MMX Blockchain & Currency

January 29, 2025

Abstract

Over time, crypto-currencies have turned into gambling instruments, losing the "currency" part.

The underlying issue with fiat is not its ability to increase supply, it is banks printing their own money because they can. A useful crypto-currency will need a dynamic supply that can adjust to demand.

Put another way: If we don't mind banks and governments covertly stealing from us, there would be no point in crypto, we could just use fiat. And if the goal was to gamble with life savings, the stock market was already invented for this purpose.

MMX aims to be a true crypto-currency, with a dynamic supply governed by decentralized consensus. To keep a stable value and allow it to be used as currency, supply has to follow demand in order to prevent inflation or deflation.

Security is provided by Proof of Space, aka HDD "farming", or SSD farming in the future. It is more decentralized than Proof of Stake while being more energy efficient than Proof of Work.

However, just creating a useful token as currency is not enough. Which is why: MMX has been designed from scratch without re-using any existing blockchain code, in order to improve everything.

A high-level virtual machine allows for much easier smart contracts, while the `C++` implementation supports high transaction throughput (500+ TPS). A tailor-made database engine was developed to improve the blockchain use-case. A minimum transaction fee is enforced to avoid spam and allow it to operate on transaction fees early on.

MMX stands for MadMaX blockchain, because it goes against everything that has come so far:

1. No set-it and forget-it ponzinomics
2. No pre-mine / pre-farm
3. No VC funded development
4. No energy wasting Proof of Work
5. No false security with Proof of Stake
6. No complicated low-level VM
7. No developer pain due to bad architecture
8. No throughput limits due to poor implementation

Bitcoin scores 4 out of the above, Ethereum and its derivatives only 1 or 2, Solana only 2 and Chia Network 3.

False Crypto

Not everything that is called "crypto" is actually crypto. Let's set the record straight and provide a solid definition of "crypto":

- Tokens with centralized backing are not crypto, because they can be rugged at any point.

- Tokens with centralized blacklists are not crypto, because the coins can be stolen or seized at any point.
- Smart contracts that can be upgraded or modified are not crypto, because the terms of the contract can be changed at any point.
- Your keys your crypto, without any gotchas, is crypto.

That means the following are not crypto:

- USDT, USDC, etc
- wBTC and other wrapped crypto, due to centralized backing
- DAI, because it's mostly backed by the above
- Tether Gold, PAX Gold, etc
- Tokenized real-world assets in general

Anything with centralized control on-top of blockchain is a glorified database. The original and true purpose of crypto is a lack of centralized control.

Dynamic Supply

As stated above, a dynamic supply is needed to avoid inflation or deflation. Inflation leads to a loss of value while deflation turns the currency into a gambling instrument.

As such, the supply of MMX is governed by consensus, one block one vote. The vote is ternary, either to increase / decrease block reward or to vote neutral. A majority control of netspace is needed to change the outcome of the voting process.

The block reward is adjusted to target an MMX price of 2000 MMX per troy ounce of Gold. At the time of writing this is around $1.33 USD per MMX. Using Gold as a reference ensures a globally fair stable value, that is independent of any fiat inflation. Other references such as energy prices, stock market prices, consumer price indices have been disregarded due to geographical biases and general volatility.

Once every 24h, votes are counted and the block reward is adjusted by 1% up or down, depending on what the majority voted for. In case half or more voted neutral, nothing will happen. The minimum step size for adjustment is 0.01 MMX. Meaning, at below 1 MMX the adjustment is higher than 1%. As a result, it takes 100 days to increase the reward from zero to 1 MMX per block. Beyond that, the reward can increase or decrease by a factor of 38 per year.

To avoid farmers being able to fill blocks with spam for free, 50% of transaction fees are burned. As a secondary effect, this allows the supply of MMX to contract. If such a reduction works against the target price, it will be canceled out by an equivalent increase in block reward.

Reward Control Loop

In order for nodes to vote on increasing or decreasing the block reward, they need to have information on the current price of MMX vs Gold.

Unless MMX is traded directly against Gold, two or three inputs are needed:

- MMX price in terms of some fiat currency
- Gold price in terms of some fiat currency
- Fiat exchange rates, if MMX price and Gold price are not based on the same currency

Nodes will gather these inputs on their own from various independent but centralized sources. They then compute the current price of an ounce of Gold in terms of MMX and apply the correct vote to target 2000 MMX per ounce.

For example: MMX is currently traded at $1.10, while an ounce of Gold is trading at $2500. That means the current price of an ounce of Gold in MMX is: $2500/1.1 = 2272$. Which is above

the target of 2000, as such farmers will vote to decrease block reward, to eventually increase the price of MMX to $1.25.

To make the system work, MMX will need to be traded against fiat somewhere. Either on an exchange or against a stable coin on the blockchain itself. The node software will continuously be updated to include all possible sources, which doesn't require a soft or hard fork. Until MMX is traded against fiat or Gold directly, the reward control loop is disabled.

The reward control loop does not directly or swiftly affect the price of MMX. It is more of a backstop, however traders should know it is there. As a result, it is expected to control the price more directly through anticipating the outcome of deviations from the target price.

In other words: Buyers would be foolish to buy above target, while sellers would be foolish to sell below target.

Minimum Block Reward

There is a minimum block reward of 0.5 MMX to incentivize early security of the blockchain. However, transaction fees are subtracted from it in a time-averaged fashion.

As such the formula to calculate the average block reward is:

$$B_{reward} + max(0.5 - T_{fees}, 0)$$

Where B_{reward} is the result of voting, starting at zero, and T_{fees} is the average total transaction fees per block.

Individual block rewards depend on recent transaction activity: Transaction fees are added up into a buffer, while at the same time a certain percentage $\frac{1}{8640}$ is taken out and subtracted from the reward. The minimum subtraction per block is 0.001 MMX. As such, subtracting a single standard transfer is spread out over 50 blocks. The reason for this averaging mechanism is to ensure farmers are still incentivized to include transactions.

Once transaction fees per block are more than 0.5 MMX, no new coins are created through the minimum reward scheme. At this point only block reward voting can cause new coins to be issued. However in effect, when combined with the 50% fee burn, the threshold is actually 0.333 MMX.

As a result, the price target can only be reached once demand is high enough to cover the minimum reward emission. Or once chain activity is high enough to cancel the minimum reward, which would be one standard transfer per second.

Minimum Security

At the time of writing, netspace on testnet has reached 150 PB. As such, it is expected for mainnet to start with at least 150 PB as well. Assuming an average of $20 per TB, this is equal to $3M in security. Meaning it would take $3M to attack the network, not counting labor and plotting cost.

Renting cloud storage for such an attack is expensive as well. Assuming a plotting time of 2 months, it would cost $2M to perform a 150 PB attack using Google Cloud, not counting compute costs.

Naturally it appears plotting faster would be cheaper. In order to plot 150 PB in 24h, one would have to rent 125,000 A100 GPUs. Assuming a spot price of $1 per hour it would cost $3M, not counting storage. In practice one cannot rent such large quantities, so the cloud rental attack would cost at least $5M for 150 PB.

Testnet Rewards

Starting with testnet8, farming on testnet has mostly been incentivized with 0.5 MMX per block. This is estimated to be around 3,300,000 MMX at the start of mainnet, which will be released on genesis.

This will most likely cause the price to be below target initially. It depends on the price farmers are willing to sell their initial MMX, which is hard to predict. Assuming these coins to be sold below target at $0.90 per MMX, it would equal $3M USD in total. That would leave potential buyers a 50% profit margin, once it reaches the target price.

The total amount of testnet rewards is equal to around two years of minimum reward. While two years might sound like a lot, it is less than the amount other blockchains mint per month, week or even day. It is called the minimum reward for a reason.

Minimum Transaction Fee

A minimum transaction fee is necessary to avoid spam attacks on high throughput blockchains. Without it security is bought through inflation, which cannot last forever. A higher fee can be paid to increase a transaction's priority.

The minimum fee for a standard transfer is 0.05 MMX. Made up by 0.02 for a transaction, 0.01 per input/output and 0.01 per signature.

A block is divided into five fee bands: 1x, 2x, 3x, 5x and 10x. To fill the first 20% of a block requires just the minimum fee. Filling the next 20% requires double the min fee, and so on. This allows to finance higher network bandwidth and faster database growth when needed.

Proof of Space

Proof of Space in MMX is somewhat similar to Chia Network:

- MMX uses 9 tables, which yields 256 X values per proof.
- Plot k-size is limited to between k29 and k32, all inclusive.
- The hash function for table 1 has been custom designed to be on-chip memory hard, requiring around 4 KiB per instance, with a memory bus of 1024-bit. The output (and internal state) of the custom hash is 1024 bits, which is hashed with SHA2-512 to produce a table entry M_i.
- The hash function for the remaining tables has been changed to: $SHA2_{512}(M_{left}|M_{right})$
- The matching function has been simplified to: $Y_{left} + 1 = Y_{right}$
- A table entry consists of a 448-bit value ($14k$ bit, truncated SHA2-512).
- The final table output is a 384-bit value ($12k$ bit, truncated SHA2-512).
- The k-bit Y value of an entry is derived as a XOR checksum over the 448-bit value.
- The plot filter has been reduced to 16, and will not change over time.
- On lookup, a set of 16 proofs is obtained on average, which is a sequential disk read. As such the effective plot filter is 1.
- Proof quality is computed via: $SHA2_{256}(C_{256}|M_{384})$, where C_{256} is the plot challenge and M_{384} is the final $12k$ bit output hash.
- There are two different plot types, one for HDDs and one for SSDs.
- HDD plots store the output hash to reduce IOPS by a factor of 2048, making them 2.5 times bigger than SSD plots.
- SSD plots have a much higher IOPS and compute load, since they need to lookup 256 X values and recompute the output hash (one proof per plot per challenge on average).
- Plotting is performed using GPUs, since CPU plotting would be too inefficient.
- Plots can optionally be compressed by up to 3% before it starts to become economically ineffective.

Developer Team

- madMAx (Main Dev, GitHub: madMAx43v3r)
- AndrejsSTX (Windows Port, Windows GUI, WebGUI, GitHub: stotiks)
- voidxno (Fast SHA2-256, SHA-NI VDF, GitHub: voidxno)

Project Funding

There is a 1% fee on transaction fees to fund continued development of MMX. There is no fee on the actual amount being transferred, like with PayPal, VISA or MasterCard.

No new coins are created to fund development, however the amount is taken from the burned 50% portion of the transaction fees.

Because 1% is very little at low transaction volume, up to 0.05 MMX is taken as project fee per block, but never more than half the total transaction fees.

The formula is:

$$min(0.05 + \frac{T_{fees}}{100}, \frac{T_{fees}}{2})$$

Note: Testnet rewards go to farmers who participated in testnets. Not to developers, who own less than 1% of the netspace.

Blockchain

MMX has a deterministic block interval of 10 seconds, governed by a recursive SHA-2 Verifiable Delay Function.

On average 4 proofs of space are generated per block, but only the best will be used to create a block. Difficulty adjustment is enforced by consensus, to avoid a competing chain running a higher difficulty to appear heavier.

Proofs are exchanged up to 50 seconds in advance, preventing multiple competing blocks to be created in most cases. In case no proof is found the next block will be delayed by another 10 seconds. This happens in around 1 out of 100 challenges.

The maximum block size is set to allow 500 standard transactions per second, which is around 2.5 MB per block. This is equivalent to a Bitcoin block size of 150 MB. A standard transaction is a simple transfer from a single address to another address (1 input, 1 output).

MMX is using the account model rather than UTXO, similar to most modern blockchains. Bitcoin's ECDSA signatures are used to sign transactions and blocks.

Transactions are processed in parallel when possible, using random nonces. A single MMX transaction can execute multiple functions sequentially, hence there is no need for sequential nonces. The execution of multiple calls to the same contract are serialized, since contracts have a mutable state with complicated logic.

The only constraint is that a smart contract can only call contracts that have been defined at deploy time. As such reentrancy is impossible by design.

By default, transactions expire if they are not included within 100 blocks of their creation. This ensures a second transaction with higher fee can be sent later without the risk of both executing.

Challenge Generation

When using Proof of Space, challenges for proof generation need to be immutable. Otherwise the effective space can be multiplied easily by trying different inputs to create different challenges, turning it into proof of work. On the other hand, if challenges are fully immutable they can be predicted into the future indefinitely, causing a security concern.

The solution is a trade-off where future challenges are periodically updated through the infusion of data which cannot be predicted. If no such infusion is made, a new challenge is created deterministically, by hashing the previous challenge.

In MMX this is achieved by randomly infusing the hash of a block's proof, every 256 blocks on average. The condition to infuse is itself based on the proof hash. As such, any single farmer

cannot predict when and how future challenges will be updated, until the time such a proof is found or received.

Naturally this approach is not perfect. For a long range attack where the attacker controls the entire netspace, a weight gain of 2% can be achieved. Due to being able to predict future challenges and select updates that yield more proofs.

All proofs found are included in blocks, while all secondary proofs have to be worse than the proof which makes the block (higher proof hash). Space difficulty is adjusted to target 4 proofs per block on average. Whenever the challenge chain is updated, the difficulty is updated as well.

It should be noted that proofs of space in MMX are not malleable, the ordering of X values is enforced to follow a unique collision-free sort. A proof's hash also includes the challenge used to look it up, as such they are not predictable.

Earlier versions of MMX used an approach similar to Chia Network, where challenges are produced by a VDF. However such a VDF generation cannot be rewarded, hence this approach was disregarded. It would also have required to verify some or all VDFs during sync, as the proofs of space would be useless without it.

Timelords

Timelords compute a Verifiable Delay Function (VDF) that governs the interval between blocks. VDFs are sequential in nature to proof that a certain amount of real time has passed.

A single honest timelord is sufficient for the operation of the network. It only requires one CPU core to run. The number of iterations per height is adjusted to yield the desired real time interval between blocks.

The VDF in MMX is simply the recursive SHA2-256 hash of an input, with occasional infusions of extra data, such as recent block hashes. Modern CPUs have special silicon (SHA-NI) for computing SHA-2 hashes, which are state of the art, making most CPUs a timelord ASIC for MMX.

The downside of using SHA-2 is that either a small GPU, or a modern mid to high-end CPU is needed to run a node. But that is an acceptable compromise to avoid having to spend a few $M to develop a custom ASIC. Each new generation of hardware is expected to make verifying VDFs easier, since parallel verification is easier to scale than single core clocks.

The VDF stream always infuses the previous block hash, at the beginning of a new VDF section. The reason for not using the peak block hash is to prevent latency between farmers and timelords to induce an extra delay in block production.

Without this infusion, a faster timelord could run ahead into the future indefinitely. Allowing a farmer to use future challenges to his advantage, by front-running other farmers and dumping his blocks with potentially lesser proofs onto them.

However with the infusion an attacker can only run one block ahead. He would have to dump a chain of at least two blocks in order to give the first block with lesser proof a chance to be accepted. In order to achieve this he would have to win many blocks in a row, to give his timelord enough time to run one more block ahead.

Assuming the attacker owns 50% of netspace and his timelord is 10% faster, it would only yield an effective gain of 0.01%. With a 25% faster timelord it starts to become effective, giving a gain of 1.5%. However it is unlikely for such an advantage to stay private for long.

Each timelord also infuses their reward address, proofing that they did compute it. The fastest timelord will get rewarded by 0.01 MMX per block. This reward is from newly minted coins, but since it is relatively small, it should not impact the price control loop much.

When making a block, farmers choose the VDF output which they received and verified first. They could delay block production and wait for their own timelord, at the risk of their block getting orphaned. But such an approach would only make sense if a farmer controls a larger percentage of netspace. Otherwise the investment into a slower but fast enough timelord would not pay off. At the same time, getting an extra 0.01 MMX per block would not make a big difference for such a farmer.

As such it is expected that most of the timelord rewards will go to operators with specialized setups: Using the currently best CPU on the market, highly overclocked, potentially nitrogen cooled. If a timelord achieves total domination, he would earn 86 MMX per day or 2592 MMX per month.

This will be difficult however, since high overclocks are usually not stable. Once a timelord crashes it will be difficult to catch up again, due to network latency and their own VDF verify delay. In order to start over a timelord first needs to receive the latest block, verify the VDF for it and only then start their timelord with the new input. They could bypass verification, however that opens them up to an attack from other timelords sending invalid blocks.

In the best case, the latency to receive a block is around 100 ms. While the fastest VDF verification is currently around 200 ms, using 40 series NVIDIA GPUs with high clocks. That means a timelord trying to gain the lead would have to be around 3% faster.

An attacker with a faster timelord would gain a proportional advantage when trying to double spend. Because of that it is important to reward honest timelords, in order to close the gap as much as possible.

Validators

Unlike proof of work, proofs of space do not depend on transactions included in a block. As a result a farmer is free to create and double sign as many blocks as he wants using the same proof.

This mostly affects the security of recent blocks, since they can always be reverted at any point, depending on how many blocks the attacker wins in a row. MMX does not use block receive time as a condition for selecting a peak, since that can split the chain into many forks temporarily, when multiple blocks are sent at the same time.

Instead, if multiple forks are competing with equal weight, the lowest peak block hash wins. This will always lead to consensus between nodes as long as every node receives every block.

In order to improve the security of recent blocks and prevent double signing, up to 33 "validators" are selected from recent block winners. Starting with the block at 24 heights prior the block to be voted for, walking backwards to find 33 unique keys.

These validators are farmers, who will sign a message containing the hash of the first block received which extends the main chain with the best known proof. They will not vote for a second block that has the same previous block. Unless their vote was for a block that was orphaned and 10 seconds have passed. This prevents vote fragmentation in case a farmer double signs.

Votes made for blocks with weaker proofs are rejected, to prevent a near 50% attack from being able to amplify itself.

Votes are collected by each node, verified and stored temporarily. They are not committed to blockchain history, as they are only used to decide between recent blocks. Voting does not require any tokens, it is purely based on blocks won, every vote is equal.

Votes start to take effect at a height 24 blocks prior to the current peak, called "root" height. From there votes are added up for all competing forks.

The peak selection logic is now as follows:

1. Heaviest peak wins, but only if root weight is *greater* as well. As such, this condition only applies to forks deeper than 24 blocks. If root weight is equal *or* peak weight is greater equal continue:
2. Highest total vote count wins. If equal continue:
3. Longest chain wins. If equal continue:
4. Peak with lowest block hash wins

Note: Weight is calculated as the sum of the product of time and space difficulty.

As can be seen, votes overrule everything else starting at the root height. This provides the most security, since farmers can always double sign or choose from multiple proofs found when possible. It also matches what users expect: The passage of time improving security of confirmed transactions.

As a result, once a new peak has received the majority of votes it cannot be reverted anymore. In practice, this has been observed to take around 1 second. It is expected to increase slightly with a larger network of nodes.

An attacker may get lucky at times to gain control over the majority of votes without having a clear majority of netspace. But this is a fundamental problem that affects any blockchain. The solution is to wait for enough confirmations depending on the value of a transaction.

During the sync process votes are not available, so the peak selection falls back to using chain weight. Once the first votes are received after sync, the peak will quickly update to match current consensus.

Votes are easy to verify and are of small size. Hence the overhead for this improvement is minimal.

Tokens

Tokens in MMX are native to the blockchain, with MMX itself just being one of them. A smart contract needs to be deployed to mint tokens, with the address of the contract being the token identifier. The MMX token itself is denoted by the zero address (zero hash) and does not have a contract.

When sending tokens (or MMX), the address of the token contract is specified for each input and output. The contract itself does not manage user balances. As such, token balances have the same security as native MMX. There is no possibility for a contract to steal user funds from a regular wallet.

Because tokens are native, approvals as in the EVM ecosystem are not needed. The only way to transfer tokens is via direct deposits from the user wallet to a contract, apart from normal transfers from wallet to wallet.

While it is possible to implement an ERC-20 equivalent on MMX, there would only be disadvantages in doing so. The only reason would be adding a centralized blacklist, but such contracts are not welcome on MMX.

Note: The amount of a single mint operation is limited to 80-bit, while balances and transaction inputs/outputs are limited to 128-bit.

Transactions

Transactions in MMX support multiple atomic operations. For example all of the following is possible within a single transaction:

- Send different tokens from many addresses to many destinations (similar to UTXO)
- Execute multiple operations on many smart contracts (sequentially)
- Deploy a single contract

When deploying a contract, the transaction ID itself will become the address of the contract. It is possible to execute functions of the deployed contract in the same transaction, by using the zero address.

The transaction fee can be paid by any wallet, a separate sender address is dedicated for this purpose. Each contract call can have its own "sender" or user, with a separate signature. A 64-bit random nonce is used to generate unique transaction IDs.

Failed transactions are recorded on the blockchain as long as the sender can pay for the transaction fee. This is to avoid free spamming of the network. Nodes will not forward transactions when the sender is out of funds, taking the whole mempool into account. It also gives clarity to users that something went wrong, instead of their transaction never confirming. For example, in case a trade failed due to higher than accepted slippage, or an offer was already accepted by someone else.

The ordering of transactions is enforced according to fee multiplier, and then by transaction ID.

Smart Contracts

A new high-level Virtual Machine has been developed for MMX, to enable easy contract development at low execution costs.

Instead of a low-level VM which deals with bytes and byte level memory addresses, the MMX high-level VM deals with variables of dynamic type. Essentially the VM is tailor made for JavaScript like languages, with "MMX script" being a restricted subset of JavaScript. `https://github.com/madMAx43v3r/mmx-node/blob/master/docs/MMX_script.md`

On average, one or two operations are generated per line of source code. As a result, the execution bottleneck is usually random database IO, rather than CPU cycles. For most applications, VM compute cost will be minimal while IO and storage are expensive.

The VM memory model is unified, similar to modern operating systems, which allows for transparent state updates and persistence. The address range is divided into sections. One for constant data, one for the stack, one for globals, and one for the heap. All variables in the global and heap sections are automatically persisted on disk. Reference counting is used to garbage collect any temporary data, before it would be persisted.

A single address can hold a value of varying type, which are the following:

- Null
- Boolean (true, false)
- Unsigned Integer (256-bit)
- String (UTF-8)
- Binary (same as String internally)
- Array (of varying types)
- Map (of varying types, objects are maps with string keys)
- Reference (64-bit address pointer, required for heap allocations)

There are no floating-point types and no signed integers. Those are simply not needed for blockchain applications. Integer overflows cause execution failure by default. Fixed point integer arithmetic can be used to work with factions.

In comparison to low-level VMs the MMX VM and compiler together are only 5k lines of code. Less code equals less problems.

OP codes in MMX are "wide". Meaning they can take many arguments, similar to how GPUs work. Arguments usually include source and destination address. An instruction can have up to four 32-bit arguments, with 8-bit of flags. Heap variables are always referenced by a 64-bit reference on the stack or global section, with the heap starting at address 2^{32}. Arrays and maps are usually allocated on the heap, since nesting is only possible with references.

Examples of OP codes are:

- OP_COPY: Copy a variable from one address to another (can make a deep copy of arrays and maps)
- OP_CLONE: Copy a variable from one address to a new heap allocated location (this is the only way to allocate heap space)
- $OP\_JUMP(I/N)$: Jump to another program address, optionally with boolean condition (I=if, N=if-not)
- OP_CALL: Call a subroutine with a specified number of arguments on the stack, return value is left before arguments
- $OP\_(ADD/SUB)$: Add / subtract two numbers and store result at a new location
- $OP\_(MUL/DIV)$: Multiply / divide two numbers and store result at a new location
- $OP\_(NOT/XOR/AND/OR)$: Bit-wise integer or boolean operations (depending on flag)
- OP_SHL/SHR: Bit-wise integer shift operations
- $OP\_CMP\_(EQ/NEQ/LT/GT/LTE/GTE)$: Comparison functions between two variables, storing the resulting boolean at a new location
- OP_SIZE: Copy the size of a variable as an integer to a new location

- OP_GET: Copy the element of an array or the value of a map entry to a new location, given an array index or map key
- OP_SET: Store a value at a given array index or set/over-write the value for a given map key
- OP_ERASE: Delete a map entry for a given key
- OP_PUSH_BACK: Append a value to the end of an array
- OP_POP_BACK: Pop a value off the back of an array
- OP_CONV: Convert a value to a different type (many features / flags)
- OP_CONCAT: Concatenate two strings and store the result at a new address
- OP_MEMCPY: Copy a section of the source string as a new string to a target destination
- OP_SHA256: Compute the SHA2-256 hash of a (binary) source string
- OP_VERIFY: Verify an ECDSA signature given a message hash and public key
- OP_SEND: Send tokens of a specified currency from the contract to an address
- OP_MINT: Mint new tokens of the contract to an address
- OP_RCALL: Call a function of another contract
- OP_FAIL: Fail the execution for a certain reason

The MMX script compiler is quite simple, with almost no optimizations needed, since the VM does most of the work. For example, a single OP_CLONE can store an entire object containing many fields with nested arrays and maps on disk. Global variables are simply persisted between contract invocations, without any compiler generated magic.

Smart contract code is stored once on-chain and can be reused by multiple contracts. This makes deploying contracts much cheaper, since the binary code is usually the biggest cost.

AMM Swap

MMX has a built-in swap similar to UniSwap:

- Anyone can deploy a new swap, there is no centralized web frontend.
- Each swap consists of 4 separate liquidity pools with different fee tiers: 0.05%, 0.25%, 1% and 5%.
- A trade is split into multiple iterations, each time choosing the best pool to trade with, considering price and fee percentage. As such the resulting trade fee is usually a linear combination of two fee tiers.
- A single wallet address is limited to providing liquidity to only one of the fee tiers.
- It is possible to switch fee tiers once every 24h.
- Liquidity is locked for 24h after it has been added.
- It is possible to provide any ratio of tokens as liquidity, even one-sided.
- There is no liquidity pool token, the contract keeps track of everything internally.
- There is no protocol fee, just transaction and pool fees.

Because MMX has native support for tokens, every interaction with the AMM swap is a single transaction. There is no need to send approval transactions in advance. Adding liquidity is a single transaction, vs 3-5 on EVM based blockchains.

Note: The actual implementation is a regular smart contract: `https://github.com/madMAx43v3r/mmx-node/blob/master/src/contract/swap.js`

Offer Market

In addition to AMM swaps, MMX has built-in offers that allow trading at a fixed price.

The maker deploys an offer contract for his chosen price point, and funds it with a bid amount, all in one transaction. Takers can then trade any portion of it, even multiple trades in the same block are possible.

The ask deposits are accumulated in the offer contract and can be withdrawn by the maker at any point. This allows for more efficient trading and avoids spamming the maker's wallet.

A maker can cancel, re-fill or increase their offer at any time. Even the price can be updated.

Typically offers would be used to trade larger amounts at less optimal price points for the taker, but with zero slippage. As such, through on-chain arbitrage they are a form of backstop to AMM markets.

Note: The actual implementation is a regular smart contract: `https://github.com/madMAx43v3r/mmx-node/blob/master/src/contract/offer.js`

Why no Rust

MMX has been developed in `C++11` due to the main developer being experienced with it. A framework `VNX` is used to gain safety of execution in multi-threaded environments, due to mutable data never being shared in the first place. Only in limited cases data is shared and protected by manual synchronization, for performance reasons.

The entire codebase has only two counts of using `new` and `delete`. Instead `std::shared_ptr<const T>` is used extensively, which is the same as `Arc<T>` in Rust.

Data parsing code has been fuzz tested. Recursive functions have depth limits. Almost no external libraries are used, as in most cases they are the root of all evil.

The complete list as follows:

- `secp256k1`: Bitcoin's ECDSA signatures
- `llhttp`: HTTP parser from Node
- `libbech32`: Small library to parse bech32 addresses
- `url-cpp`: Small library to parse URLs
- `lexy`: Parser generator used in MMX script compiler
- `uint256_t`: Unsigned 256-bit integer arithmetic. Has been forked and fixed.

All other dependencies have been developed by the team themselves.

In combination with defensive programming and unit testing, a crash and bug free software is created nevertheless. Without the syntactic complexity of Rust.

Road Map

- Web Wallet (similar to MetaMask)
- Hardware wallet support (for example Trezor / Ledger)
- E-commerce plug-ins (similar to PayPal)

Links

- Repository: `https://github.com/madMAx43v3r/mmx-node`
- Website: `https://mmx.network/` (soon)
- Discord: `https://discord.gg/BswFhNkMzY`
- RPC Block Explorer: `https://explore.mmx.network/`
- Fancy Block Explorer: `https://mmxplorer.com/`