDEA7331F7423BA4AE8073156
2318840100010000000001080000020BCF000100010000000205869DDC397979D9529AD92658C
0FACC183D94F3E23F7E8028E02B183DE2FAB8B0000000B52455455524E205452554500012C2
C0101000000107B226E616D65223A226D796E6674227D00020BCF0100020BF40001010000000
0208C518BEDACB2720EB9D72C0CB7B1C4E856CA1A3977F162374B0D48C1B1B99B602C141
18427B3B4A05BC8A8A4DE8459867FE8B78917FF000101000000535 2455455524E205349474E4
```

```
5444259283078463836333245313141344632423445374339454544463838314338444434394538
3445443831463230343035393435373243454437463938373335393144414129000001 00
```

Gets an address for himself.. And adds an output sending the NFT to himself.

```
txnoutput id:swap
address:Mx3H6YQPJT9UW2UYAWFZSD2635WWPREEUABBFWY2UYH4Y1BFQZJ91W8JK3SJ
amount:1
tokenid:0x02C30D6CD58E0230481C0A5075E40A837EDEA7331F7423BA4AE8073156231884
```

And finally adds the 10 Minima input.. Which he finds with coins again..

```
txninput id:swap
coinid:0x84352ECF59CAC79103C07254B5158AF8560DE68F609A346154FA812F0051A
B44 scriptmmr:true
```

OK - almost there.. We need to sign it now..

```
txnsign id:swap publickey:auto
```

NOW - export that.. via a file.. The signature makes it too large for command line copy paste..

```
txnexport id:swap file:swap.txn
```

Back on the other instance of Minima.. Import the Txn..

```
txnimport file:swap.txn
```

NOW - CHECK the transaction is as you would want it.. And sign it..

```
txnsign id:swap publickey:auto
```

And.. finally - post it..

```
txnpost id:swap
```

.. And like that - It's done.

You just performed a trustless atomic secure token swap! Well done..

It may seem slow and clunky to type this all in by hand but this entire exchange of contracts would take less than a second when automated.

This procedure allows for truly secure peer 2 peer exchanges to take place and can be used as a base unit for many other interesting products, such as a DEX, trading pit, liquidity pools etc etc.. With all the negotiation / order book management done off chain.

MAST Contracts

MAST stands for **M**erkelized **A**bstract **S**yntax **T**ree. It is a technique that not only allows for very large scripts to be used, it also greatly increases privacy. Instead of providing the entire script - you can MAST sections of it and only present the information required for the path of execution. It involves making hash trees of values, providing the root of the tree, the leaf node and accompanying proof. The proof grows as log(n) for n - standard binary hash tree.

In other words.. You have this script :

```
IF SIGNEDBY(0xFF) THEN
  //RUN SOME CODE - CODEBLOCK 1

ELSEIF @BLOCK GT 100 THEN
  //RUN SOME OTHER CODE - CODEBLOCK 2

ENDIF
```

If codeblock1 is run then you never need to know what was in codeblock2. That code is never executed. These blocks of code could be large, very large. This way you reduce the amount of data needed to be sent with the transaction, decreasing size and increasing privacy. The code could be MegaBytes in size.. all that matters is that the path of execution through the code is below the Minima KISSVM limit. Currently this is set to 512 operations.

Since a MAST block is a hash tree of different values.. The leaf nodes of the tree are the allowed values.. so you can have *millions* of possible script blocks, all accessed via a different merkle proof to root. Say you wanted to have a 1 of 10,000 multisig. You could not add 10,000 public keys to a script - as it would make the transaction too large to send over the network. But - each leaf node could be a different SIGNEDBY code block, and you would present that code with the correct proof to be allowed to spend that coin. Or.. perhaps you could use this technique to check for valid game states - tic tac toe would require 3^9 possible final states.. And when you play the game you provide the winning state (a leaf node + proof) which is checked in the script. (More on that later..)

The way you would package this up is :

```
IF SIGNEDBY(0xFF) THEN
 MAST 0x72BE56DFD48B785139A72512FEAAC7E339B8F48132E9B9340A248EFC00F4A5DA
ELSEIF @BLOCK GT 100 THEN
```

```
 MAST 0xFA1B16685F09FA56581614AC55E731697C46926392129F3A6BF8FA5EE202A251
ENDIF
```

Then you provide the script proof for the particular MAST block. You can have MAST blocks inside other MAST blocks of course.

Let's now go through a complete example. Let's create a MAST block..

```
mmrcreate nodes:["RETURN TRUE","RETURN FALSE"]
```

This will create an MMR tree, with 2 leaf nodes. You can check the proofs with the data provided by mmrcreate..

```
mmrproof data:"RETURN TRUE"
proof:0x000001010000000002068073D52B5CE60A854BA2AA42CFB2E27D9FADFC9C4F7EA52F
E48E58F604EBEC7
root:0x0E321692DA8996A833F88EC8A73F3AA8A5E949AD12FF48207130C7AE6F9DC115
```

The root value is the value you give MAST. The above returns true.. but you could also use :

```
mmrproof data:"RETURN FALSE"
proof:0x000001010100000002072BE56DFD48B785139A72512FEAAC7E339B8F48132E9B9340A2
48EFC00F4A5DA
root:0x0E321692DA8996A833F88EC8A73F3AA8A5E949AD12FF48207130C7AE6F9DC115
```

This will also return true. What is important to note is that BOTH have the same root. It is the merkle proof that is different for both.

So how to check this in a script..

```
runscript script:"MAST
0x0E321692DA8996A833F88EC8A73F3AA8A5E949AD12FF48207130C7AE6F9DC115"
extrascripts:{"RETURN
TRUE":"0x000001010000000002068073D52B5CE60A854BA2AA42CFB2E27D9FADFC9C4F7EA5
2FE48E58F604EBEC7"}
```

You could create interesting scripts with multiple ways of them being executed.. You can even use these merkelized proofs as state variables and check them yourself in script.

In this case..

```
runscript script:"LET script=[RETURN TRUE] LET
proof=0x000001010000000002068073D52B5CE60A854BA2AA42CFB2E27D9FADFC9C4F7EA52F
E48E58F604EBEC7 LET
root=0x0E321692DA8996A833F88EC8A73F3AA8A5E949AD12FF48207130C7AE6F9DC115
ASSERT PROOF(script proof root) EXEC script"
```

PROOF takes the same arguments as mmrproof. You could put those variables as state variables and have a generic contract that can run ANY of the leaf node scripts you create in your tree.. like so :

```
runscript script:"LET script=STATE(0) LET proof=STATE(1) ASSERT PROOF(script proof 0x0E321692DA8996A833F88EC8A73F3AA8A5E949AD12FF48207130C7AE6F9DC115) EXEC script" state:{"0":"[RETURN TRUE]",
"1":"0x00000101000000002068073D52B5CE60A854BA2AA42CFB2E27D9FADFC9C4F7EA52FE48E58F604EBEC7"}
```

You would still want a signature - or put that as a requirement in the leaf node script.

When adding extra scripts to a custom txn you can use txnscript. Here is an example.

First let's calculate the simplest MAST script

```
scripts action:clean script:"RETURN TRUE"
```

This will return the clean version.. And the address :
0x72BE56DFD48B785139A72512FEAAC7E339B8F48132E9B9340A248EFC00F4A5DA

Now let's create a MAST script that uses that

```
newscript script:"MAST 0x72BE56DFD48B785139A72512FEAAC7E339B8F48132E9B9340A248EFC00F4A5DA"
```

This will create a new address :
0x459C3CE5EDDF8E78F901A7289981640A8A3A83E2B95558435BFEBBD674CF8D50

Now send funds to it.

```
send amount:1 address:0x459C3CE5EDDF8E78F901A7289981640A8A3A83E2B95558435BFEBBD674CF8D50
```

Now wait for those to clear. You can use 'coins relevant:true' to see your coins AND to get the CoinID.

Now construct a txn..

```
txncreate id:txnmast
```

```
txninput id:txnmast coinid:0xB2F2E123F6A1E00E956390B3BEDAF48CB475279CD73EC7D4BD2E0D56823A09A3
```

Your coinid will be different..

```
txnoutput id:txnmast address:0xFF amount:1
```

And now we should have a simple txn.

Try and post it.

```
txnpost id:txnmast auto:true
```

We use auto:true which is the same as txnbasics.

It FAILS!.. On the console it will print..

'Script FAIL 0 MAST
0x72BE56DFD48B785139A72512FEAAC7E339B8F48132E9B9340A248EFC00F4A5DA'

This is because it does not know how to handle the MAST script.. You need to add the scripts that MAST uses. You can clear all the witness data with :

```
txnclear id:txnmast
```

Now add the details about that script :

```
txnscript id:txnmast scripts:{"RETURN TRUE":""}
```

The JSON holds the script and the proof. If it is a single script, and not one created with mmrcreate, just leave the proof blank. If it is an mmrcreate script, copy the proof in.

Now the transaction knows what that MAST script is.. So try and post it again..

And this time it WORKS!

The Vault

Now we will try a far more interesting script than we have previously. It is similar in principle to the multisig we played with earlier that had a cold and hot key for added security. But instead of trusting someone to be the co-signer, you use a smart contract to ensure funds are sent to an intermediary address before moving on..

Vault wallets are special. They use 'covenants' to ensure that funds are ALWAYS sent to a safe house first. Vaults use 2 sets of keys. One that is HOT. These are kept online and are used to send funds from the vault to the safe house and eventually beyond. Another set is COLD. These are NEVER kept online and are only used in the event of a hack - or error in recipient. The HOT keys can be hacked.. and it's really no big deal.

What this means is that when you wish to send funds they MUST first be sent to a special address for a period of time - the Safe House. This is enforced by the smart contract. Even the owner of the vault must do this, and so therefore even the hacker who has pinched the keys. Once the funds are at the safe house, you must wait a certain amount of time until you can send them to ONLY the pre-specified address. If they were sent by a hacker - you the owner can claim them back, within a certain amount of time, with your COLD key.

Hot wallet convenience with cold wallet security and all it takes is a little delay. An exchange, for instance, can keep large amounts online and available to send this way. Simple (yet secure).

Here's the script for the Vault :

```
/* You need 2 keys */
LET pkcold = 0xCOLD_KEY
LET pkhot = 0xHOT_KEY

/* COLD key has root access*/
IF SIGNEDBY ( pkcold ) THEN RETURN TRUE ENDIF

/* The Amount and Recipient are specified in the State */
LET amt = STATE ( 20 ) LET recip = STATE ( 21 )

/* Now generate the Safe House Address */
LET safehouse = [ LET pkcold = 0xCOLD_KEY LET pkhot = 0xHOT_KEY
                IF SIGNEDBY ( pkcold ) THEN RETURN TRUE ENDIF
                IF SIGNEDBY ( pkhot ) THEN IF @COINAGE GT 20 THEN
                    RETURN VERIFYOUT ( @INPUT PREVSTATE ( 21 ) @AMOUNT @TOKENID TRUE ) ENDIF
ENDIF ]

/* Make sure The Safe House Address is the opposite output */
ASSERT VERIFYOUT ( @INPUT ADDRESS(safehouse) amt @TOKENID TRUE )

/* Send the change back to the Vault */
LET chg = @AMOUNT - amt
IF chg GT 0 THEN
    ASSERT VERIFYOUT ( INC(@INPUT) @ADDRESS ( @AMOUNT - amt ) @TOKENID TRUE )
ENDIF

/* Make sure is signed by the HOT key */
RETURN SIGNEDBY ( pkhot )
```

Here is the expanded safe house script - that is generated by the vault.

```
/* Same Keys as Vault */
LET pkcold = COLD_KEY
LET pkhot = HOT_KEY

/* COLD key again has root access */
IF SIGNEDBY ( pkcold ) THEN RETURN TRUE ENDIF

/* ONLY spend after 20 blocks and ONLY to the pre-specified address for the whole amount */
IF SIGNEDBY ( pkhot ) THEN
  IF @COINAGE GT 20 THEN
    RETURN VERIFYOUT ( @INPUT PREVSTATE ( 21 ) @AMOUNT @TOKENID TRUE )
  ENDIF
ENDIF
```

Quite a lot going on there. Basically the vault itself creates the address to send the funds to. You can see when someone spends funds from the vault. If it wasn't you, you have till the timeout expires to get the cold key and rescue the funds.

Coin Flip

And finally, here is a full example of a 'game' that can be played via smart contracts. It is a coin toss. 2 users flip a coin and heads one wins tails the other wins. The randomness is added to the system by both players picking a random number and committing to it via the hash. Thereby forcing them not to change it when either shows the preimage of that hash. So both players pick a random number, those numbers are hashed, and if the first byte of the final hash is less than 128 Player 1 wins else Player 2 wins. This game is very very simple but more complex games with more participants can be written in a similar fashion.

Importantly - this is running on Layer 1. This is running on chain. So it is slow and expensive. This is not where you would play this game. You want to be running it on Layer 2 - which is instant and free. You want to be off chain. You don't want to be on Layer 1 - you want to be on Layer 2, and this is where the true power of the UTXO model (Unspent Transaction Outputs) coupled with the ELTOO framework really shines. Because this simple game can be 'lifted' off layer 1 and played out on layer 2. In fact any sequence of transactions between a group of users can be lifted off layer 1 and onto layer 2.

More on that later. For now - let's see the script :

You would need an application to manage this sequence, and keep track of where you were, checking the chain for responses and firing off the next required transaction in the sequence. Originally, way back when, I had a web app that did just that over RPC with the Minima node.

A MiniDAPP called CoinFlip - for those that remember.

```
/*
  COIN FLIP SCRIPT

  Funky little script.. with comments.

  Each 'Round' is another transaction.

  The State Variables are :

  0 - Round Number

  PLAYER 1 puts his details in state 1-3
  1 - Payout address
  2 - Cancel / Last Resort key
  3 - Hash of random number

  PLAYER 2 adds his details
  4 - Last Resort key
  5 - Hash of random number

  Then player 1 reveals..
  6 - Preimage of 3

  Then player 2 reveals and pays out ( wins 95% or takes back 5% - incentive to finish game )
  7 - Preimage of 5

  If Player 1 or 2 do not reveal or pay out in a time limit
  the other player gets everything as a last resort..
*/

/* What state are we at */
LET round = STATE ( 0 )
LET prevround = PREVSTATE ( 0 )

/* Make sure we are 1 round ahead of before */
ASSERT round EQ INC ( prevround )

/* PLAYER 2 Joins OR Player 1 Cancels */
IF round EQ 1 THEN

    /* Player 1 can still cancel at this stage */
    IF SIGNEDBY ( PREVSTATE ( 2 ) ) THEN RETURN TRUE ENDIF

    /* Make sure all the details are kept */
    ASSERT SAMESTATE ( 1 3 )

    /* This round someone accepts.. check double the money*/
    RETURN VERIFYOUT ( @INPUT @ADDRESS ( @AMOUNT * 2 ) @TOKENID TRUE )

/* PLAYER 1 REVEALS HIS HAND */
ELSEIF round EQ 2 THEN

    /* If player 1 does NOT reveal.. in time limit (here 20 mins).. player 2 gets everything */
    IF @COINAGE GT 64 AND SIGNEDBY ( PREVSTATE ( 4 ) ) THEN RETURN TRUE ENDIF

    /* make sure all the details of both players are kept.. */
    ASSERT SAMESTATE ( 1 5 )

    /* Now check that the preimage of player 1 is correct */
    LET ponehash = STATE ( 3 )
    LET preimage = STATE ( 6 )
    ASSERT KECCAK ( preimage ) EQ ponehash

    /* OK - He has shown his random number.. continue */
    RETURN VERIFYOUT ( @INPUT @ADDRESS @AMOUNT @TOKENID TRUE )
```

```
/* PLAYER 2 REVEALS AND PAYS OUT */
ELSEIF round EQ 3 THEN

   /* If player 2 does NOT reveal.. in time limit (here 20 mins).. player 1 gets everything */
   IF @COINAGE GT 64 AND SIGNEDBY ( PREVSTATE ( 2 ) ) THEN RETURN TRUE ENDIF

   /* make sure all the details of both players are kept.. */
   ASSERT SAMESTATE ( 1 6 )

   /* Now check that the preimage of player 2 is correct */
   LET ptwohash = STATE ( 5 )
   LET ptwopreimage = STATE ( 7 )
   ASSERT KECCAK ( ptwopreimage ) EQ ptwohash

   /* OK - lets see who wins..! */
   LET ponepreimage = STATE ( 6 )
   LET rand = KECCAK ( CONCAT( ponepreimage ptwopreimage ) )

   /* GET THE FIRST BYTE*/
   LET val = NUMBER ( SUBSET ( 0 1 rand ) )
   IF ( val LT 128 ) THEN LET winner = 1 ELSE LET winner = 2 ENDIF

   /* Calculate the Payout for each */
   LET paywinner = @AMOUNT * 0.95
   LET payloser = @AMOUNT - paywinner

   /* Check that State 8 states the correct winner.. for future easy lookup*/
   ASSERT STATE ( 8 ) EQ winner
   ASSERT STATE ( 9 ) EQ paywinner

   /* Now check the payout! */
   LET poneaddress = STATE ( 1 )
   IF winner EQ 1 THEN
      ASSERT VERIFYOUT ( @INPUT poneaddress paywinner @TOKENID TRUE )
   ELSE
      ASSERT VERIFYOUT ( @INPUT poneaddress payloser @TOKENID TRUE )
   ENDIF

   /* And finally check the signature  - MUST sign as otherwise someone else could claim */
   RETURN SIGNEDBY ( PREVSTATE ( 4 ) )

ENDIF
```

That's quite a long script.. You could MAST the sections in each round to reduce the size you needed to post on chain.

Essentially the game starts when Player 1 sends some coins to this Contract with the initial 4 state variables defined, 0-3. Then player 2 responds, by spending the coin, adding more state variables, and copying the previous ones so that the memory is intact. And so on..

You can follow the logic through, with each round incrementing the state(0) value, so the script knows which section to run. The state is verified in the 'ASSERT round EQ INC (prevround) ' ensuring that every transaction in the sequence MUST increase the state(0) / round number by one.

Next - we get exciting.. we go Layer 2.. we go ELTOO.

Layer 2 - Off Chain

Everything we have done so far is known as 'On Chain'. This is Layer 1. This is the final settlement layer. Everyone on the network processes Layer 1 transactions. This is why it is slow, expensive and does not scale - to limitless amounts.

What we need is a way of transacting with other users without having to go on chain. A way of only dealing with people involved in the specific transaction itself. This is known as Layer 2. This is Off Chain. Off chain transactions are instant, free and scale.

Effectively on Layer 2 Users swap contracts with each other, and as long as everyone plays their part, they can remain on Layer 2 happily transacting, at speed, countless times. Should the Users wish to stop or should there be an issue - that is when the transactions jump down to Layer 1 - the arbitration layer, the settlement layer, and the Layer 2 transactions are 'collapsed' into a single Layer 1 transaction.

Let's take a look at the most basic example.

Uni-Directional Payment Channel

I want to pay someone 1 Minima a day, everyday, for 30 days. How could I do this without using normal on chain transactions ?

Well - one way could be..

1) Ask the recipient for their address
2) On day 1 create a valid signed transaction paying them 1 Minima - but **don't** post it on chain. Just give it to them.
3) On day 2 create another valid signed transaction paying them 2 Minima - but don't post it on chain. Just give it to them. They discard the old transaction, and keep the newer one paying them more..
4) Keep doing this for 30 days
5) On the last day the recipient posts the last transaction and collects their 30 Minima.

There we go, 30 transactions, off chain, with only one transaction actually going on chain at the end of our financial relationship.

BUT - although this *seems* to work, an element of 'Trust' is required. The recipient can check the transaction is valid when they get it - but there is nothing to stop me spending the coins I use in the transactions, late at night on the 29th day, thus invalidating their transaction. I effectively run off with my money. And now the transaction they have is worthless - as they can't post it. The coins are spent.

If they are a friend, someone I know, I would not do this - wouldn't do it anyway.. BUT can we remove the element of Trust altogether so that I can't do that ? Of course we can.. and we use a trick we learnt earlier with our TimeLock and MultiSig contracts.

Here is the timelock multisig contract we could use :

My Public Key is : 0xMyKey
Their Public Key is : 0xTheirKey

```
IF @COINAGE GT (35*1728) AND SIGNEDBY(0xMyKey) THEN RETURN TRUE ENDIF

RETURN MULTISIG(2 0xMyKey 0xTheirKey)
```

1) Create this MultiSig contract with the recipient, with a time lock - that allows me to collect the funds in 35 days. (There are 1728 blocks in a day)
2) I send 30 Minima to that address. I have now locked those coins up for ~35 days and the only other way of accessing them is if BOTH of us sign.
3) On day 1 create a valid transaction spending this 30 Minima coin and send 1 Minima to the recipient and the remaining 29 Minima back to me. I sign it. Then give this partially signed transaction to the recipient. They sign it - but **DON'T** give it back to me. They now have a valid fully signed transaction that they can post at any time and collect 1 Minima.
4) On day 2 create a valid transaction spending this 30 Minima coin and send 2 Minima to the recipient and the remaining 28 Minima back to me. I sign it. Then give this partially signed transaction to the recipient. They sign it - but **DON'T** give it back to me. They now have a valid fully signed transaction that they can post at any time and collect 2 Minima.
5) Keep doing this for 30 days.
6) On the last day the recipient posts the latest valid transaction and collects 30 Minima.

Now - we have a system where I cannot spend the funds before the recipient collects his money. This is trustless and secure. If the recipient doesn't collect the funds within 35 days then I can collect the money and send the 30 Minima back to myself. This is why the timelock has to be more than 30 days.

Even better - 1 transaction per day is used as a convenience. I could be paying him by the hour or by the minute. None of this is posted on chain until the final transaction so we are only limited by the speed at which we can create newly signed transactions and swap them. You could even do real time payments by the second.

One way payments have their uses, but far more useful, are two way.

Bi-Directional Channel

We want to start up a two way payment channel with someone. So I can send them Minima and they can send coins back to me.

If we use the technique we have just demonstrated then what would happen ?

Two (or more) people set up a MultiSig where they send 10 Minima each to the contract. They create a new transaction every time there is an update.

1) User1 sends User2 3 Minima. They create a new fully signed transaction that pays User1 7 Minima and User2 13 Minima.
2) User2 sends User1 1 Minima. They create a new fully signed transaction that pays User1 8 Minima and User2 12 Minima.
3) And so on..
4) When they are finished they post the latest transaction and both collect their funds.

Again - although this *seems* like it is working.. It is not.

1) The timelock part of the contract is more complicated. If one side does nothing for 30 days, how do both parties recover their initial 10 Minima funds? It is not as simple as one person spending the funds, trusting that person, what forces them to send back the correct amount to both parties ?
2) As before what stops someone posting an *earlier* transaction - one where they were receiving more funds than they are by the end ? Both parties now have a valid signed transaction whereas before only one party had the complete transaction.

1 - Can be fixed by creating a valid fully signed transaction spending the MultiSig coin BEFORE it is even posted on chain.

So as before the 2 players create the MultiSig contract. Then they use a 10 Minima coin each and send 20 Minma to this script in a single transaction. But they do not sign it and they do not post it.

```
RETURN MULTISIG(2 0xMyKey 0xTheirKey)
```

There is no timelock on this contract. Both parties must sign for the coin to be spent.

Now - some Minima low level info. All transactions reference the coins they use as inputs by their CoinID. This is a globally unique 32 byte value that each coin has. You can construct the coinid for an output coin by hashing the Transaction Hash + The Output number of the coin in the transaction. This means you can know the coinid of a coin *before* it is posted on chain. This means you can construct a transaction that spends a coin that does not exist yet!

And this is exactly what we do here.. The 2 parties create a transaction that spends the as-yet-unposted coin, and pays 10 Minima back to both, and they both sign that. Now both parties have a valid fully signed transaction that could be used to spend the MultiSig coin, once it is on chain. Then they both sign the initial transaction and post the full 20 Minima on chain to the MultiSig contract. If one of them should disappear and become unresponsive the other has a transaction that sends all the funds back to them. (This trick will be used more later on..)

2 - Unfortunately the previous pre-made spend transaction still does nothing to prevent either user sending the transaction in the sequence where they have the most. In fact.. It makes matters worse, since if by the end one of the Users has less than 10 Minima he can use the initial transaction to send the original funds back to everyone, and collect their initial 10 Minima.

What we need is some way of ordering the transactions so that only the *latest* transaction is recognised as the valid one. Some way of ensuring that the latest transaction in the sequence will always be the final outcome of the financial relationship..

Enter ELTOO.

ELTOO Channel

You should read the ELTOO whitepaper, which I reference in the Minima whitepaper, before diving in here. Essentially what ELTOO does is allow you to have a sequence number, which we will store as a state variable, and then a special contract that can be run as long as it has a higher sequence number (but not allow lower older sequence numbers). Each phase has 2 possible transactions that can be executed, an update transaction that moves you onto the next number in the sequence - the next phase - and a settlement transaction that ends the sequence and pays out.

First let's make sure we have all the components required.

1) We need to be able to create a transaction with a coin that has not been posted on chain yet.
2) We need a 'floating coin' that can be attached to multiple different existing coins as long as it has the same address, amount and tokenid - but different coinid.

Pre-Coin Transaction

Creating a transaction that spends an as yet created coin is not as tricky as it sounds.

Create 2 new addresses and send some funds to the first.

```
newaddress;newaddress
```

Not that you can chain functions with ;

```
send amount:1
address:0x0BD4C2C80609CA63CAF5B004037E6F4F1A4A56F97904B6C1B791802C2F13E504
```

Now create transaction spending that input.. sending 1 Minima to the other new output..

```
txncreate id:pretxn
```

```
txnoutput id:pretxn amount:1
address:0x01C0771C94D680E2E6CADA900D28061223E383EAF9646CA7F045209907D81DA7
```

To find the input coin use :

```
coins relevant:true
```

```
txninput id:pretxn
coinid:0x2156B370A764DFB7D25B8A2F71305B9BECAE6A804E076A0333DF0645C2449CB3
```

You should now have an unsigned transaction with 1 Minima coin input and 1 Minima coin output.

You can sign it automagically - since the inputs are 'simple' with

```
txnsign id:pretxn publickey:auto
```

Now we have a fully valid transaction but have not posted it yet.

Let's create a transaction that spends the output. This is important as the signatures will be dependent on the coinid used in the previous transaction, so any changes will mean they will be invalid.

```
txncreate id:posttxn
```

You will see that when you do 'txnlist'.. the last piece of data in the JSON is 'outputcoindata'. This is a HEX representation of the coin outputs and can be used as an input to another transaction - with the correct CoinID. Copy that and use it like so..

```
txninput id:posttxn
coindata:0x0000002015B0B62181547D83C563D8B1925B53655A2A8F7FF8B1F7479FB202829B
89838200000001FF00010600000001000001000100000000010000000100000010000
```

And add an output to 0xFF.. Just for giggles.

```
txnoutput id:posttxn address:0xFF amount:1
```

Sign the post transaction..

```
txnsign id:posttxn publickey:auto
```

We now have a transaction that cannot be changed, as the signatures would be invalid, using a coin that does not currently exist.

Let's post the FIRST 'pretxn'

```
txnpost id:pretxn  auto:true
```

You will need to wait for it to confirm. Use 'balance' to see,,

And once you have all the funds - the change and the coin you just sent - let's post the second 'post' transaction.

```
txnpost id:posttxn  auto:true
```

If all went well - that should all work and the final result is 1 Minima sent to 0xFF!..

Well Done!

You may have to delete old transactions, using txndelete, if you don't need them anymore..

```
txndelete id:pretxn;txndelete id:posttxn
```

The next ability we will look at is a 'floating' coin.

Floating Coin

A floating coin is one that can be attached to any existing coin, as long as it has the same address, amount and tokenid, irrespective of the coinid.

Use a previous address and send funds to it..

```
send amount:2
address:0x921FC56E2948CCC51DB46525E459F1DB7331C65CC0E830FBC0E63CF273C6B592
```

You will now have a coin that can be used as a floating input. Use coins relevant:true to see the details. In the coin details floating will be set to true.

Create a floating transaction that spends this coin..

```
txncreate id:floattxn
```

```
txnoutput id:floattxn address:0xFF amount:2
```

And this is the special bit, when defining an input coin use the 'floating' param. You could specify the coin using a coinid or with coindata (from a custom transaction) - but you can just specify the address, amount and tokenid, since it is a floating coin.

```
txninput id:floattxn floating:true amount:2
address:0x921FC56E2948CCC51DB46525E459F1DB7331C65CC0E830FBC0E63CF273C6B592
```

Now - you will see that in your custom transaction the coinid for that input is 0x01. This means it is special. It is not specified. It does not matter which coin you use - as long as address, amount and tokenid are the same.

Sign it to complete the transaction

```
txnsign id:floattxn publickey:auto
```

Infact let's SPEND the old coin and output it to the same address.

In my coins relevant:true I have only 2 coins.. The 2 minima floating one and the big one (left over from the billion). Importantly the floating 2 Minima coin is the latest - so it will be picked up first.

So - if you do (using the same address and amount)

```
send amount:2
address:0x09B9782AA11B0B1F3D658016E2FD1E120DC0619A7BCD4F4DA6A4D79F0C6A5783
```

It will spend that coin, and create another coin - with a different coinid, but with the same address, amount and tokenid, still floating.

```
coins relevant:true
```

This will show the new coin with different coinid.

We can now post the floating transaction we made, with a coin that has already been spent, and it will still work as it uses the NEW coin we just created.

```
txnpost id:floattxn auto:true
```

BOOM. You just spent a floating coin..

Full ELTOO Sequence

We now have all the pieces required to attempt a complete ELTOO transaction sequence. What we can do is create a payment channel between 2 users and update that with bi-directional payments, in a way that does not allow either User to change the expected outcome.

This is the ELTOO contract required by Minima

```
LET st=STATE(99)
LET ps=PREVSTATE(99)
IF st EQ ps AND @COINAGE GT 256 AND MULTISIG(2 0xUser1Settle 0xUser2Settle) THEN
   RETURN TRUE
ELSEIF st GT ps AND MULTISIG(2 0xUser1Update 0xUser2Update) THEN
   RETURN TRUE
ENDIF
```

The Public keys used by the users are 0xUser1Settle and 0xUser2Settle, and 0xUser1Update and 0xUser2Update

The sequence number is stored in state variable 99.

You will note the settlement clause can only be attached to a single output with the exact state sequence number. The update clause can be used as long as it has a higher sequence number - and both use floating coins.

How this works (pls read the ELTOO whitepaper to follow more closely):

1) User1 and User2 wish to open a payment channel with 10 Minima each.
2) They start by creating a setup transaction that sends funds to a simple 2of2 multisig worth 20 Minima - the **Funding** transaction - but do not sign and publish it.
3) They create a **trigger** transaction that spends the Funding transaction and sends the funds to the ELTOO contract,, with state variable 99 equal to 0. This will effectively start the final settlement sequence - since the @COINAGE timeout is now on chain
4) They create the first **settlement** transaction that spends the trigger transaction, with state variable 99 set to 0, and sends 10 Minima back to each user.
5) They sign and share the trigger and initial settlement transaction, and do the same with the funding transaction, and finally publish the Funding transaction.
6) Now - they both have a valid transaction spending the Funding transaction to create the trigger transaction, and both have an initial settlement transaction spending THAT to give them the money back. Phew.. stay with me.
7) Currently ONLY the Funding transaction has been published.
8) Both users can create new update transactions, and their corresponding settlement transactions, that allocate different amounts of the 20 Minima to each user.
9) An Update transaction simply spends their ELTOO coin, and sends the funds back to the same ELTOO coin but with a higher sequence number - in this case state 99.
10) Every Update transaction has a settlement transaction that spends it. And can append ONLY it.
11) So if User1 wants to pay User2 1 Minima - they create a new update transaction with an incremental sequence number, that spends the current ELTOO output and pays to the same ELTOO output, and a new settlement transaction paying 9 Minima to User1 and 11 Minima to User2, with the exact current sequence number as a state variable. They create the settlement FIRST - then the update (so the update can't be posted without a valid settlement).
12) This continues indefinitely for as long and as fast as the Users want (there could be more than 2 users of course)
13) When they want to close the channel - they publish the trigger transaction, then the latest update transaction, and then the latest settlement transaction.

14) IF one of the users publishes an earlier Update transaction, by mistake or maliciously, no problem.. the other user can publish their latest Update transaction on top of that one, because of the @COINAGE timeout - and because they all use floating coins!
15) And finally - as a nice optimisation - since all parties have a valid trigger, update and settlement transaction, there is no need to *actually* publish them all on chain - is there ? The final outcome is the outputs of the final settlement transactions, so the parties can negotiate a new transaction, that spends the original funding transaction, instead of the trigger transaction, and pays everyone what they are owed..

So in conclusion - the whole process only requires 1 transaction to start the channel, and 1 to close it. But - and this is important - you can run a sequence.. and NOT close at the end. Just use it again when you need to.

'Re-balancing' channels allows for the amounts currently owed to each user to be changed so that the channel need not be closed.

So if Alice and Bob have a channel, but currently ALL the funds are being sent to Bob, how can Alice continue to use it, and continue to send Bob funds ?

Alice and Bob both have channels with Claire.. Alice pays Claire, Claire pays Bob, and Bob pays (rebalances) Alice. At the end of this trick everyone still has the same total amount of funds, but the channels have been re-balanced. You would use HTLC contracts to perform this.. Which work just as well off chain as on.

Running this from the command line can be quite.. extensive.. But if we simplify this a bit.. and change it to a single signature.. We can play with the transaction script in different scenarios..

So in this simplified - just to play - version..

0xFF is the settlement key
0xEE is the update key

The script we use is :

```
LET st=STATE(99)
LET ps=PREVSTATE(99)
IF st EQ ps AND @COINAGE GT 20 AND SIGNEDBY(0xFF) THEN
   RETURN TRUE
ELSEIF st GT ps AND SIGNEDBY(0xEE) THEN
   RETURN TRUE
ENDIF
```

So - we are assuming the users sign correctly..

We can now run..

```
runscript script:"LET st=STATE(99) LET ps=PREVSTATE(99) IF st EQ ps AND @COINAGE GT 20
```

```
AND SIGNEDBY(0xFF) THEN RETURN TRUE ELSEIF st GT ps AND SIGNEDBY(0xEE) THEN
RETURN TRUE ENDIF" globals:{"@COINAGE":"23"} state:{"99":"0"} prevstate:{"99":"0"}
signatures:["0xFF"]
```

..and at least play with various scenarios. This variant returns TRUE.

State Chains

State Chains are similar to a normal ELTOO channel with a couple of differences.

1) The State Chain is run by a single entity, and it is the co-signer of any ELTOO Channel, with an additional User.
2) When the User wishes to give the Coin, which is in the ELTOO channel, to someone, they give them the Private Key of the Public Key that is used as one half of the MultiSig. A new Update and Settlement transaction is created that pays out the coin to the NEW user - a new address.
3) You can only give the WHOLE coin to another user. This works well for NFTs.
4) So every settlement transaction simply assigns the funds to a different new address controlled by the current owner.
5) When a User is given the coin the old user signs a chain of transfers. So that there is a 'State Chain' that defines who had and who was given the Coin.
6) This means any previous User + the State Chain entity can in theory write themselves a new update and settlement and take the funds - so trust is required in the state chain entity,
7) Neither the State Chain entity nor the User ever have BOTH the keys required to update the channel.
8) It is impossible to take the Coin without it being known. Since the State Chain defines who should have it. So any attempt to steal the funds is 'known'.
9) The State Chain entity does not need to know 'who' the coin is being given to. All it promises is to create a new Update and Settlement transaction when the current owner wishes it - by signing with a public key, and that a new owner can be set with a new public key when the current owner wishes it.

Yes - the trust assumptions are different to a normal ELTOO transaction - but what is really powerful about State Chains is that unlike a normal ELTOO channel - users can be onboarded onto this Layer 2 protocol WITHOUT having to do an on chain transaction. What you are doing is sending the 'coin' itself (or NFT) to layer 2 and then any user can be involved, since all you do is share the Private Key.

Coin Flip v2

And now - how does this relate to our original slow clunky expensive coin flip game ?

Well - the CoinFlip game is just a sequence of rounds. Just like an ELTOO sequence. So we start an ELTOO channel with 1 Minima from each player - as normal. Then the settlement transaction of each update, pays out to the CoinFlip script, with the latest updated state variables.. It's really that simple. If at any stage either User stops cooperating or becomes unresponsive, no problem, the latest transactions are posted and the game continues on chain. With both parties knowing that this is the case, there is no reason not to finish the game off chain.

And so the Players can play Coin Flip instantly and for free, as many times as their channel will allow before either closing the game or rebalancing, to fight another day.

Coin Flip itself is not a very complicated nor a very interesting game.. but it shows the basics of a possible structure required for any round based game. This is also not limited to 2 individuals either - there is nothing to stop more participants entering the ELTOO contract. The maximum size of the transaction coupled with the signatures is the limiting factor here.

On one side you could be writing far more complex scripts that checked chess or poker dice, and on the other you could be writing House buying contract sequences that require different parties to sign at different times, with non-refundable deposits after a certain point etc etc..

Most importantly is that you can play out different ELTOO 'scenarios' without always having to do an on-chain transaction to start or finish. Once you have an open channel you can run through any sequence as often as your channel amounts will allow - by always agreeing on the final state at the end, without the need for arbitration on Layer 1. Layer 1 arbitration should only happen when one party is forced due to technical issues to become unresponsive - otherwise there seems little point to not cooperating since the final outcome is the same.

Tokens

Minima natively supports Tokens. Tokens on Minima are created using coloured coins. So we take fractional amounts of Minima and 'colour' them so that they get a unique and individual TokenID. Tokens can be used inplace of Minima in any of the transactions we have been talking about earlier and in all the Layer 2 ELTOO protocols as well. They are treated in the same way as Minima itself, and can be used in smart contracts just like Minima, they just have a different TokenID.

Start with :

```
tokencreate name:mycoin amount:1000
```

This constructs a special transaction, and will convert a fractional amount of Minima into a Token. You can see this with 'balance'


```
..
  {
    "token":{
      "name":"mycoin"
    },

"tokenid":"0x0C5EEB946E8C18B610AFA1D9A9F7EB05688E21A72608D5F43C925D1D6804A16E
",
    "confirmed":"1000",
    "unconfirmed":"0",
    "sendable":"1000",
    "total":1000
  }]
}
```


And if you use 'tokens'


```
..
  {
    "name":{
      "name":"mycoin"
    },

"coinid":"0x8C8DB5BA960C8AC2CD3BD314BDBC280E941C2D45469C7685083D19387024CA54
",
    "total":"1000",
    "decimals":8,
    "script":"RETURN TRUE",
    "totalamount":"0.00000000000000000000000000000001",
    "scale":36,

"tokenid":"0x0C5EEB946E8C18B610AFA1D9A9F7EB05688E21A72608D5F43C925D1D6804A16E
"
  }]
}
```

The Token details tell you how many tokens there are in total, the number of decimal places - which defaults to 8, the Script for the token, and the total amount of Minima used to create the token. The total amount of Minima * the scale is the actual Token Amount. The TokenID is always globally unique.

The 'name' is just a string so it can be populated by a JSON and is by default - if you only specify a single word name. Then you can embed description, icons, links to external websites etc etc..

You can send them to anyone using the same 'send' command by constructing custom transactions. Simply specify the tokenid.

```
send address:0xFF amount:1
tokenid:0x0C5EEB946E8C18B610AFA1D9A9F7EB05688E21A72608D5F43C925D1D6804A16E
```

This will create a more descriptive token..

```
tokencreate amount:10 name:{"name":"newcoin","link":"http:mysite.com","description":"A very cool token"}
```

You can specify the number of decimals.. Which in actuality changes the scale so no decimal places are possible, and easily create non-fungible tokens.

```
tokencreate name:mynft amount:10 decimals:0
```

And now you cannot send fractional amounts of the token. Only whole numbers - good for tickets. Trying to send 1.5 as the amount will fail. If you create a token and specify amount:1 decimals:0 you would have an 'NFT'.. 1 unit, non-divisible.

Tokens can have scripts. A token script is 'RETURN TRUE' by default. Whenever you try and spend a token the address script AND the token script must return TRUE. That's all it does.

You can use this to have a counter, or to make sure a payment is made whenever the token is used, or to keep track of certain state variables and ensure they remain, or any other reason you can think of.

For instance.. If you set this as the script in a token..

```
tokencreate name:charitycoin amount:1000 script:"ASSERT VERIFYOUT(@TOTOUT-1 0xMyAddress 1 0x00 TRUE )"
```

You would be saying - make sure that in every transaction this token is used, the last output is 1 Minima sent to 0xMyAddress. Trying to 'send' it as normal will fail - see the error. You will need to construct a custom transaction that does just that.. and then it will work.

APPENDIX

Here is a breakdown of the simple and complete **KISSVM** language.

Grammar

```
ADDRESS     ::= ADDRESS ( BLOCK )
BLOCK       ::= STATEMENT_1 STATEMENT_2 ... STATEMENT_n
STATEMENT   ::= LET VARIABLE = EXPRESSION |
                LET ( EXPRESSION_1 EXPRESSION_2 ... EXPRESSION_n ) = EXPRESSION |
                IF EXPRESSION THEN BLOCK [ELSEIF EXPRESSION THEN BLOCK]* [ELSE
                BLOCK] ENDIF |
                WHILE EXPRESSION DO BLOCK ENDWHILE |
                EXEC EXPRESSION |
                MAST EXPRESSION |
                ASSERT EXPRESSION |
                RETURN EXPRESSION
EXPRESSION  ::= RELATION
RELATION    ::= LOGIC AND LOGIC  | LOGIC OR LOGIC  |
                LOGIC XOR LOGIC  | LOGIC NAND LOGIC |
                LOGIC NOR LOGIC  | LOGIC NXOR LOGIC | LOGIC
LOGIC       ::= OPERATION EQ OPERATION  | OPERATION NEQ OPERATION  |
                OPERATION GT OPERATION  | OPERATION GTE OPERATION  |
                OPERATION LT OPERATION  | OPERATION LTE OPERATION  | OPERATION
OPERATION   ::= ADDSUB & ADDSUB | ADDSUB | ADDSUB | ADDSUB ^ ADDSUB | ADDSUB
ADDSUB      ::= MULDIV + MULDIV | MULDIV - MULDIV | MULDIV % MULDIV |
                MULDIV << MULDIV | MULDIV >> MULDIV | MULDIV
MULDIV      ::= PRIME * PRIME | PRIME / PRIME | PRIME
PRIME       ::= NOT PRIME |  NEG PRIME | NOT BASEUNIT | NEG BASEUNIT | BASEUNIT
BASEUNIT    ::= VARIABLE | VALUE | -NUMBER | GLOBAL | FUNCTION | ( EXPRESSION )
VARIABLE    ::= [a-z]+
VALUE       ::= NUMBER | HEX | STRING | BOOLEAN
NUMBER      ::= ^[0-9]+(\\\\.[0-9]+)?
HEX         ::= 0x[0-9a-fA-F]+
STRING      ::= [UTF8_String]
BOOLEAN     ::= TRUE | FALSE
FALSE       ::= 0
TRUE        ::= NOT FALSE
GLOBAL      ::= @BLOCK | @BLOCKMILLI | @CREATED | @COINAGE | @INPUT |
                @AMOUNT | @ADDRESS | @TOKENID | @COINID |
                @SCRIPT | @TOTIN | @TOTOUT
FUNCTION    ::= FUNC ( EXPRESSION_1 EXPRESSION_2 .. EXPRESSION_n )
FUNC        ::= CONCAT | LEN | REV | SUBSET | GET | EXISTS | OVERWRITE |
                CLEAN | UTF8 | ASCII | REPLACE | SUBSTR |
                BOOL | HEX | NUMBER | STRING | ADDRESS |
                ABS | CEIL | FLOOR | MIN | MAX | INC | DEC | SIGDIG | POW |
                BITSET | BITGET | BITCOUNT | PROOF | KECCAK | SHA2 | SHA3 |
                FUNCTION | SUMINPUT | SUMOUTPUT |
                SIGNEDBY | MULTISIG | CHECKSIG |
                GETOUTADDR | GETOUTAMT | GETOUTTOK | GETOUTKEEPSTATE | VERIFYOUT |
                GETINADDR | GETINAMT | GETINTOK | GETINID | VERIFYIN |
                STATE | PREVSTATE | SAMESTATE
```

Globals

```
@BLOCK       : Block number this transaction is in
@BLOCKMILLI  : Block time in milliseconds
@CREATED     : Block number when this output was created
@COINAGE     : Difference between @BLOCK and @CREATED
@INPUT       : Input number in the transaction
@COINID      : CoinID of this input
@AMOUNT      : Amount of this input
@ADDRESS     : Address of this input
@TOKENID     : TokenID of this input
@SCRIPT      : Script for this input
@TOTIN       : Total number of inputs for this transaction
@TOTOUT      : Total number of outputs for this transaction
```

Functions

```
CONCAT ( HEX_1 HEX_2 ... HEX_n )
Concatenate the HEX values.

LEN ( HEX|SCRIPT )
Length of the data

REV ( HEX )
Reverse the data

SUBSET ( HEX NUMBER NUMBER )
Return the HEX subset of the data - start - length

OVERWRITE ( HEX NUMBER HEX NUMBER NUMBER)
Copy bytes from the first HEX and pos to the second HEX and pos, length the
last NUMBER

GET ( NUMBER NUMBER .. NUMBER )
Return the array value set with LET ( EXPRESSION EXPRESSION .. EXPRESSION )

EXISTS ( NUMBER NUMBER .. NUMBER )
Does the array value exists

ADDRESS ( STRING )
Return the address of the script

REPLACE ( STRING STRING STRING )
Replace in 1st string all occurrence of 2nd string with 3rd

SUBSTR ( STRING NUMBER NUMBER )
Get the substring
```

CLEAN (STRING)
Return a CLEAN version of the script

UTF8 (HEX)
Convert the HEX value to a UTF8 string

ASCII (HEX)
Convert the HEX value to an ASCII string

BOOL (VALUE)
Convert to TRUE or FALSE value

HEX (SCRIPT)
Convert SCRIPT to HEX

NUMBER (HEX)
Convert HEX to NUMBER

STRING (HEX)
Convert a HEX value to SCRIPT

ABS (NUMBER)
Return the absolute value of a number

CEIL (NUMBER)
Return the number rounded up

FLOOR (NUMBER)
Return the number rounded down

MIN (NUMBER NUMBER)
Return the minimum value of the 2 numbers

MAX (NUMBER NUMBER)
Return the maximum value of the 2 numbers

INC (NUMBER)
Increment a number

DEC (NUMBER)
Decrement a number

POW (NUMBER NUMBER)
Returns the power of N of a number. N must be a whole number.

SIGDIG (NUMBER NUMBER)
Set the significant digits of the number

BITSET (HEX NUMBER BOOLEAN)
Set the value of the BIT at that Position to 0 or 1

BITGET (HEX NUMBER)
Get the BOOLEAN value of the bit at the position.

BITCOUNT (HEX)
Count the number of bits set in a HEX value

PROOF (HEX HEX HEX)
Check the data, mmr proof, and root match. Same as mmrproof on Minima.
KECCAK (HEX|STRING)
Returns the KECCAK value of the HEX value.

SHA2 (HEX|STRING)
Returns the SHA2 value of the HEX value.

SHA3 (HEX|STRING)
Returns the SHA3 value of the HEX value.

SIGNEDBY (HEX)
Returns true if the transaction is signed by this public key

MULTISIG (NUMBER HEX1 HEX2 .. HEXn)
Returns true if the transaction is signed by N of the public keys

CHECKSIG (HEX HEX HEX)
Check public key, data and signature

GETOUTADDR (NUMBER)
Return the HEX address of the specified output

GETOUTAMT (NUMBER)
Return the amount of the specified output

GETOUTTOK (NUMBER)
Return the token id of the specified output

GETOUTKEEPSTATE (NUMBER)
Is the output keeping the state

VERIFYOUT (NUMBER HEX NUMBER HEX BOOL)
Verify the output has the specified address, amount, tokenid and keepstate

GETINADDR (NUMBER)
Return the HEX address of the specified input

GETINAMT (NUMBER)
Return the amount of the specified input

GETINTOK (NUMBER)
Return the token id of the specified input

VERIFYIN (NUMBER HEX NUMBER HEX)
Verify the input has the specified address, amount and tokenid

STATE (NUMBER)
Return the state value for the given number

PREVSTATE (NUMBER)
Return the state value stored in the coin MMR data - when the coin was created.

SAMESTATE (NUMBER NUMBER)
Return TRUE if the previous state and current state are the same for the start and end positions

SUMINPUTS (HEX)
Sum the input values of this token type

SUMOUTPUTS (HEX)
Sum the output values of this token type

FUNCTION (STRING VALUE1 VALUE2.. VALUEn)
Generic Function. Run the script after replacing $1, $2.. $n in the script with the provided parameters and use the variable 'returnvalue' as the returned result.