We recently (finally…) launched our stablecoin USM (which I've written about here before), and I thought I'd briefly list some of the juicier questions/novelties-to-me that came up during the ~15 months of developing it from an idea to a deployed smart contract. Many of the lessons learned are relevant to other projects having nothing to do with USM.

(I'm definitely

not a Solidity expert, this was all quite new to me, though I do have pre-blockchain coding experience. Don't take any of this as gospel, just my 2c up for discussion, and I apologize if some of these are obvious to the more seasoned smart contract devs among y'all.)

1. Immutable contracts?

I decided from early on to make the smart contracts immutable, on the principle that the crudest way to ensure decentralization/prevent abuse of power is to just "throw away the keys". This is a long topic (the downside of giving up the ability to fix bugs kinda speaks for itself…) but I think it's a model worth exploring further, especially for relatively simple, minimalist projects like ours. (It should be obvious that there are many projects where deploying immutably isn't practical.) Uniswap and Tornado.cash are two other projects I know of deployed this way, I'm sure there are many more.

1. Immutability → versioning.

If you can't change your code, then your options are either a) instant total ossification or b) release new versions (while leaving the old one running) and hope people migrate to them. (Again, we're following in Uniswap's footsteps here.) We considered releasing our token with symbol "USM1" rather than "USM", to emphasize that we expect to release future versions like "USM2", not fungible with v1. Similarly, we released "v1" (really "v1-rc1" - still a "release candidate" for now), not "v1.0", because v1.0 could suggest that it might be upgraded to a compatible ("non-breaking") v1.1. But v1 can never be upgraded: only perhaps followed by a v2.

1. Simple sends (eg MetaMask) as UI.

The essence of USM is you either give some ETH and get back some USM, or vice versa. Rather than build a real UI, for now we opted to just make the contract process sends

of ETH or USM as operations: if you send it ETH, it will send back some USM in the same transaction, and vice versa (see [USM.receive()

](https://github.com/usmfum/USM/blob/a0de09203d6d8929a23c311cc41f3be48898c836/contracts/USM.sol#L157)/[USM._transfer()

](https://github.com/usmfum/USM/blob/master/contracts/USM.sol#L169)). This is a little risky (presumably some users will send to the wrong address…) but I gotta say, it's addictive to be able to do complex ops via just a send in MetaMask!

(We even implemented a slightly wacky scheme for specifying limit prices via the least significant digits of the sent quantity, though that may be a tad overengineered…)

1. Preventing accidental sends of (eg) v2 tokens to the v1 address.

One pitfall of the op-via-send approach is, supposing in the future we release a v2 contract, a very natural user error will be to send USMv2 tokens to the USMv1 contract, or vice versa (or for v3, etc): naively this would irretrievably destroy the tokens. We tried to mitigate this risk via [OptOutable

](https://github.com/usmfum/USM/blob/a0de09203d6d8929a23c311cc41f3be48898c836/contracts/OptOutable.sol), a mechanism for letting contracts tell each other "Reject sends of your token to my address." There may be better ways to handle it, but anyway this user hazard is worth mitigating somehow.

1. Uniswap v3 oracles.

USM uses the Uniswap v3 TWAP oracle infra and I can strongly recommend it: ourcode using the v3 oracles is much

simpler (~25 lines excluding comments) and better-UX than the code we were going to resort to for the v2 oracles. (This is one small reason I'm glad we launched in October, rather than last January as originally planned…) The v3 oracles still seem quite hot-off-the-presses (one annoyance is lack of support thus far for Solidity 0.8), so some users may want to wait till they're more battle-tested, but I think their fundamental design is fantastic. I believe Uniswap are also working on some further v3 oracle helper libs - if they do, definitely use those rather than code like ours.

1. MedianOracle

.

Fundamentally, USM uses a "median of three" price oracle: median of (Chainlink ETH/USD, Uniswap v3 ETH/USDC TWAP, Uniswap v3 ETH/USDT TWAP). There are various circumstances in which this could go down in flames but given some of the clunkier/riskier alternatives we considered, I'm pretty happy with it. You're welcome to use or adapt the oracle contract yourself (eg just call latestPrice()

): like USM, the oracle is immutable and since the Uniswap pairs are too, in theory it should run forever. (But please keep in mind that this is still relatively un-battle-tested code, caveat emptor! Also keep an eye on @usmfum on Twitter for news of urgent bugs, re-releases etc.)

1. Gas-saving hack: inheritance instead of composition

.

[MedianOracle

](https://github.com/usmfum/USM/blob/a0de09203d6d8929a23c311cc41f3be48898c836/contracts/oracles/MedianOracle.sol) inherits

from three oracle classes (including [UniswapV3TWAPOracle

](https://github.com/usmfum/USM/blob/a0de09203d6d8929a23c311cc41f3be48898c836/contracts/oracles/UniswapV3TWAPOracle.sol) and, uh, [UniswapV3TWAPOracle2

]
(https://github.com/usmfum/USM/blob/a0de09203d6d8929a23c311cc41f3be48898c836/contracts/oracles/UniswapV3TWAPOracle2.sol)). The much more natural design would be to give it three member variables holding the addresses of three separate oracle contracts and call latestPrice()

on each of them: but that would mean three calls to external contracts, which eats a lot of gas. So to save gas, we instead have a single contract that implements all three oracle classes + MedianOracle

itself. See the code for the gruesome details.

We drew the line at combining USM

and MedianOracle

into a single contract (just too gross, though would have saved a bit more gas). We also kept USM and FUM (the other ERC-20 token in the USM system) discrete contracts: there may be some cunning way to make a single contract implement two distinct ERC-20 tokens, but again that exceeded our grossness/cleverness threshold.

1. ls = loadState()

, pass around ls

(the loaded state), _storeState(ls)

.

The main purpose of this pattern is to avoid loading state variables repeatedly in the code, since those loads are pricey in gas terms. Instead we load once at the top-level start of each operation, and pass around the state as an in-memory

struct, then call [_storeState(ls)

](https://github.com/usmfum/USM/blob/a0de09203d6d8929a23c311cc41f3be48898c836/contracts/USM.sol#L290) at the very end to write any modified elements.

Another benefit of this pattern is, since the stored format is only accessed in two places (loadState()

and _storeState()

), those two functions can get quite cute in how they pack the bits. In our case we store two timestamps, two prices, and an adjustment factor (all to reasonable precision) in a single 256-bit word ( the StoredState

struct). By contrast, the unpacked LoadedState

struct that's actually used by all the other functions is much more legible (all 256-bit values) and intuitive.

1. Don't store ETH balance in a separate variable.

It's a simple thing, but we originally had an ethPool

var that we updated to track the total amount of ETH held in the contract. This was redundant: just use address(this).balance

. (Which we call once, in loadState()

.)

1. WAD math everywhere.

Fixed-point math sucks, but one way to make it suck even harder is to try to do math on a bunch of different vars all storing different numbers of decimal/binary digits - multiplying a 1018

-scaled number, by a 296

-scaled number, divided by a 1012

-scaled number… We just store everything as wads

, ie, with 18 decimal digits (123.456 stored as 123,456,000,000,000,000,000). When we encounter numbers scaled differently (eg from our price sources, Uniswap and Chainlink), we immediately rescale them to wads

. I think this avoided a lot of scary little oopsies.

1. Logarithms/exponents on-chain.

USM needs to calculate some exponents so we used some clever/hairy math ([WadMath

](https://github.com/usmfum/USM/blob/a0de09203d6d8929a23c311cc41f3be48898c836/contracts/WadMath.sol)), partly adapted from various StackOverflow threads, partly from way-over-our-heads mathemagic from the brilliant ABDK guys. This was all pretty scary and I dearly hope good standard libs emerge (maybe @PaulRBerg's PRBMath?) to spare amateurs like us from wading into these waters.

1. Put convenience/UI/view-only functions in their own stateless contracts, separate from the key balance-changing contracts.

We kept the core, sensitive transactional logic in the USM

and FUM

contracts, and carved out peripheral dependent logic into separate contracts with no special permissions: [USMView

](https://github.com/usmfum/USM/blob/a0de09203d6d8929a23c311cc41f3be48898c836/contracts/USMView.sol) for those (eg, UIs) that just want to grab handy view

stats like the current debt ratio, and [USMWETHProxy

](https://github.com/usmfum/USM/blob/a0de09203d6d8929a23c311cc41f3be48898c836/contracts/USMWETHProxy.sol) for users who want to operate on WETH rather than ETH. This is especially important for an immutably-deployed project like USM: if it turns out there's a bug in USMView

/USMWETHProxy

, we can fix it and redeploy them without needing to redeploy the key ETH-holding USM

/FUM

contracts.

1. Fergawdsake mark your immutable vars as immutable

.

This is the easiest way to save a considerable chunk of gas and we almost missed a couple…

May think of more… Big thanks to Alberto Cuesta Cañada and @alexroan for guiding me on my smart contract journey! I learned a shitload, for the first time in years honestly.