English 🇬🇧

Frax: Fractional-Algorithmic Stablecoin Protocol

A new category of decentralized stablecoin with a novel mechanism

Frax is the first fractional-algorithmic stablecoin protocol. Frax is open-source, permissionless, and entirely on-chain – currently implemented on Ethereum and other chains. The end goal of the Frax protocol is to provide a highly scalable, decentralized, algorithmic money in place of fixed-supply digital assets like BTC.

Frax is a new paradigm in stablecoin design. It brings together familiar concepts into a never before seen protocol:

- **Fractional-Algorithmic** – Frax is the first and only stablecoin with parts of its supply backed by collateral and parts of the supply algorithmic. This means FRAX is the first stablecoin to have part of its supply floating/unbacked. The stablecoin (FRAX) is named after the "fractional-algorithmic" stability mechanism. The ratio of collateralized and algorithmic depends on the market's pricing of the FRAX stablecoin. If FRAX is trading at above $1, the protocol decreases the collateral ratio. If FRAX is trading at under $1, the protocol increases the collateral ratio.
- **Decentralized & Governance-minimized** – Community governed and emphasizing a highly autonomous, algorithmic approach with no active management.
- **Fully on-chain oracles** – Frax v1 uses Uniswap (ETH, USDT, USDC time-weighted average prices) and Chainlink (USD price) oracles.
- **Two Tokens** – FRAX is the stablecoin targeting a tight band around $1/coin. Frax Shares (FXS) is the governance token which accrues fees, seigniorage revenue, and excess collateral value.
- **Crypto Native CPI** – Frax's end vision is to build the first crypto native version of the CPI called the Frax Price Index (FPI) governed by FXS holders (and other protocol tokens). FRAX is currently pegged to USD but aspires to become the first decentralized, permissionless native unit of account which holds standard of living stable.

Website: https://app.frax.finance
Telegram: https://t.me/fraxfinance
Telegram (announcements / news): https://t.me/fraxfinancenews
Twitter: https://twitter.com/fraxfinance
Medium / Blog: https://fraxfinancecommunity.medium.com/
Governance (discussion): https://gov.frax.finance/
Governance (voting): https://snapshot.org/#/frax.eth

Whitepaper (Core) - Frax v1

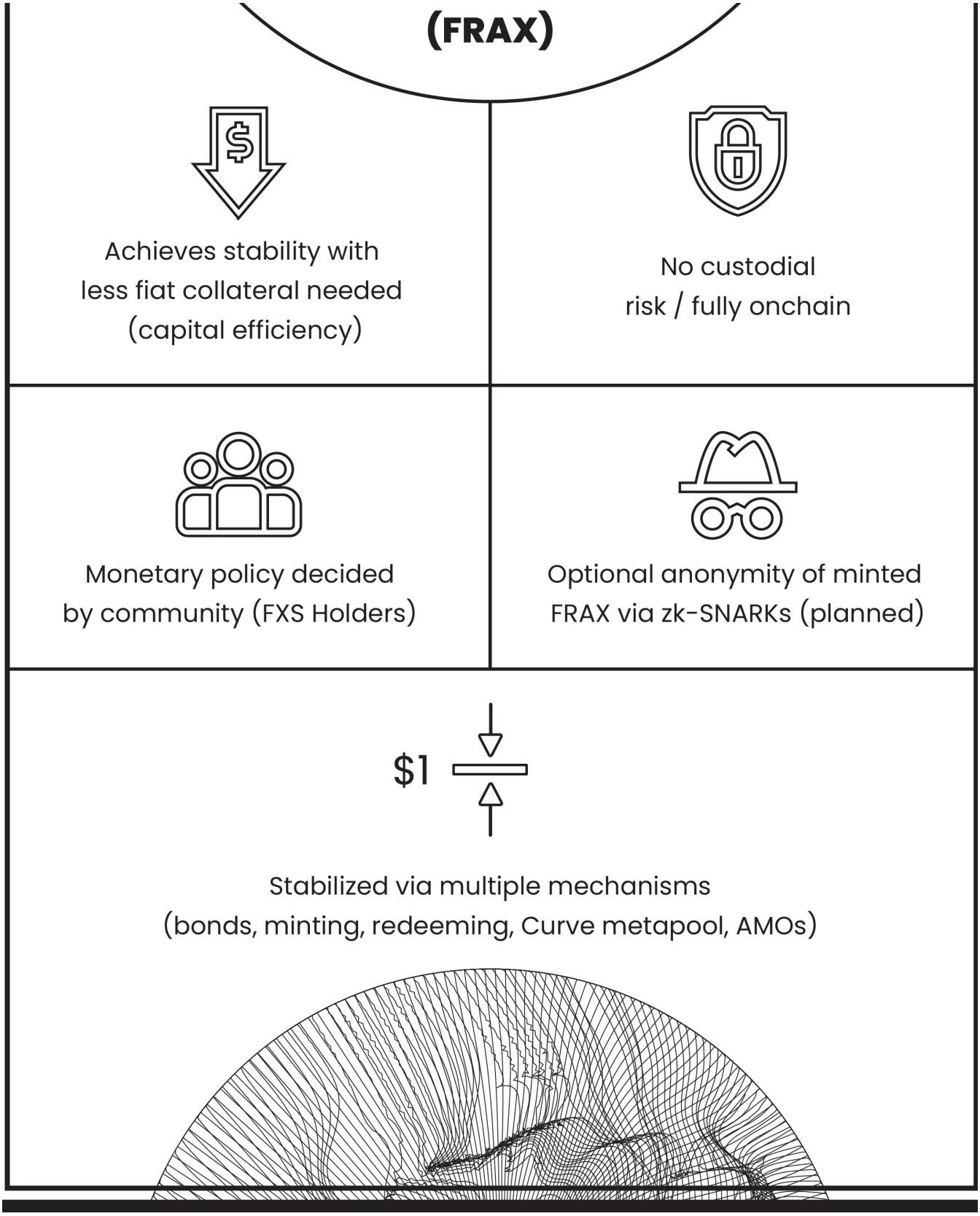
Introduction

Many stablecoin protocols have entirely embraced one spectrum of design (entirely collateralized) or the other extreme (entirely algorithmic with no backing). Collateralized stablecoins either have custodial risk or require on-chain overcollateralization. These designs provide a stablecoin with a fairly tight peg with higher confidence than purely algorithmic designs. Purely algorithmic designs such as Basis, Empty Set Dollar, and Seigniorage Shares provide a highly trustless and scalable model that captures the early Bitcoin vision of decentralized money but with useful stability. The issue with algorithmic designs is that they are difficult to bootstrap, slow to grow (as of Q4 2020 none have significant traction), and exhibit extreme periods of volatility which erodes confidence in their usefulness as actual stablecoins. They are mainly seen as a game/experiment than a serious alternative to collateralized stablecoins.

Frax attempts to be the first stablecoin protocol to implement design principles of both to create a highly scalable, trustless, extremely stable, and ideologically pure on-chain money. The Frax protocol is a two token system encompassing a stablecoin, Frax (FRAX), and a governance token, Frax Shares (FXS). The protocol also has a pool contract which holds USDC collateral. Pools can be added or removed with governance.

Although there's no predetermined timeframes for how quickly the amount of collateralization changes, we believe that as FRAX adoption increases, users will be more comfortable with a higher percentage of FRAX supply being stabilized algorithmically rather than with collateral. The collateral ratio refresh function in the protocol can be called by any user once per hour. The function can change the collateral ratio in steps of .25% if the price of FRAX is above or below $1. When FRAX is above $1, the function lowers the collateral ratio by one step and when the price of FRAX is below $1, the function increases the collateral ratio by one step. Both refresh rate and step parameters can be adjusted through governance. In a future update of the protocol, they can even be adjusted dynamically using a PID controller design. The price of FRAX, FXS, and collateral are all calculated with a time-weighted average of the Uniswap pair price and the ETH:USD Chainlink oracle. The Chainlink oracle allows the protocol to get the true price of USD instead of an average of stablecoin pools on Uniswap. This allows FRAX to stay stable against the dollar itself which would provide greater resiliency instead of using a weighted average of existing stablecoins only.

FRAX stablecoins can be minted by placing the appropriate amount of its constituent parts into the system. At genesis, FRAX is 100% collateralized, meaning that minting FRAX only requires placing collateral into the minting contract. During the fractional phase, minting FRAX requires placing the appropriate ratio of collateral and burning the ratio of Frax Shares (FXS). While the protocol is designed to accept any type of cryptocurrency as collateral, this implementation of the Frax Protocol will mainly accept on-chain stablecoins as collateral to smoothen out volatility in the collateral so that FRAX can transition to more algorithmic ratios smoothly. As the velocity of the system increases, it becomes easier and safer to include volatile cryptocurrency such as ETH and wrapped BTC into future pools with governance.



Price Stability

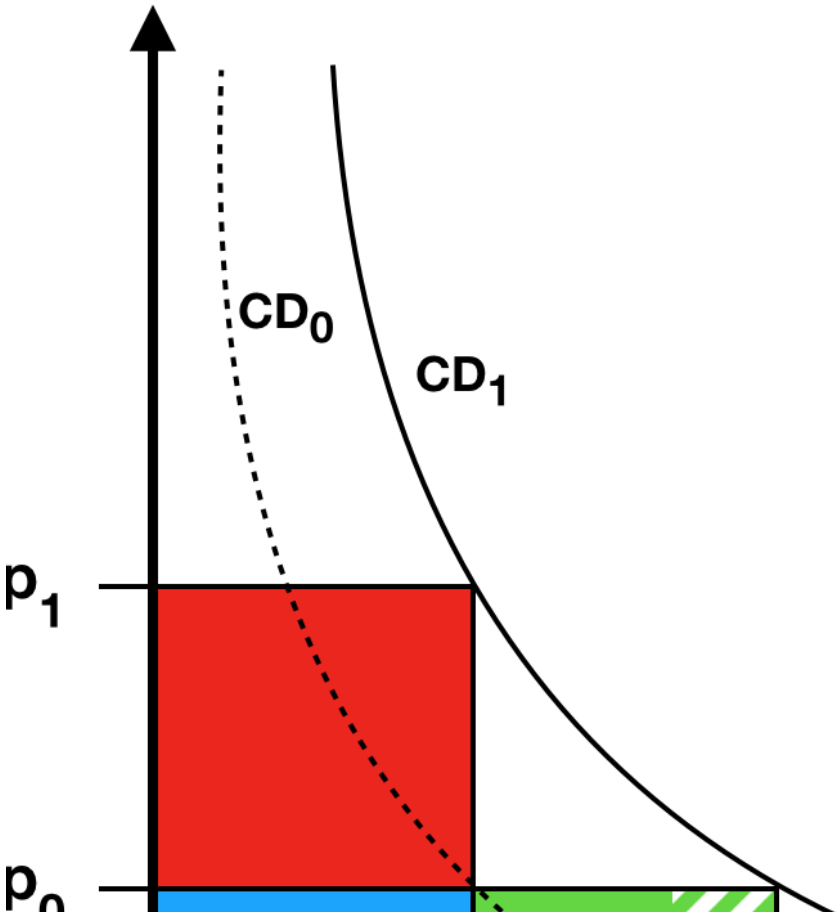
How arbitrage keeps FRAX price-stable

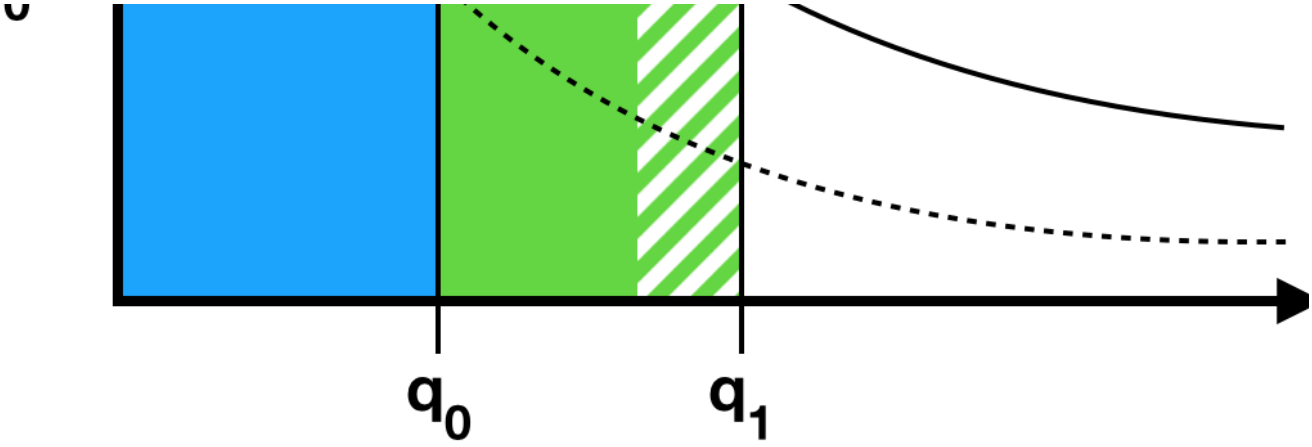
FRAX can always be minted and redeemed from the system for $1 of value. This allows arbitragers to balance the demand and supply of FRAX in the open market. If the market price of FRAX is above the price

target of $1, then there is an arbitrage opportunity to mint FRAX tokens by placing $1 of value into the system per FRAX and sell the minted FRAX for over $1 in the open market. At all times in order to mint new FRAX a user must place $1 worth of value into the system. The difference is simply what proportion of collateral and FXS makes up that $1 of value. When FRAX is in the 100% collateral phase, 100% of the value that is put into the system to mint FRAX is collateral. As the protocol moves into the fractional phase, part of the value that enters into the system during minting becomes FXS (which is then burned from circulation). For example, in a 98% collateral ratio, every FRAX minted requires $.98 of collateral and burning $.02 of FXS. In a 97% collateral ratio, every FRAX minted requires $.97 of collateral and burning $.03 of FXS, and so on.

If the market price of FRAX is below the price range of $1, then there is an arbitrage opportunity to redeem FRAX tokens by purchasing cheaply on the open market and redeeming FRAX for $1 of value from the system. At all times, a user is able to redeem FRAX for $1 worth of value from the system. The difference is simply what proportion of the collateral and FXS is returned to the redeemer. When FRAX is in the 100% collateral phase, 100% of the value returned from redeeming FRAX is collateral. As the protocol moves into the fractional phase, part of the value that leaves the system during redemption becomes FXS (which is minted to give to the redeeming user). For example, in a 98% collateral ratio, every FRAX can be redeemed for $.98 of collateral and $.02 of minted FXS. In a 97% collateral ratio, every FRAX can be redeemed for $.97 of collateral and $.03 of minted FXS.

The FRAX redemption process is seamless, easy to understand, and economically sound. During the 100% phase, it is trivially simple. During the fractional-algorithmic phase, as FRAX is minted, FXS is burned. As FRAX is redeemed, FXS is minted. As long as there is demand for FRAX, redeeming it for collateral plus FXS simply initiates minting of a similar amount of FRAX into circulation on the other end (which burns a similar amount of FXS). Thus, the FXS token's value is determined by the demand for FRAX. The value that accrues to the FXS market cap is the summation of the non-collateralized value of FRAX's market cap. This is the summation of all past and future shaded areas under the curve displayed as follows.





The demand-supply curve illustrates how minting and redeeming FRAX keeps the price stabilized (q is quantity, p is price). At CD_0 the price of FRAX is at q_0. If there is more demand for FRAX, the curve shifts right to CD_1 and a new price, p_1, for the same quantity q_0. In order to recover the price to $1, new FRAX must be minted until q_1 is reached and the p_0 price is recovered. Since market capitalization is calculated as price times quantity, the market cap of FRAX at q_0 is the blue square. The market cap of FRAX at q_1 is the sum of the areas of the blue square and green square. Notice that in this example the new market cap of FRAX would have been the same if the quantity did not increase because the increase in demand is simply reflected in the price, p_1. Given an increase in demand, market cap increases either through an increase in price or increase in quantity (at a stable price). This is clear because the red square and green square have the same area and thus would have added the same amount of value in market cap. Note: the semi-shaded portion in the green square denotes the total value of FXS shares that would be burned if the new quantity of FRAX was generated at a hypothetical collateral ratio of 66%. This is important to visualize because FXS market cap is intrinsically linked to demand for FRAX.

Lastly, it's important to note that Frax is an agnostic protocol. It makes no assumptions about what collateral ratio the market will settle on in the long-term. It could be the case that users simply do not have confidence in a stablecoin with 0% collateral that's entirely algorithmic. The protocol does not make any assumptions about what that ratio is and instead keeps the ratio at what the market demands for pricing FRAX at $1. It could be the case that the protocol only ever reaches, for example, a 60% collateral ratio and only 40% of the FRAX supply is algorithmically stabilized while over half of it is backed by collateral. The protocol only adjusts the collateral ratio as a result of demand for more FRAX and changes in FRAX price. When the price of FRAX falls below $1, the protocol recollateralizes and increases the ratio until confidence is restored and the price recovers. It will not decollateralize the ratio unless demand for FRAX increases again. It could even be possible that FRAX becomes entirely algorithmic but then recollateralizes to a substantial collateral ratio should market conditions demand. We believe this deterministic and reflexive protocol is the most elegant way to measure the market's confidence in a non-backed stablecoin. Previous algorithmic stablecoin attempts had no collateral within the system on day 1 (and never used collateral in any way). Such previous attempts did not address the lack of market confidence in an algorithmic stablecoin on day 1. It should be noted that even USD, which Frax is pegged to, was not a fiat currency until it had global prominence.

Collateral Ratio

The protocol adjusts the collateral ratio during times of FRAX expansion and retraction. During times of expansion, the protocol decollateralizes (lowers the ratio) the system so that less collateral and more FXS must be deposited to mint FRAX. This lowers the amount of collateral backing all FRAX. During times of retraction, the protocol recollateralizes (increases the ratio). This increases the ratio of collateral in the system as a proportion of FRAX supply, increasing market confidence in FRAX as its backing increases.

At genesis, the protocol adjusts the collateral ratio once every hour by a step of .25%. When FRAX is at or above $1, the function lowers the collateral ratio by one step per hour and when the price of FRAX is below $1, the function increases the collateral ratio by one step per hour. This means that if FRAX price is at or over $1 a majority of the time through some time frame, then the net movement of the collateral ratio is decreasing. If FRAX price is under $1 a majority of the time, then the collateral ratio is increasing toward 100% on average.

In a future protocol update, the price feeds for collateral can be deprecated and the minting process can be moved to an auction based system to limit reliance on price data and further decentralize the protocol. In such an update, the protocol would run with no price data required for any asset including FRAX and FXS. Minting and redemptions would happen through open auction blocks where bidders post the highest/lowest ratio of collateral plus FXS they are willing to mint/redeem FRAX for. This auction arrangement would lead to collateral price discovery from within the system itself and not require any price information via oracles. Another possible design instead of auctions could be using PID-controllers to provide arbitrage opportunities for minting and redeeming FRAX similar to how a Uniswap trading pair incentivizes pool assets to keep a constant ratio that converges to their open market target price.

PIDController (update)

As of Feburary 2021, the system uses a PIDController to control the collateral ratio according to the change in the growth ratio, defined as such:

$$G_r = \frac{\sum_{a_i}^{a_n} Z_i * P_z}{F}$$

G_r is the growth ratio

Z_i is the supply of FXS provided as liquidity to a pair on a decentralized AMM (Uniswap, Sushiswap, etc.)

a_i to a_n are the FXS pairs on the AMMs

P_z is the price of FXS

F is the total supply of FRAX

At its core, the growth ratio measures how much FXS liquidity there is against the overall supply of FRAX. The reasoning is that the higher the growth ratio, the more FRAX that could be redeemed with less overall percentage change in the FXS supply. If redeemers were to sell their FXS minted from redeemed FRAX, a higher growth ratio would imply less price slippage on FXS and thus less likelihood of any undesireable negative feedback loops.

As the collateral ratio is changed by the change in the growth ratio, a low overall CR implies more

preceeding periods of net positive growth ratio change than net negative. This can be caused by periods of sustained positive FXS price increases, redemptions of FRAX that do not affect the FXS price from the newly minted FXS, or more FXS liquidity entering AMMs.

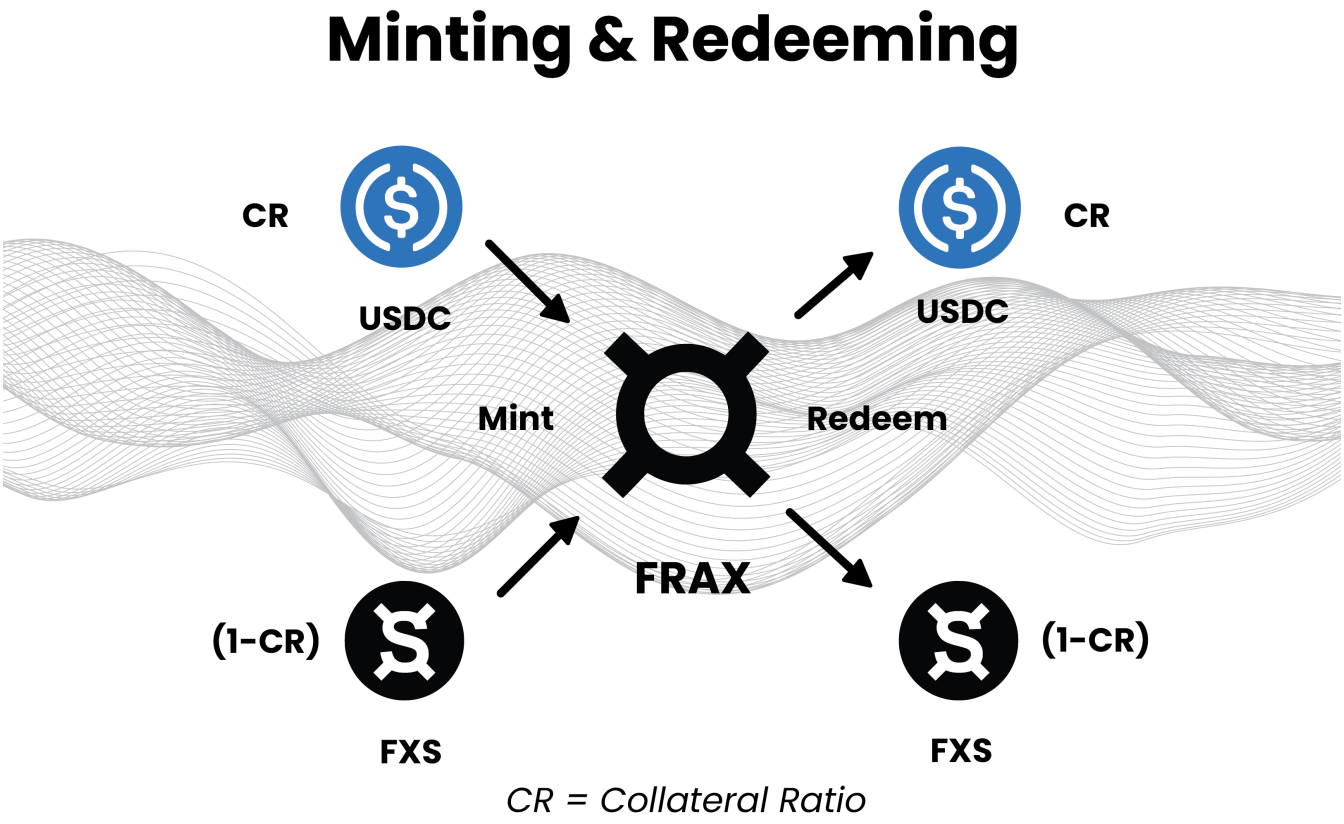
The motivation for the growth ratio is to take in the signal of the market cap of FRAX and FXS, such that a change in the collateral ratio can be supported by current conditions. For example, a situation of $5 million of FXS liquidity with 50 million outstanding FRAX is much less fragile than one with the same FXS liquidity but 500 million outstanding FRAX.

In the previous model, looking only at the price of FRAX to change the collateral ratio was sufficient for the protocol's bootstrap phase, but the recent growth and current size of the system merits a change in the model to consider the growth ratio to allow for more accurrate feedback. The new system still uses a price band, but only adjusts the collateral ratio up or down when the price of FRAX is outside of the targeted band.

Minting and Redeeming

Detailing the process of minting and redeeming FRAX

Minting



All FRAX tokens are fungible with one another and entitled to the same proportion of collateral no matter what collateral ratio they were minted at. This system of equations describes the minting function of the Frax Protocol:

$$F = \overbrace{(Y * P_y)}^{\text{collateral value}} + \overbrace{(Z * P_z)}^{\text{FXS value}}$$

$$(1 - C_r)(Y * P_y) = C_r(Z * P_z)$$

F is the units of newly minted FRAX

C_r is the collateral ratio

Y is the units of collateral transferred to the system

P_y is the price in USD of Y collateral

Z is the units of FXS burned

P_z is the price in USD of FXS

Example A: Minting FRAX at a collateral ratio of 100% with 200 USDC ($1/USDC price)

To be explicit, we can start by finding the FXS needed to mint FRAX with `200 USDC` (`$1/USDC`) at a collateral ratio of 1.00

$$(1 - 1.00)(100 * 1.00) = 1.00(Z * P_z)$$

$$0 = (Z * P_z)$$

Thus, we show that no FXS is needed to mint FRAX when the protocol collateral ratio is 100% (fully collateralized). Next, we solve for how much FRAX we will get with the `200 USDC`.

$$F = (200 * 1.00) + (0)$$

$$F = 200$$

`200 FRAX` are minted in this scenario. Notice how the entire value of FRAX is in dollar value of the collateral when the ratio is at `100%`. Any amount of FXS attempting to be burned to mint FRAX is returned to the user because the second part of the equation cancels to `0` regardless of the value of Z and P_z.

Example B: Minting FRAX at a collateral ratio of 80% with 120 USDC ($1/USDC price) and an FXS price of $2/FXS.

First, we need to figure out how much FXS we need to match the corresponding amount of USDC.

$$(1 - 0.8)(120 * 1.00) = 0.8(Z * 2.00)$$

$$Z = 15$$

Thus, we need to deposit 15 FXS alongside 120 USDC at these conditions. Next, we compute how much FRAX we will get.

$$F = (120 * 1.00) + (15 * 2.00)$$

$$F = 150$$

`150 FRAX` are minted in this scenario. `120 FRAX` are backed by the value of USDC as collateral while the remaining `30 FRAX` are not backed by anything. Instead, FXS is burned and removed from circulation proportional to the value of minted algorithmic FRAX.

Example C: Minting FRAX at a collateral ratio of 50% with 220 USDC ($.9995/USDC price) and an FXS price of $3.50/FXS

First, we start off by finding the FXS needed.

$$(1 - .50)(220 * .9995) = .50(Z * 3.50)$$

$$Z = 62.54$$

Next, we compute how much FRAX we will get.

$$F = (220 * .9995) + (62.54 * 3.50)$$

$$F = 437.78$$

`437.78 FRAX` are minted in this scenario. Proportionally, half of the newly minted FRAX are backed by the value of USDC as collateral while the remaining 50% of FRAX are not backed by anything. `62.54 FXS` is burned and removed from circulation, half the value of the newly minted FRAX. Notice that the price of the collateral affects how many FRAX can be minted – FRAX is pegged to 1 USD, not 1 unit of USDC.

If not enough FXS is put into the minting function alongside the collateral, the transaction will fail with a `subtraction underflow` error.

Redeeming

Redeeming FRAX is done by rearranging the previous system of equations for simplicity, and solving for the units of collateral, Y, and the units of FXS, Z.

$$Y = \frac{F * (C_r)}{P_y}$$

$$Z = \frac{F * (1 - C_r)}{P_z}$$

F is the units of FRAX redeemed
C_r is the collateral ratio
Y is the units of collateral transferred to the user
P_y is the price in USD of Y collateral
Z is the units of FXS minted to the user
P_z is the price in USD of FXS

Example D: Redeeming 170 FRAX at a collateral ratio of 65%. Oracle price is $1.00/USDC and

$3.75/FXS.

$$Y = \frac{170 * (.65)}{1.00}$$

$$Z = \frac{170 * (.35)}{3.75}$$

Thus, $Y = 110.5$ and $Z = 15.867$

Redeeming `170 FRAX` returns `$170` of value to the redeemer in `110.5 USDC` from the collateral pool and `15.867 of newly minted FXS` tokens at the current FXS market price.

Additionally, there is a `2` block delay parameter (adjustable by governance) on withdrawing redeemed collateral to protect against flash loans.

NOTE: These examples do not account for the minting and redemption fees, which are set between 0.20% and 0.45%

Frax Shares (FXS)

FXS is the value accrual and governance token of the entire Frax ecosystem. All utility is concentrated into FXS.

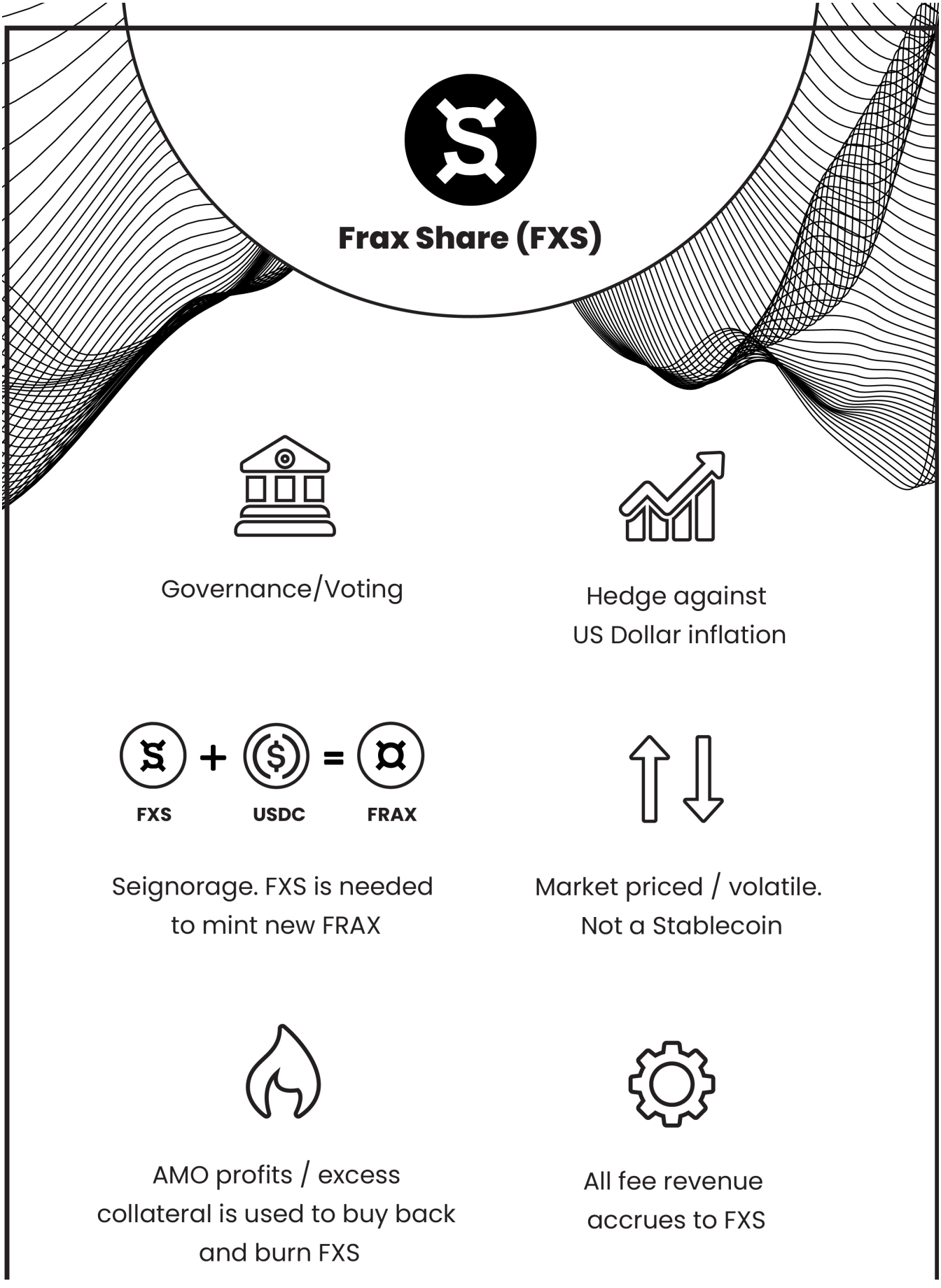
The Frax Share token (FXS) is the non-stable, utility token in the protocol. It is meant to be volatile and hold rights to governance and all utility of the system. It is important to note that we take a highly governance-minimized approach to designing trustless money in the same ethos as Bitcoin. We eschew DAO-like active management such as MakerDAO. The less parameters for a community to be able to actively manage, the less there is to disagree on. Parameters that are up for governance through FXS include adding/adjusting collateral pools, adjusting various fees (like minting or redeeming), and refreshing the rate of the collateral ratio. No other actions such as active management of collateral or addition of human-modifiable parameters are possible other than a hardfork that would require voluntarily moving to a new implementation entirely.

The FXS token has the potential of upside utility and downside utility of the system, where the delta changes in value are always stabilized away from the FRAX token itself. FXS supply is initially set to 100 million tokens at genesis, but the amount in circulation will likely be deflationary as FRAX is minted at higher algorithmic ratios. The design of the protocol is such that FXS would be largely deflationary in supply as long as FRAX demand grows.

The FXS token's market capitalization should be calculated as the future expected net value creation from seigniorage of FRAX tokens in perpetuity, the cash flow from minting and redemption fees, and utilization of unused collateral. Additionally, as the market cap of FXS increases, so does the system's ability to keep FRAX stable. Thus, the priority in the design is to accrue maximal value to the FXS token while maintaining FRAX as a stable currency. As Robert Sam's described in the original Seigniorage Shares whitepaper: "Share tokens are like the asset side of a central bank's balance sheet. The market capitalisation of shares at any point in time fixes the upper limit on how much the coin supply can be reduced." Likewise, the Frax protocol takes inspiration from Sams' proposal as Frax is a hybrid (fractional) seigniorage shares model.

veFXS & Long Term Staking

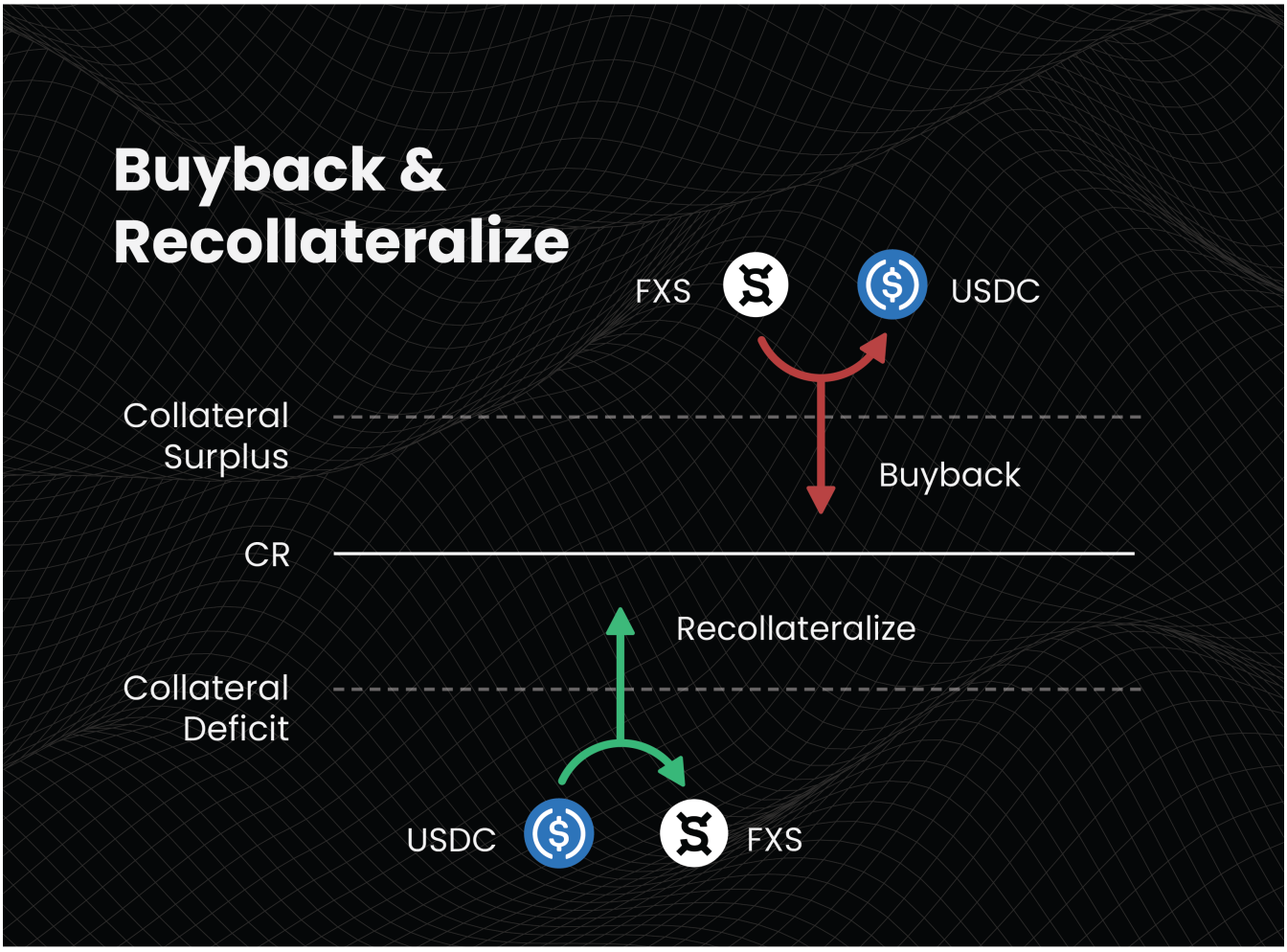
In May 2020, the protocol now allows FXS holders to lock up FXS tokens to generate veFXS and earn special boosts, special governance rights, and AMO profits. Check the in depth veFXS specs for more information on how all veFXS features function.



Schematic of current utility of FXS (with more value accrual mechanisms to be added in Frax v2)

Buybacks & Recollateralization

The protocol at times will have excess collateral value or require adding collateral to reach the collateral ratio. To quickly redistribute value back to FXS holders or increase system collateral, two special swap functions are built into the protocol: buyback and recollateralize.



Recollateralization

Anyone can call the recollateralize function which then checks if the total collateral value in USD across the system is below the current collateral ratio. If it is, then the system allows the caller to add up to the amount needed to reach the target collateral ratio in exchange for newly minted FXS at a bonus rate. The bonus rate is set to `.20%` to quickly incentivize arbitragers to close the gap and recollateralize the protocol to the target ratio. The bonus rate can be adjusted or changed to a dynamic PID controller adjusted variable through

governance.

$$FXS_{received} = \frac{(Y * P_y)(1 + B_r)}{P_z}$$

Y is the units of collateral needed to reach the collateral ratio

P_y is the price in USD of Y collateral

B_r is the bonus rate for FXS emitted when recollateralizing

P_z is the price in USD of FXS

Example A: There is 100,000,000 FRAX in circulation at a 50% collateral ratio. The total value of collateral across the USDT and USDC pools is 50m USD and the system is balanced. The price of FRAX drops to $.99 and the protocol increases the collateral ratio to 50.25%.

There is now `$250,000` worth of collateral needed to reach the target ratio. Anyone can call the recollateralize function and place up to $250,000 of collateral into pools to receive an equal value of FXS plus a bonus rate of `.20%`.

Placing `250,000 USDT` at a price of `$1.00/USDT` and a market price of `$3.80/FXS` is as follows:

$$FXS_{received} = \frac{(250000 * 1.00)(1 + .0075)}{3.80}$$

$$FXS_{received} = 66282.89$$

Buybacks

The opposite scenario occurs when there is excess collateral in the system than required to hold the target collateral ratio. This can happen a number of ways:

- The protocol has been lowering the collateral ratio successfully keeping the price of FRAX stable
- Interest bearing collateral is accepted into the protocol and its value accrues
- Minting and redemption fees are creating revenue

In such a scenario, any FXS holder can call the buyback function to exchange the amount of excess collateral value in the system for FXS which is then burned by the protocol. This effectively redistributes any excess value back to the FXS distribution and holders don't need to actively participate in buybacks to gain value since there is no bonus rate for the buyback function. It effectively models a share buyback to the governance token distribution.

$$Collateral_{received} = \frac{Z * P_z}{P_y}$$

Z is units of FXS deposited to be burned

P_y is the price in USD of the collateral

P_z is the price in USD of FXS

Example B: There is 150,000,000 FRAX in circulation at a 50% collateral ratio. The total value of

collateral across the USDT and USDC pools is 76m USD. There is $1m worth of excess collateral available for FXS buybacks.

Anyone can call the buyback function and burn up to `$1,000,000 worth of FXS` to receive excess collateral.

Burning `238,095.238 FXS` at a price of `$4.20/FXS` to receive USDC at a price of `$.99/USDC` is as follows:

$$USDC_{received} = \frac{238095.238 * 4.20}{.99}$$

$$USDC_{received} = 1010101.01$$

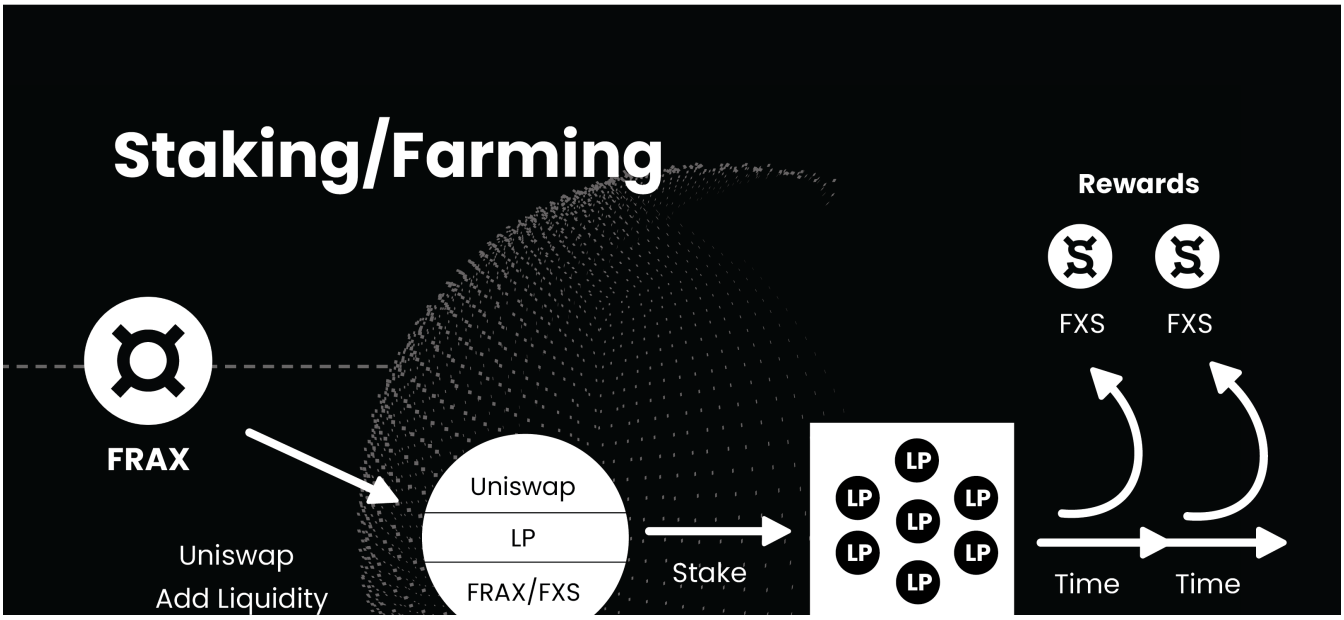
Liquidity Programs & Staking

Distributing FXS tokens to farmers and incentivizing DeFi programs with novel rules

FXS rewards will be claimable for users who deposit **Uniswap LP tokens** to incentivized pairs, which can be attained by **adding liquidity** to token pairs on Uniswap. Each incentivized pair will have its own emission rate, and the sum of all FXS rewards across the incentivized pairs will emit at a base rate of `18,000,000 FXS` for the first year, and formerly had a boost of up to `2x` the more FRAX became algorithmic. For more details about the specific pairs and the overall LP program, check the Token Distribution section of the docs.

Collateral Ratio Boost (deprecated 4/17/2021)

The rate of emission in every pool is multiplied by a CR Boost factor that is inversely proportional to the collateral ratio. This means that as FRAX becomes more algorithmic, the rate of emission of FXS for all pools increases. The CR Boost constant is set to a `2x` maximum multiplier, meaning that if FRAX is entirely algorithmic with a CR of `0%`, then the rate of emission of FXS increases by `2x`. For example, the first year of emission of `18,000,000 FXS` per year would increase to a rate of `36,000,000 FXS` per year if the CR reaches `0%`. If FRAX is `50%` collateralized, the boost will be `1.5x`, putting the emission rate at `27,000,000 FXS` per year. This boost was deprecated in April 2021 per governance action.



Time Locked Staking

Any LP can lock their LP tokens up to `1095 days (3 years)`. LP stakes are multiplied by two boost factors: time locked & collateral ratio. The collateral ratio boost applies to the base emission rate of FXS, so an increase in the collateral ratio boost means more FXS distributed across the whole system. The time locked boost applies to an individual's stake as a proportion of all of the stakes in the pool, making it a zero-sum outcome when someone gets a boost from time locked stakes. In other words, a time locked boost will increase the amount of FXS a single user gets by increasing their proportion of the pool which decreases the proportion of rewards for everyone else in the pool. This is done to help balance the risk/reward of locking liquidity into the system for a fixed amount of time. Time locked staking is intended to further reward LPs who have a long term belief in the Frax Protocol and want to commit to providing liquidity for an extended period of time. If any pool emission rate is changed due to a governance action, the time locked stakes of the pool are automatically unlocked so that emission rates don't change on LPs who have committed to locking funds.

veFXS Boost

veFXS holders will get an additive boost to their weight when farming. See the veFXS specs for more details.

> veFXS
> Frax Finance ¤

Conclusion

Ending thoughts & a functional way to visualize the Frax Protocol

Frax uses ideas from Uniswap and AMMs to build a novel hybrid stablecoin design never seen before. In a Uniswap pool, the ratio of asset A and B has to be proportional due to the constant product function. The LP token is just a pro rata claim on the pool + fees so it is usually increasing in value (if fees higher than impermanent loss) or loses value (if impermanent loss greater than fees). The LP token is just passive claims on the pool.

Frax takes that idea and turns it over to design a unique stablecoin. The LP token is the stablecoin, FRAX. It is the object of stabilization and always mintable/redeemable for $1 worth of collateral and the governance (FXS) token at the collateral ratio. This ratio of the two assets (collateral and FXS) dynamically changes based on the price of the stablecoin. If the stablecoin price is dropping, then the protocol tips the ratio in favor of collateral and less in the FXS token to regain confidence in FRAX. An arbitrage opportunity arises for people wanting to put in collateral into the pool at the new ratio for discounted FXS which the protocol mints for this "recollateralization swap." This recollateralizes the protocol to the new, higher collateral ratio.

If FRAX is over $1, then the protocol tips the collateral ratio to the FXS token to measure the market's confidence in more FRAX supply being stabilized algorithmically. As FRAX becomes more algorithmic, the excess collateral can go back to FXS holders through a buyback shares function that anyone can call to burn their FXS tokens for an equal value of excess collateral. The "buyback swap" function always keeps value accruing to the governance token any time there is excess fees/collateral/value in the system.

This 'Frax dance' is always happening and uses AMM game theory to test different ratios of collateralization, incentivize recollateralizing through arbitrage swaps, and redistribute excess value back to FXS holders through a buyback swap. The protocol starts at a 100% collateral ratio at genesis and might or might not ever get to purely algorithmic. The novel insight is to use market forces itself to see how much of a stablecoin can be algorithmically stabilized with its own seigniorage token so that it keeps a tight band around $1 like fiatcoins. Purely algorithmic/rebase designs like Basis, ESD, and Seigniorage Shares have wildly fluctuating prices as much as +-40% around $1 that take days/weeks to stabilize before going through another cycle. This is counterproductive and assumes the market actually wants/needs a stablecoin with 0% collateralization. Frax doesn't make this assumption. Instead, it measures the market's preference and finds the actual collateral ratio which holds a stablecoin tightly around $1, periodically testing small differences in the ratio when the price of FRAX slightly rises/drops. Frax uses AMM concepts to make a real-time fractional-algorithmic stablecoin that is as fast at price recovery as Uniswap is at keeping trading pools correctly priced.

As this system gets more efficient and the velocity of the system increases, collateral pools can include other assets instead of stablecoins like volatile crypto such as ETH and wrapped BTC. As the price of the volatile asset rises, users will use the buy back shares function to distribute the excess value to FXS holders. When the price of the volatile collateral drops, there is an instant arbitrage opportunity to put in more crypto for discounted FXS to keep the collateral ratio at the target. Just like a Uniswap trading pair keeps its constant product function balanced, the Frax Protocol keeps its target collateral ratio balanced to what the market needs for FRAX to be $1.

The above example uses "collateral" and "FXS" as the two assets within the protocol, but in reality, Frax can have multiple pools of collateral and multiple algorithmic token pools with weights, similar to Balancer. The protocol currently has USDC collateral pools and just 1 algorithmic token: FXS. In v2, we will release a second algorithmic token, the Frax Bond token (FXB) which represents pure debt with an interest rate attached.

We believe that the fractional-algorithmic design of Frax is paradigm shifting for stablecoins. It is fast, real-

time balancing, algorithmic, governance-minimized, and extremely resilient. We strongly believe the Frax protocol can become a foil to Bitcoin's "hard money" narrative by demonstrating algorithmic monetary policy to create a trustless stablecoin that all of the crypto community can embrace

FRAX V2 - Algorithmic Market Operations (AMO)

AMO Overview

A framework for composable, autonomous central banking legos

Frax v2 expands on the idea of fractional-algorithmic stability by introducing the idea of the “Algorithmic Market Operations Controller” (AMO). An AMO module is an autonomous contract(s) that enacts arbitrary monetary policy so long as it *does not change the FRAX price off its peg*. This means that AMO controllers can perform open market operations algorithmically (as in the name), but they cannot arbitrarily mint FRAX out of thin air and break the peg. This keeps FRAX’s base layer stability mechanism pure and untouched, which has been the core of what makes our protocol special and inspired other smaller projects.

Frax v1: Background

In Frax v1, there was only a single AMO, the fractional-algorithmic stability mechanism. We refer to this as the **base stability mechanism**. You can read about it in the Core Whitepaper.

In Frax v1, the collateral ratio of the protocol is dynamically rebalanced based on the market price of the FRAX stablecoin. If the price of FRAX is above $1, then the collateral ratio (CR) decreases ("decollateralization"). If the price of FRAX is below $1 then the CR increases ("recollateralization"). The protocol always honors redemptions of FRAX at the $1 peg, but since the CR is dynamic, it must fund redemptions of FRAX by minting Frax Share tokens (FXS) for the remainder of the value. For example, at an 85% CR, every redeemed FRAX gives the user $.85 USDC and $.15 of minted FXS. It is a trivial implementation detail whether the protocol returns to the redeemer $.15 worth of FXS directly or atomically sells the FXS for collateral onchain to return the full $1 of value in collateral – the economic implementation is the same.

This base mechanism can be abstracted down to the following:

1. Decollateralize - Lower the CR by some increment x every time t if FRAX >$1
2. Equilibrium - Don't change the CR if FRAX = $1
3. Recollateralize - Increase the CR by some increment x every time t if FRAX <$1
4. FXS value accrual mechanism - burn FXS with minted unbacked FRAX, extra collateral, & fees

At its fundamental core, the Frax Protocol is a banking algorithm that adjusts its balance sheet ratio based on the market's pricing of FRAX. The collateral ratio is simply the ratio of the protocol's capital (collateral) over its liabilities (FRAX stablecoins). The market 'votes' on what this ratio should be by selling/exiting the stablecoin if it's too low (thereby slightly pushing the price below $1) or by continuing to demand FRAX (thereby slightly pushing the price above $1). This decollateralization and recollateralization helps find an equilibrium reserve requirement for the protocol to keep a very tight peg and maximize capital efficiency of money creation. **By definition, the protocol mints the exact amount of FRAX stablecoins the market demands at the exact collateral ratio the market demands for $1 FRAX.**

Frax v2: AMOs

We can therefore generalize the previous mechanism to any arbitrarily complex market operation to create a Turing-complete design space of stability mechanisms. Thus, each AMO can be thought of as a central bank money lego. Every AMO has 4 properties:

1. Decollateralize - the portion of the strategy which lowers the CR
2. Market operations - the portion of the strategy that is run in equilibrium and doesn't change the CR
3. Recollateralize - the portion of the strategy which increases the CR
4. FXS1559 - a formalized accounting of the balance sheet of the AMO which defines exactly how much FXS can be burned with profits above the target CR.

With the above framework clearly defined, it's now easy to see how Frax v1 is the simplest form of an AMO. It is essentially the base case of any possible AMO. In v1, decollateralization allows for expansion of the money supply and excess collateral to flow to burning FXS. Recollateralization mints FXS to increase the collateral ratio and lower liabilities (redemptions of FRAX).

The base layer fractional-algorithmic mechanism is always running just like before. If FRAX price is above the peg, the CR is lowered, FRAX supply expands like usual, and AMO controllers keep running. If the CR is lowered to the point that the peg slips, the AMOs have predefined recollateralize operations which increases the CR. The system recollateralizes just like before as protocol liabilities (stablecoins) are redeemed and the CR goes up to return to the peg. This allows all AMOs to operate with input from market forces and preserve the full design specs of the v1 base case.

AMOs enable FRAX to become one of the most powerful stablecoin protocols by creating maximum flexibility and opportunity without altering the base stability mechanism that made FRAX the leader of the algorithmic stablecoin space. AMO modules open a modular design space that will allow for constant upgrades and improvements without jeopardizing design elegance, composability, or increasing technical complexity. Lastly, because AMOs are a complete "mechanism-in-a-box," anyone can propose, build, and create AMOs which can then be deployed with governance as long as they adhere to the above specifications.

References and Resources/Links

1. Original Announcement Post

2. Quick Twitter explainer thread

FXS1559

The most powerful utility accretive mechanism for the FXS token

FXS1559 calculates all excess value in the system above the collateral ratio and uses this value to buy FXS for burning. Every AMO proposal must include an FXS1559 function which calculates how much value over the collateral ratio (CR) there is accumulated. This value goes to burning FXS.

FXS1559 is named in homage to EIP1559, the Ethereum improvement proposal which burns ETH during block production given certain gas prices/usage metrics. EIP1559 has completely changed the ETH native asset's value proposition and formalizes value capture on the protocol level. EIP1559 tightly binds ETH's economic value on the protocol level. Similarly, FXS1559 binds FXS value capture on the AMO level. Because AMOs have an infinite, Turing-complete design space for conducting any market operation strategy, it's important to formalize how FXS captures value across every possible AMO design.

Specifically, every time interval t, FXS1559 calculates the excess value above the CR and mints FRAX in proportion to the collateral ratio against the value. It then uses the newly minted currency to purchase FXS on FRAX-FXS AMM pairs and burn it.

Example:
There is 100m FRAX in circulation with $86m of collateral value across the protocol at an 86% CR. The system is in equilibrium with FRAX trading at $1.00. Collateral is deployed through multiple AMOs, such as the Collateral Investor AMO and Curve AMO. The various market operations of the protocol earn yield, transaction fees, and interest. Each day there is $20,000 worth of revenue earned from various AMOs. This would increase the CR by .023% each day since it is a surplus of $20,000 of collateral value. After t = 24 hours, the CR is now 86.023% which is higher than the 86% target. Given that the CR is 86%, the protocol can rebalance to the CR in two ways. It can use the $20,000 worth of collateral profit to purchase FXS from AMMs. However, a more efficient and advantageous method is to mint 20,000/(.86) = 23,255.814 FRAX

It then takes the newly minted 23,255.814 FRAX and purchases FXS from the most liquid onchain market(s) (currently the FRAX-FXS Uniswap pair). The FXS is then immediately burned. This second method has the distinct advantage of expanding the FRAX supply, accruing value to the FXS token holders, as well as rebalancing the protocol to the CR.

Essentially, FXS1559 is an in-protocol rule for every AMO to formally channel excess value above the current target collateral ratio to FXS holders.

Collateral Investor

Invests idle collateral into various DeFi vaults/protocols

The Collateral Investor AMO moves idle USDC collateral to select DeFi protocols that provide reliable yield. Currently, the integrated protocols include: Aave, Compound, and Yearn. More can be added by governance. The main requirement for this AMO is to be able to pull out invested collateral immediately with

no waiting period in case of large FRAX redemptions. Collateral that is invested with an instant withdrawal ability does not count as lowering the CR of the protocol since it is spontaneously available to the protocol at all times. Nevertheless, the decollateralize function in the specs pulls out invested collateral starting with any time-delayed withdrawals (which there are none currently and not planned to be as of now).

Any investment revenue generated that places the protocol above the CR is burned with FXS1559.

AMO Specs

1. Decollateralize - Places idle collateral in various yield generating protocols. Investments that cannot be immediately withdrawn lower the CR calculation. Investments that can always be withdrawn at a 1 to 1 rate at all times such as Yearn USDC v2 and Compound do not count as lowering the CR.
2. Market operations - Compounds the investments at the CR.
3. Recollateralize - Withdraws investments from vaults to free up collateral for redemptions.
4. FXS1559 - Daily revenue that accrues from investments over the CR.

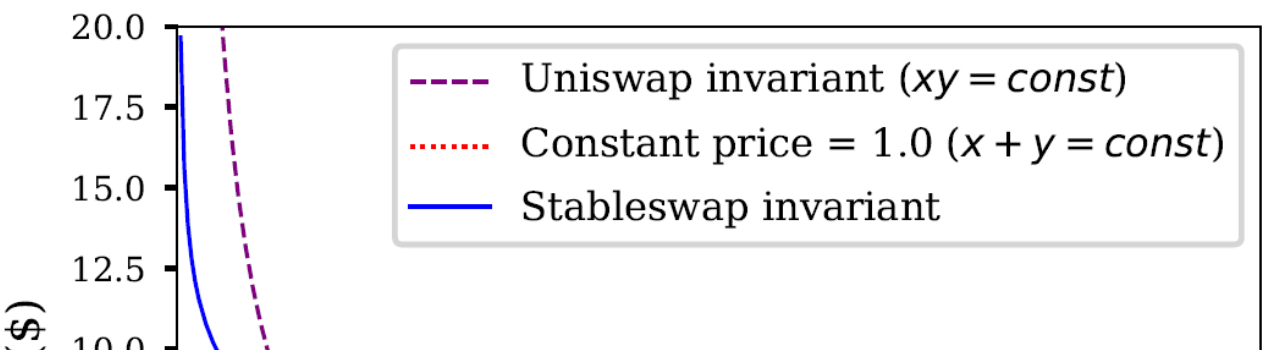
Curve

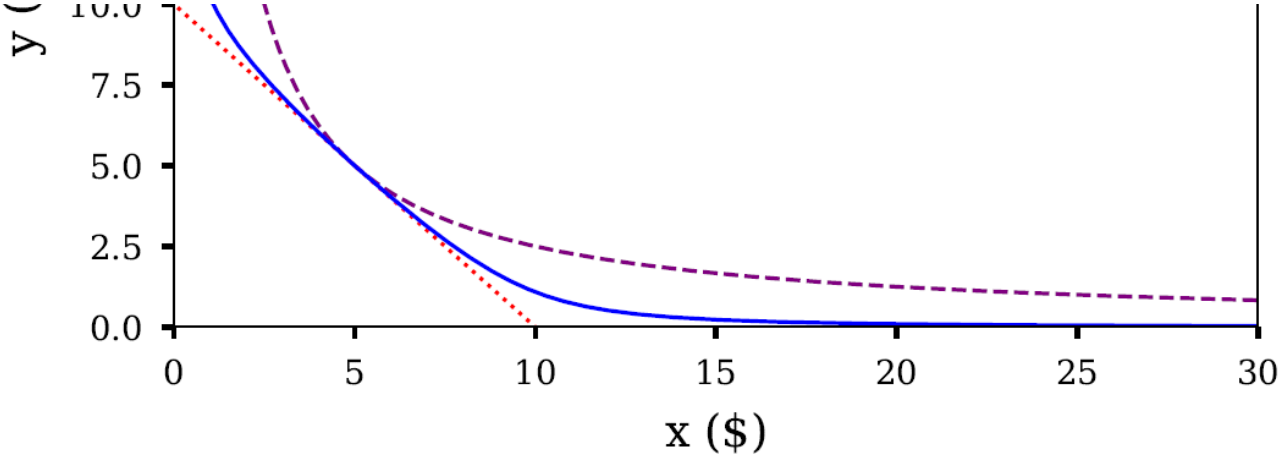
A stableswap pool with liquidity controlled and owned predominantly by the protocol

The Curve AMO puts FRAX and USDC collateral to work providing liquidity for the protocol and tightening the peg. Frax has deployed its own FRAX3CRV metapool. This means that the Frax deployer address owns admin privileges to its own Curve pool. This allows the Curve AMO controller to set and collect admin fees for FXS holders and various future functions. The protocol can move idle USDC collateral or new FRAX to its own Curve pool to create even more liquidity and tighten the peg while earning trading revenue.

AMO Specs

1. Decollateralize - Places idle collateral and newly minted FRAX into the FRAX3CRV pool.
2. Market operations - Accrues transaction fees, CRV rewards, and periodically rebalances the pool. The FRAX3CRV LP tokens are deposited into Yearn crvFRAX vault, Stake DAO, and Convex Finance for extra yield.
3. Recollateralize - Withdraws excess FRAX from pool first, then withdraws USDC to increase CR.
4. FXS1559 - Daily transaction fees and LP value accrued over the CR. (currently in development)





A comparison of the Uniswap and Stableswap curves, taken from the Curve whitepaper

Curve's Stableswap invariant allows for dampened price volatility between stablecoin swaps when reserves are not extremely imbalanced, approximating a linear swap curve when doing so.

$$\sum x_i = D;$$

Linear swap curve generalized to N coins

In cases of extreme imbalance, the invariant approaches the Uniswap constant-product curve.

$$\prod x_i = \left(\frac{D}{n}\right)^n$$

Constant-product swap curve generalized to N coins

The combination of two such curves allows for the expression of one or another, depending on what the ratio of the balances in the pool are, according to a coefficient. Using a dimensionless parameter χD^{n-1} as the coefficient, one may generalize the combination of the two curves to N coins.

$$\chi D^{n-1} \sum x_i + \prod x_i = \chi D^n + \left(\frac{D}{n}\right)^n$$

Absolute magic

Curve AMO

The protocol calculates the amount of underlying collateral the AMO has access to by finding the balance of USDC it can withdraw if the price of FRAX were to drop to the CR. Since FRAX is always backed by collateral at the value of the CR, it should never go below the value of the collateral itself. For example, FRAX should never go below $.85 at an 85% CR. This calculation is the safest and most conservative way to calculate the amount of collateral the Curve AMO has access to. This allows the Curve AMO to mint FRAX to place inside the pool in addition to USDC collateral to tighten the peg while knowing exactly how much collateral it has access to if FRAX were to break its peg.

Additionally, the AMO's overall strategy allows for optimizing the minimum FRAX supply Y such that selling all of Y at once into a Curve pool with Z TVL and A amplification factor will impact the price of FRAX by less than X%, where X is the CR's band sensitivity. Said in another way, the Curve AMO can put FRAX+USDC into its own Curve pool and control TVL. Since the CR recollateralizes when FRAX price drops by more than 1 cent under $1, that means that there is some value of FRAX that can be sold directly into the Curve pool before the FRAX price slips by 1%. The protocol can have at least that amount of algorithmic FRAX circulating on the open market since a sale of that entire amount at once into the Curve pool's TVL would not impact the price enough to cause the CR to move. These amounts are quite large and impressive when considering Curve's stablecoin optimized curve. For example, a 330m TVL FRAX3Pool (assuming balanced underlying 3Pool) can support at minimum a $39.2m FRAX sell order without moving the price by more than 1 cent. If the CR band is 1% then the protocol should have at least 39.2m algorithmic FRAX in the open market at minimum.

The above strategy is an extremely powerful market operation which would mathematically create a floor of algorithmic FRAX that can circulate without any danger of breaking the peg.

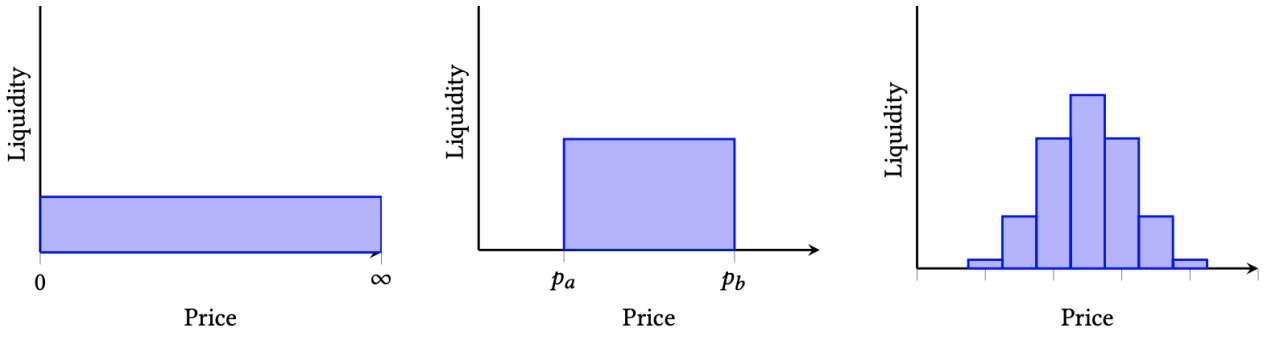
Additionally, Curve allocates CRV tokens as rewards for liquidity providers to select pools (called gauge rewards). Since the Frax protocol will likely be the largest liquidity provider of the FRAX3CRV pool, it can allocate all its FRAX3CRV LP tokens into Curve gauges to earn a significant return. The CRV tokens held within the Curve AMO can be used to vote in future Curve DAO governance controlled by FXS holders. This essentially means that the more the protocol employs liquidity to its own Curve pool, the more of the Curve protocol it will own and control through its earned CRV rewards. The long term effect of the Curve AMO is that Frax could become a large governance participant in Curve itself.

The Curve AMO contract is deployed at: `0xbd061885260F176e05699fED9C5a4604fc7F2BDC`

Uniswap v3

Deploying idle collateral to stable-stable pairs on Uni v3 with FRAX

The key innovation of Uniswap v3's AMM algorithm allowing for LPs to deploy liquidity between specific price ranges allows for stablecoin-to-stablecoin pairs (e.g. FRAX-USDC) to accrue extremely deep liquidity within a tight peg. Compared to Uniswap v2, range orders in Uniswap v3 concentrate the liquidity instead of spreading out over an infinite price range.



(I) Uniswap v2 **(II)** A single position on $[p_a, p_b]$ **(III)** A collection of custom positions

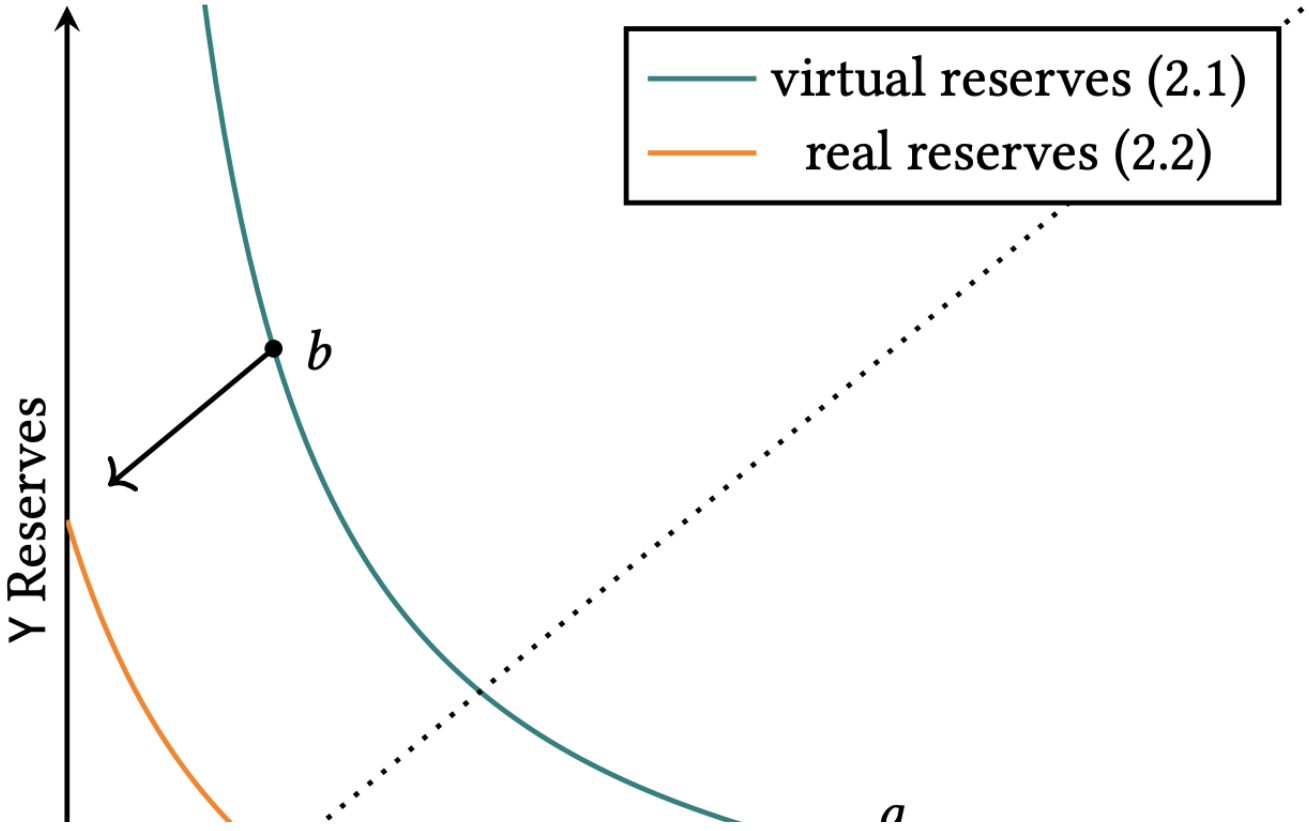
Taken from the Uniswap v3 whitepaper

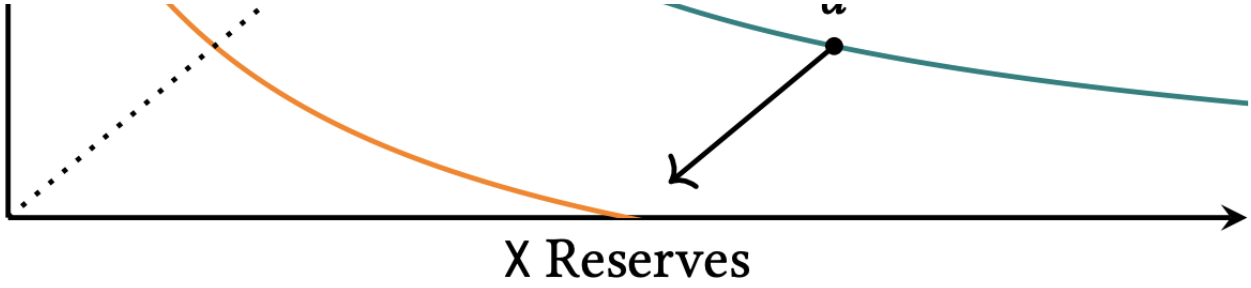
The Uniswap v3 Liquidity AMO puts FRAX and collateral to work by providing liquidity to other stablecoins against FRAX. Since the AMO is able enter any position on Uni v3 and mint FRAX against it, it allows for expansion to any other stablecoin and later volatile collateral on Uni v3. Additionally, the function `collectFees()` can be periodically called to allocate AMO profits to market operations of excess collateral.

AMO Specs

1. Decollateralize - Deposits idle collateral and newly minted FRAX into the a Uni v3 pair.
2. Market operations - Accrues Uni v3 transaction fees and swaps between collateral types.
3. Recollateralize - Withdraws from the Uni v3 pairs, burns FRAX and returns USDC to increase CR.
4. FXS1559 - Daily transaction fees accrued over the CR.

Derivation





A diagram showing range-order virtualized reserves using the constant-product invariant

All prices exist as ratios between one entity and another. Conventially, we select a currency as the shared unit-of-account in the denominator (e.g. USD) to compare prices for everyday goods and services. In Uniswap, prices are defined by the ratio of the amounts of reserves of x to reserves of y in the pool.

Uniswap v3's range-order mechanic fits into the existing $x * y = k$ constant-product market-making invariant (CPMM) by "virtualizing" the reserves at a specific price point, or `tick`. Through specifying which ticks a liquidity position is bounded by, range-orders are created that follow the constant-product invariant without having to spread the liquidity across the entire range $(0, \infty)$ for a specific asset.

A price in Uniswap v3 is defined by the value `1.0001` to the tick value i. The boundaries for the prices of ticks can be represented by the algebraic group $G = \{g^i \mid i \in \mathbb{Z}, g = 1.0001\}$. This mechanism allows for easy conversion of integers to price boundaries, and has the convenience of discretiating each tick-price-boundary as one basis point (`0.01%`) in price from another.

$$p(i) = 1.0001^i$$

Defining price in terms of ticks

Virtual reserves are tracked by tracking the `liquidity` and `tick` bounds of each position. Crossing a tick boundary, the liquidity L available for that tick may change to reflect positions entering and leaving their respective price ranges. Within the tick boundaries, swaps change the price $\sqrt{P}$ according to the virtual reserves, i.e. it acts like the constant-product ($x * y = k$) invariant. The virtual reserves `x` and `y` can be calculated from the liquidity and price:

$$L = \sqrt{xy}$$

$$\sqrt{P} = \sqrt{\frac{y}{x}}$$

L: Liquidity; P: Price; x, y: reserves of X and Y

Note that the actual implementation uses a square root of the price, since it saves a square-root operation from calculating intra-tick swaps, and thus helps prevent rounding errors.

$$\sqrt{p}(i) = \sqrt{1.0001^i} = 1.0001^{\frac{i}{2}}$$

Converting price to square root of ticks

Liquidity can be thought of as a virtual k in the $x * y = k$ CPMM, while ΔY corresponds to amount of asset Y and $\Delta\sqrt{P}$ represents the intra-tick price slippage.

$$L = \frac{\Delta Y}{\Delta\sqrt{P}}$$

Describing the relationship between liquidity, price, and the amount of one asset swapped

Since L is fixed for intra-tick swaps, ΔX and ΔY can be calculated from the liquidity and square root of the price. When crossing over a tick, the swap must only slip until the $\sqrt{P}$ boundary, and then re-adjust the liquidity available for the next tick.

Liquidity AMO

The Uniswap v3 Liquidity AMO (stable-stable) contract is deployed at:
`0x3814307b86b54b1d8e7B2Ac34662De9125F8f4E6`

Collateral Hedge

Helps stabilize the FRAX peg

This controller takes some amount of collateral and perfectly hedges against it with an inverse/short position to create a basis trade of net 0 volatility to minimize against significant drops in collateral price. This AMO would allow FRAX to be backed by more diverse and volatile collateral. For example, we are exploring using Synthetix inverse assets with the asset itself to form perfectly hedged units of collateral (ex: ETH-iETH).

FRAX Lending

Earns APY from lending out FRAX to DeFi platforms

This controller mints FRAX into money markets such as Compound or CREAM to allow anyone to borrow FRAX by paying interest instead of the base minting mechanism. FRAX minted into money markets don't enter circulation unless they are overcollateralized by a borrower through the money market so this AMO does not lower the direct collateral ratio (CR). This controller allows the protocol to directly lend FRAX and earn interest from borrowers through existing money markets. Effectively, this AMO is MakerDAO's entire protocol in a single market operations contract. The cash flow from lending can be used to buy back and burn FXS (similar to how MakerDAO burns MKR from stability fees). Essentially the Lending AMO creates a new avenue to get FRAX into circulation by paying an interest rate set by the money market.

AMO Specs

1. Decollateralize - Mints FRAX into money markets. The CR does not lower by the amount of minted FRAX directly since all borrowed FRAX are overcollateralized.
2. Market operations - Accrues interest revenue from borrowers.
3. Recollateralize - Withdraws minted FRAX from money markets.
4. FXS1559 - Daily interest payments accrued over the CR. (currently in development)

Adjusting Interest Rates and Capital Efficiency

The AMO can increase or decrease the interest rate on borrowing FRAX by minting more FRAX (lower rates) or removing FRAX and burning it (increase rates). This is a powerful economic lever since it changes the cost of borrowing FRAX on all lenders. This permeates all markets since the AMO can mint and remove FRAX to target a specific rate. This also effectively makes the cost of shorting FRAX more or less expensive depending on which direction the protocol wishes to target.

Additionally, the fractional-algorithmic design of the protocol allows for unmatched borrowing rates compared to other stablecoins. Because the Frax Protocol can mint FRAX stablecoins at will until the market responds with pricing FRAX at $.99 and recollateralizing the protocol, this means that money creation costs are minimal compared to other protocols. This creates unmatched, best-in-class rates for lending if the protocol decides to outcompete all other stablecoin rates. Thus, the AMO strategy can optimize for conditions for when to lower the rates (and also bring them under other stablecoin rates) and also increase rates in opposing conditions. Ironically, the lending rate on their own token is something other stablecoin projects have difficulty controlling. Frax has total control over this property through this AMO.

Tornado Cash

Mints FRAX directly into Tornado anonymity sets

Since FRAX aims to be an entirely permissionless, decentralized money, the next natural area of focus should be full privacy akin to traditional cash. To expand our privacy features, we've proposed making FRAX the newest asset added to Tornado.cash, the premier zk privacy tech on Ethereum. We've proposed an AMO Controller that would allow minting of FRAX directly into the FRAX-Tornado anonymity set which would use FRAX's expansion events as privacy enhancing ammunition which is a positive sum outcome for

the protocol since any increase in privacy is value generating. Arbitrageurs and users will have the ability to mint FRAX directly into Tornado.cash's anonymity set to mix new FRAX from genesis. The gas and optimization costs of this strategy are being explored to leverage L2 solutions. The eventual end goal for this AMO is to anonymize the entirety of incoming FRAX supply when gas/L2 optimization solutions are fully implemented.

Decentralization Ratio (DR)

Reducing reliance on centralized assets

The Frax Decentralization Ratio (DR) is the ratio of decentralized collateral value over the total stablecoin supply backed/redeemable for those assets. Collateral with excessive off-chain risk (i.e. fiatcoins, securities, & custodial assets such as gold/oil etc) are counted as 0% decentralized. The DR goes through underlying constituent pieces of collateral that a protocol has claims on, not just what is inside its system contracts. The DR is a recursive function to find the base value of every asset.

For example, FRAX3CRV LP is 50% FRAX so remove that, as you cannot back yourself with your own coin. The other half is 3CRV which is 33% USDC, 33% USDT, and 33% DAI. DAI itself is about 60% fiatcoins. So each $1 of FRAX3CRV LP only has about $.066 ($1 x 0.5 * 0.33 * 0.4) of value coming from decentralized sources.

In contrast, Ethereum, as well as reward tokens like CVX and CRV, are counted as 100% decentralized. FRAX minted through Lending AMOs also counts as decentralized since borrowers overcollateralize their loan w/ crypto sOHM, RGT, etc. This is the same reason DAI's vaults give it high DR.

The DR is a generalized algorithm that can be used to compute any stablecoin's excessive off-chain risk. Other stablecoins like LUSD are much easier to calculate: their DR is 100%. FEI is around 90% DR.

As of Nov 2, 2021, FRAX is roughly 40%. It is tracked daily and viewable at https://app.frax.finance/

veFXS & Gauges

veFXS

Locked FXS that provides multiple benefits

Background

veFXS is a vesting and yield system based off of Curve's veCRV mechanism. Users may lock up their FXS for up to 4 years for four times the amount of veFXS (e.g. 100 FXS locked for 4 years returns 400 veFXS). veFXS is not a transferable token nor does it trade on liquid markets. It is more akin to an account based point system that signifies the vesting duration of the wallet's locked FXS tokens within the protocol.

The veFXS balance linearly decreases as tokens approach their lock expiry, approaching 1 veFXS per 1 FXS at zero lock time remaining. This encourages long-term staking and an active community. Sushiswap has proposed a similar implementation with their recently announced oSushi token A sample graph (Curve's veCRV) illustrating the decrease can be found at this address.

veFXS Governance Whitelisting

Smart contracts & DAOs require whitelisting by governance to stake for veFXS. Only externally owned accounts and normal user wallets can directly call the veFXS stake locking function. In order to build veFXS functionality into your protocol, begin the governance process with the FRAX community at gov.frax.finance by submitting a whitelisting proposal.

Voting Power

Each veFXS will have 1 vote in governance proposals. Staking 1 FXS for the maximum time, 4 years, would generate 4 veFXS. This veFXS balance itself will slowly decay down to 1 veFXS after 4 years, at which time the user can redeem the veFXS back for FXS. In the meantime, the user can also increase their veFXS balance by locking up FXS, extending the lock end date, or both. It should be noted that veFXS is non-transferable and each account can only have a single lock duration meaning that a single address cannot lock certain FXS tokens for 2 years then another set of FXS tokens for 3 years etc. All FXS per account must have a uniform lock time.

Farming Boosts

Holding veFXS will give the user more weight when collecting certain farming rewards. All farming rewards that are distributed directly through the protocol are eligible for veFXS boosts. External farming that are promoted by other protocols (such as Sushi Onsen) are typically not available for veFXS boosts since they are independent of the Frax protocol itself. A user's veFXS boost does not increase the overall emission of rewards. The boost is an additive boost that will be added to each farmer's yield proportional to their veFXS balance. The veFXS boost can be different for each LP pair by the discretion of the community and team based on partnership agreements and governance votes.

Farming boosts are given in ratios of veFXS per 1 FRAX in LP. For example, a FRAX-IQ pair with a 2x boost ratio of 10 veFXS per 1 FRAX means that a user that has 50,000 veFXS gets a 2x boost for an LP position of $10,000

The current veFXS per FRAX requirements are 4 veFXS to 1 FRAX

Yield

After the implementation of veFXS, holders may be able to collect rewards periodically. The emission rate would vary, depending on the implementation of how the yield is obtained and market price of FXS.

Currently, the FXS rewards for holding veFXS come from the AMO profits of the protocol that is above the collateral ratio. 50% of the profits are being used to buy back FXS and burn it, and the other 50% is sent to the yield distributor contract for distribution to veFXS holders.

Future Functionality

The veFXS system is modular and all-purpose. In the future, it can be expanded to vote on AMO weights, earn additional yield in new places/features, and be treated as a governance token bond rate of sorts.

This **benefits Frax** as a whole by:

- Allocate voting power to long-term holders of FXS through veFXS
- Incentivizing farmers to stake FXS
- Creating a bond-like utility for FXS and create a benchmark APR rate for staked FXS

Staking guide

> Guide: How to stake your FXS and earn veFXS yield.
> Medium

Gauge

Gauge weighted system for controlling FXS emissions & FRAX expansions

Curve Finance introduced the gauge system for their CRV token emissions. Users lock their veCRV to vote on "weights" of different Curve liquidity pools. FRAX is introducing a similar system for any FRAX pair across various protocols.

The FRAX gauge system allows FXS holders to stake for up to 4 years to generate veFXS and vote where future FXS emissions are directed. Users can vote for FRAX gauge weights with their veFXS balance. They can distribute their voting power across multiple gauges or a single gauge. This allows veFXS holders who are the most long term users of the protocol to have complete control over the future FXS emission rate. Additionally, the gauge system lowers the influence of FRAX pairs where the majority of rewards are sold off since those LPs will not have veFXS to continue voting for their pair. This system strongly favors LP providers who continually stake their rewards for veFXS to increase their pool's gauge weight. Essentially, FRAX gauges align incentives of veFXS holders so that the most long term oriented FXS holders control where FXS emissions go until the full community FXS allocation is distributed. **Gauge weights are**

updated once every week every Wednesday at 5pm PST. This means that the FXS emission rate for each pair is constant for 1 week then updates to the new rate on each Wednesday. Any user can change their weight allocation every 10 days.

After all FXS has been distributed, the gauge system will transition to controlling FRAX stablecoin expansions as rewards for LPs. This switch from FXS to FRAX rewards will not occur for a few years until FXS emissions are nearing completion and allows the stablecoin to build trust, confidence, and Lindy effect first. Additionally, veFXS stakers can feel confident staking the maximum duration of 4 years knowing that the gauge program is not temporary and won't be deprecated when FXS rewards end. The gauge weights will simply transition to distributing FRAX stablecoins as rewards perpetually.

veFXS Boosts & Locked LP Boosts

Users who stake LP tokens in a gauge weight contract earn further boosts to their APR based on the amount of veFXS they have. Additionally, users that lock their LP tokens within the staking contract for a specific period of time will earn a further additive boost on top thus enabling stacking of both boosts for maximal APR. Since gauge weights change weekly, locked LPs in gauges do not get their LPs unlocked if the gauge weight changes. See the veFXS spec page for an explanation of how boosts are calculated.

Gauge Agnostic Pairs

FRAX gauges allow veFXS stakers to directly control the FXS emission rate to any pool that integrates FRAX. There is no restriction on which protocols or pairs can have a gauge weight other than they use FRAX stablecoins and pass the gauge governance vote. While Uniswap V3 stablecoin range gauges will be the debut gauge (and Curve's FRAX3CRV pool), any FRAX pool (including cross-chain pools) can be added as a gauge in the future. The veFXS gauge system is completely agnostic compared to Curve's because FRAX gauges simply require the stablecoin being in the pool while Curve requires that all gauges be a Curve pool. Essentially, veFXS gauges are the money layer gauge weights of DeFi while Curve gauge weights are the DEX layer weights. Since veFXS stakers can control emissions into any protocol that integrates FRAX, many protocols and communities might compete for controlling the future cash flow of an algorithmic stablecoin protocol.

It's important to note for any smart contract (non-EOA wallet) to stake veFXS, they must be whitelisted by a governance vote. For a full list of benefits of holding veFXS such as AMO profits and farming boosts, see the veFXS full specs.

Removing USDC Collateral Dependency

The gauge platform is the first phase in removing all "fiatcoin" dependence from the Frax Protocol through a system of dynamic liquidity incentives where any potential FRAX pair can be rewarded to create the exact balance of liquidity needed. Additionally, since gauges can be deployed for any protocol (and in the future to any cross-chain liquidity pool) the Frax Protocol has no claims on any of the underlying collateral in any gauge pools so blacklisting FRAX system contracts or preventing interaction with FRAX contracts will not have any effect on the stability of the peg. This system is conceivably as distributed as possible while retaining the ability to incentivize stabilization of a censorship resistant digital asset. A wide array of gauges on multiple protocols (and chains/L2s) is required to reach sufficient decentralization where it would be infeasibly difficult for a central actor (such as a fiatcoin issuer or state actor) to prevent pseudo-anonymous users from adding liquidity to gauge rewarded pools.

Cross Chain FRAX & FXS

Bridge Mechanism

Multichain FRAX+FXS that is fungible across many networks

The Frax Protocol is a multichain protocol with global state consistent across all deployments. FRAX+FXS tokens are a single distribution across all networks. There is no independent Frax implementation for each chain. For this reason, the protocol has a bridging system that allows it to maintain a tight peg and fungibility in a unique & novel way.

The protocol treats each individual bridged FRAX/FXS as a unique liability of that bridge system and names FRAX/FXS moved from other chains with the identifier of that bridge. For example, AnySwap bridged FRAX is referred to as anyFRAX and FRAX bridged with the Wormhole bridge is called wormFRAX.

Each chain has 1 canonical FRAX and canonical FXS contract that are simply referred to as "FRAX" and "FXS" (with no prefix). These tokens are what AMOs expand/contract and users themselves can trustlessly mint/redeem.

Canonical (native) FRAX/FXS: The pure protocol liability natively issued/minted/redeemed by the protocol & AMOs. Canonical FRAX has the default colors of that specific network and the logo of that network at the center. Native FXS has the native colors and logo of the chain at the bottom right. Thus, any canonical/native FRAX/FXS always displays the network logo somewhere on the coin.

Bridged FRAX/FXS: Tokens that are brought to the current chain from another network using a supported bridge protocol. The naming convention for bridged tokens maintains a prefix designation for the bridge used to bring them to the current network. Ex: AnySwap bridged FRAX from ETH to AVAX is known as anyFRAX on AVAX while it is simply canonical FRAX on ETH. The colors and logo of the bridge protocol are prominently display on the FRAX/FXS token to clearly distinguish which bridge they originated from within the current network.

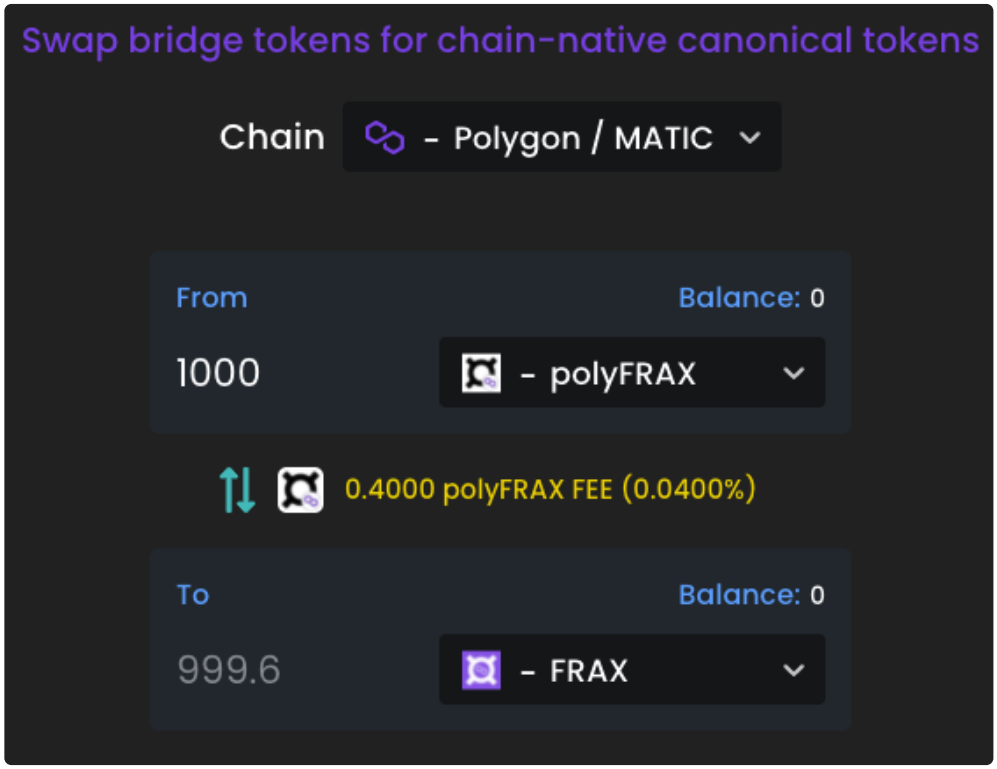
Swapping FRAX/FXS and Peg Arbitrage Between Chains

Each canonical FRAX/FXS ERC20 token contract has a 1 to 1 stableswap AMM built into the token which allows swapping to/from the canonical FRAX/FXS of the network for any supported bridged FRAX/FXS. This allows tight arbitrage of the FRAX peg and also maintains the single distribution of FXS across all chains. For example, let's assume that canonical FRAX on Fantom is $.990. An arbitrager can purchase as much canonical FRAX as possible at $.990 knowing that she can swap them 1 to 1 for anyFRAX in the stableswap pool within the ERC20 token contract then bridge the anyFRAX back to ETH mainnet (or any other chain) where FRAX is at peg to make a profit.

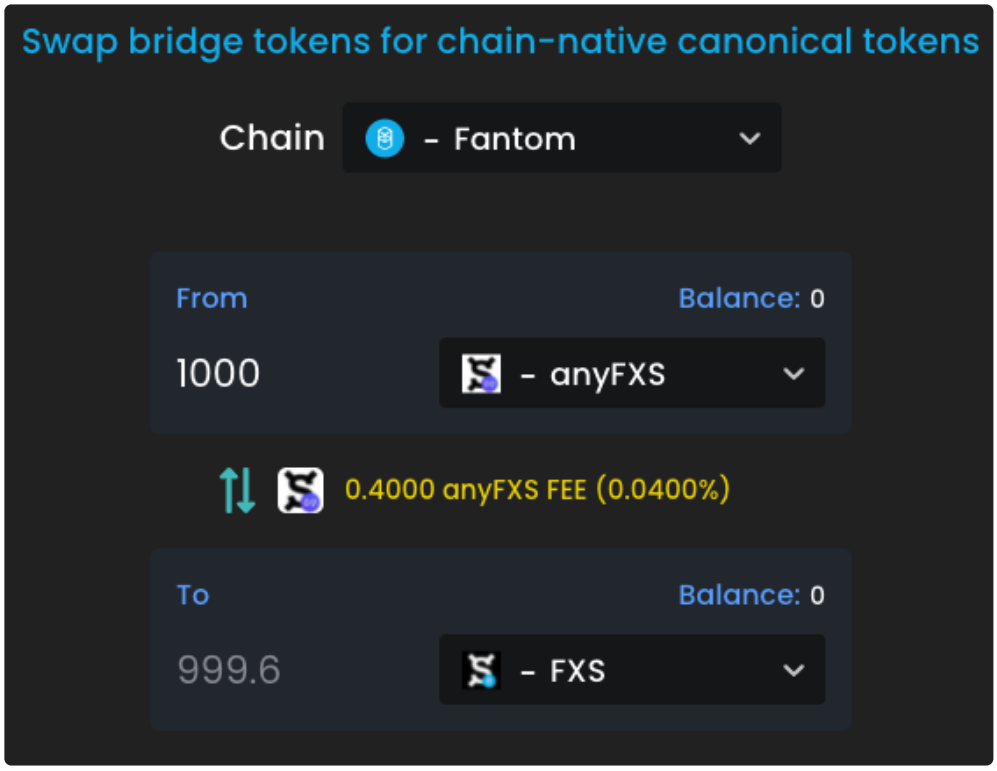
Therefore, purchasing canonical FRAX/FXS on one chain is the same as purchasing FRAX/FXS on another chain. If you bridge FRAX/FXS using any of the supported bridges (more to be added soon), you can swap the bridged token for native FRAX/FXS on that chain to farm/LP/hold etc. Additionally, when canonical FRAX/FXS is minted with AMOs on any chain, the protocol checks that there is enough swap liquidity

available with the token contract to move canonical tokens across chains so that the peg is always global

Swaps can be done any time at https://app.frax.finance/crosschain or interacting directly with the native token's smart contract on any chain. The swap mechanism is built into the native ERC20 FRAX/FXS tokens on every chain (except Ethereum L1).



Any example swap on https://app.frax.finance/crosschain of FRAX bridged from the Polygon PoS bridge (named polyFRAX) for canonical, native issued FRAX on Polygon.



An example swap on https://app.frax.finance/crosschain of AnySwap bridged FXS (named anyFXS) for canonical, native FXS on Fantom.

Guide

NOTE: This guide is for Fantom, but is applicable to other chains. For Polygon, use their proprietary bridge for the bridging step.

> Guide: How to swap your bridged FRAX & FXS to canonical native tokens.
> Medium

Canonical Token Addresses

FRAX: https://docs.frax.finance/smart-contracts/frax

FXS: https://docs.frax.finance/smart-contracts/fxs

Frax Price Index

Inflation Hedge

System overview

The Frax Price Index is a brand new, unique protocol within the Frax ecosystem. Centered around its own native stablecoin (FPI) and governance token (FPIS), the system will adjust every month according to an on-chain Consumer Price Index oracle so that holders of the FPI will increase their dollar-denominated value each month according to the reported CPI increase. It does this by earning yield on the underlying FPI treasury, created from users minting and redeeming FPI with FRAX.

The Frax Price Index Share (FPIS) token is the governance token of the system, which is also entitled to seigniorage from the protocol. Excess yield will be directed from the treasury to FPIS holders, similar to the FXS structure. During times in which the FPI treasury does not create enough yield to maintain the increased backing per FPI due to inflation, new FPIS may be minted and sold to increase the treasury. Since the protocol is launched from within the Frax ecosystem, the FPIS token will also direct a variable part of its revenue to FXS holders.

Mandate

The Frax Price Index has a two-fold mandate:

1. Maintain the $FPI peg (tracking monthly inflation)
2. Accrue value to the $FXS (Frax Share) token

To maintain the mandate, $FPIS token holders are encouraged to remain aware of the purpose of the protocol.

Airdrop + Token Distribution

Token distribution information

The $FPIS snapshot occured on February 21st, midnight UTC at block height `14246086`. The resulting eligible addresses have been posted to github (link), with claims available preceding system release at a later date.

The airdrop will be weighted as such:

- veFXS holders will receive 1.0 units of airdrop weight per 1 veFXS
- tFXS holders will receive 1.0 units of airdrop weight per 1 tFXS
- FRAX-FXS Uni v2 LP stakers will receive 0.5 units of airdrop weight per the boosted balance (timelock boost) of the account's FRAX-FXS Uni v2 LP inside the staking contract. Note that the tokens **must be staked** inside the staking contract to be eligible for the airdrop.

Note: for Convex cvxFXS depositors, users will be able to claim their airdrop through Convex at a later point after the airdrop is released.

Example

Say a given user has 10 veFXS, 10 tFXS, and a Uni v2 position of 5 FXS against 125 FRAX.

- 10 veFXS -> 10 airdrop units
- 10 tFXS -> 10 airdrop units
- 5 FXS against 125 FRAX in Uni v2
  - $\sqrt{5 * 125} = \sqrt{625} = 25$ Uni v2 LP tokens
  - 25 Uni v2 LP tokens staked, no timelock -> 12.5 airdrop units
  - 25 Uni v2 LP tokens staked, 3 year lock -> 37.5 airdrop units

If the above user chooses not to timelock their Uni v2 stake, they would receive (10 + 10 + 12.5 = 32.5) units of the $FPIS airdrop.

Unit of Account

Protocol vision

The $FPI token aims to be the first on-chain stablecoin to have its own unit of account derived from a basket of goods, both crypto and non-crypto. At first, the treasury will be comprised solely of $FRAX, but will look to include other crypto-native assets such as bridged BTC, ETH, and non-crypto consumer goods and services.

Token Distribution

Frax Share (FXS)

The distribution of FXS across the system

Community (65% – 65,000,000 FXS)

DeFi Protocols have made use of liquidity programs to jumpstart growth and distribute protocol tokens to community members. To that end, 60% of all FXS tokens are to be distributed through various yield farming, liquidity incentives, and exclusive governance proposals across a number of years.

60% – Liquidity Programs / Farming / Community – Via gauges & governance halving naturally every 12 months

Since FIP-16 veFXS gauges, the token distribution has changed to `25,000 FXS` per day to the FXS gauges. The original FRAX-FXS Uniswap v2 pool still accrues `16,438.37 FXS` per day.

Per the original design specs for FXS distribution, the FXS supply will halve every 12 months on December 20th each year. This means that on December 20th, 2021 the gauge emission will reduce by 50% to `12,500 FXS` per day and FRAX-FXS Uniswap v2 to `8,219.18 FXS` per day. These changes will not unlock locked LPs as they are the normal emission schedule of the FXS supply.

As DeFi is an evolving landscape, these emissions can be changed by a full Frax Improvement Proposal (FIP) governance vote where LP locks and boost weights can be redone if the FIP is passed. Full votes require 2 weeks of discussion followed by a token holder vote per the official governance process.

Community governance can decide which pools, programs, and initiatives to support with the emission schedule, but it cannot be increased past the `100,000,000 FXS` supply max. Thus, a maximum of `60,000,000 FXS` will be distributed to the community for liquidity programs and other DeFi initiatives as they appear in the space as voted by governance. New programs can be added by governance to the remaining allocation, but no more than `60,000,000 FXS` can be allocated due to the hard cap of `100,000,000 FXS` distributed. This is to put a hard cap on the amount of FXS as well as to put a hard duration on the number of years required to distribute the FXS. This emission rate was chosen to balance the need for a large amount of rewards for early adopters while not distributing all FXS too early which is

needed for long term community sustainability. The FXS emission should be thought of and modeled more after Bitcoin mining than anything else. FXS distribution needs to be multi-year, extended, and sustainable until the protocol reaches ubiquity.

5% – Project Treasury / Grants / Partnerships / Security-Bug-Bounties – via Team and Community discretion
The Project Treasury is an entirely community and team governed pool of FXS. It should be used for making grants for development of the Frax technology, open source upkeep of the code, future audits of smart contracts, bug bounties through responsible disclosure, possible cross-chain implementations, creation of new protocol level features and updates, Gitcoin grants for the Ethereum community, Frax Improvement Proposals (FIPs), partnerships with exchanges and DeFi projects, providing liquidity on AMMs at launch. The usage of this fund is dependent on the discretion of the team and community.

Team and Investors (35% – 35,000,000 FXS)

20% – Team / Founders / Early Project Members – 12 months, 6 month cliff
Team tokens are retained for founders and original early contributors to Frax. The Frax Protocol was conceived in late 2018 and work began in early 2019. The Frax concept has been over 2 years old since conception. Although, the mainnet is just now being launched, the contributions of founders and early members that have been working on Frax was crucial to releasing the protocol. The team will continue to work on Frax for its lifetime along with the greater community.

3% – Strategic Advisors / Outside Early Contributors – 36 months
Advisory tokens which are allotted for strategic work done in legal, technical, and business efforts to advance the adoption of the Frax protocol. The tokens are vested evenly over 3 years.

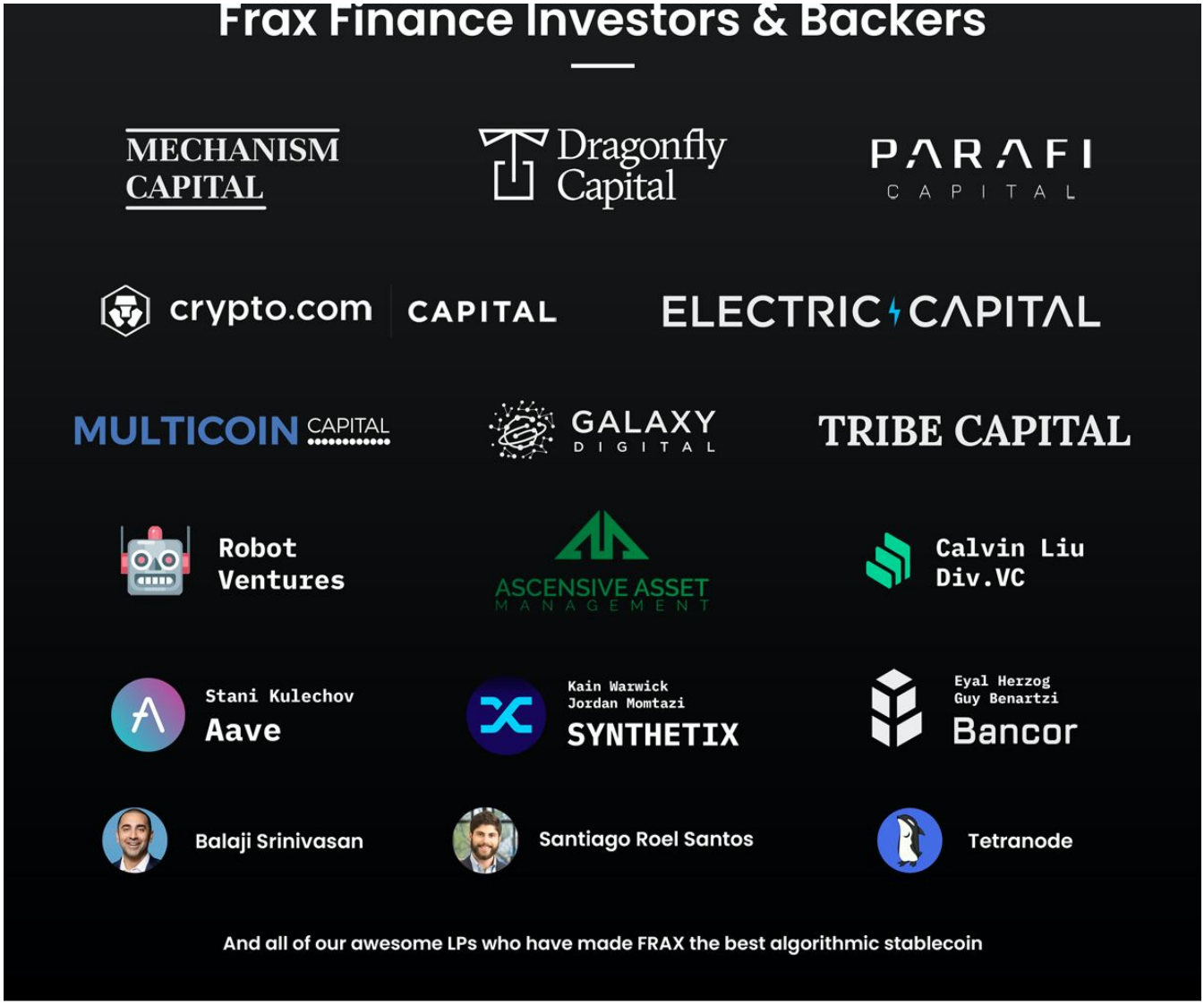
12% – Accredited Private Investors – 2% unlocked at launch, 5% vested over the first 6 months, 5% vested over 1 year with a 6 month cliff
The first round in Frax was done in August of 2020 with a small allocation that was sold out in under 2 hours. This allocation will have a small amount of their tokens, ~2% unlocked at launch. The remainder of the round was done individually through private placements.The remaining 10% is vested evenly over 1 year, half of which has a 6 month cliff.

Multisig Addresses

Multisigs
Frax Finance ¤

List of addresses where the above funds are stored

Investors and Backers



Guides & FAQ

FAQ

Why use a fractional stablecoin?

TLDR: capital efficiency, as well as decentralization.

Staking

How to add Liquidity & Stake on Uniswap to get daily FXS rewards:

Guide : How to add Liquidity to Frax Finance Pools on UNISWAP.
Medium

How to add Liquidity & Stake on Curve to get CRV rewards:

Guide : How to provide liquidity to FRAX3CRV Pool on Curve.fi.
Medium

How to stake your FXS and earn veFXS yield:

Guide: How to stake your FXS and earn veFXS yield.
Medium

Uniswap FRAX / IQ staking

https://everipedia.org/blog/guide-how-to-add-liquidity-to-frax-iq-pool-on-uniswap-and-receive-i…

Earn Rewards using the Saddle D4 Pool
Medium

How to add liquidity to FRAX3CRV pool and stake your LPs on StakeDao

Guide: How to add liquidity to FRAX3CRV pool and stake your LPs on StakeDao.
Medium

Uniswap Migration / Uniswap V3

Migrating from Uniswap V2 to Uniswap V3

How to migrate your UNI-V2 FRAX-USDC LPs to UNI-V3

Guide: How to migrate your UNI-V2 FRAX-USDC LPs to UNI-V3.
Medium

How to add liquidity to the FRAX-USDC pool on UNISWAP V3

Guide: How to add liquidity to FRAX-USDC pool on UNISWAP V3.
Medium

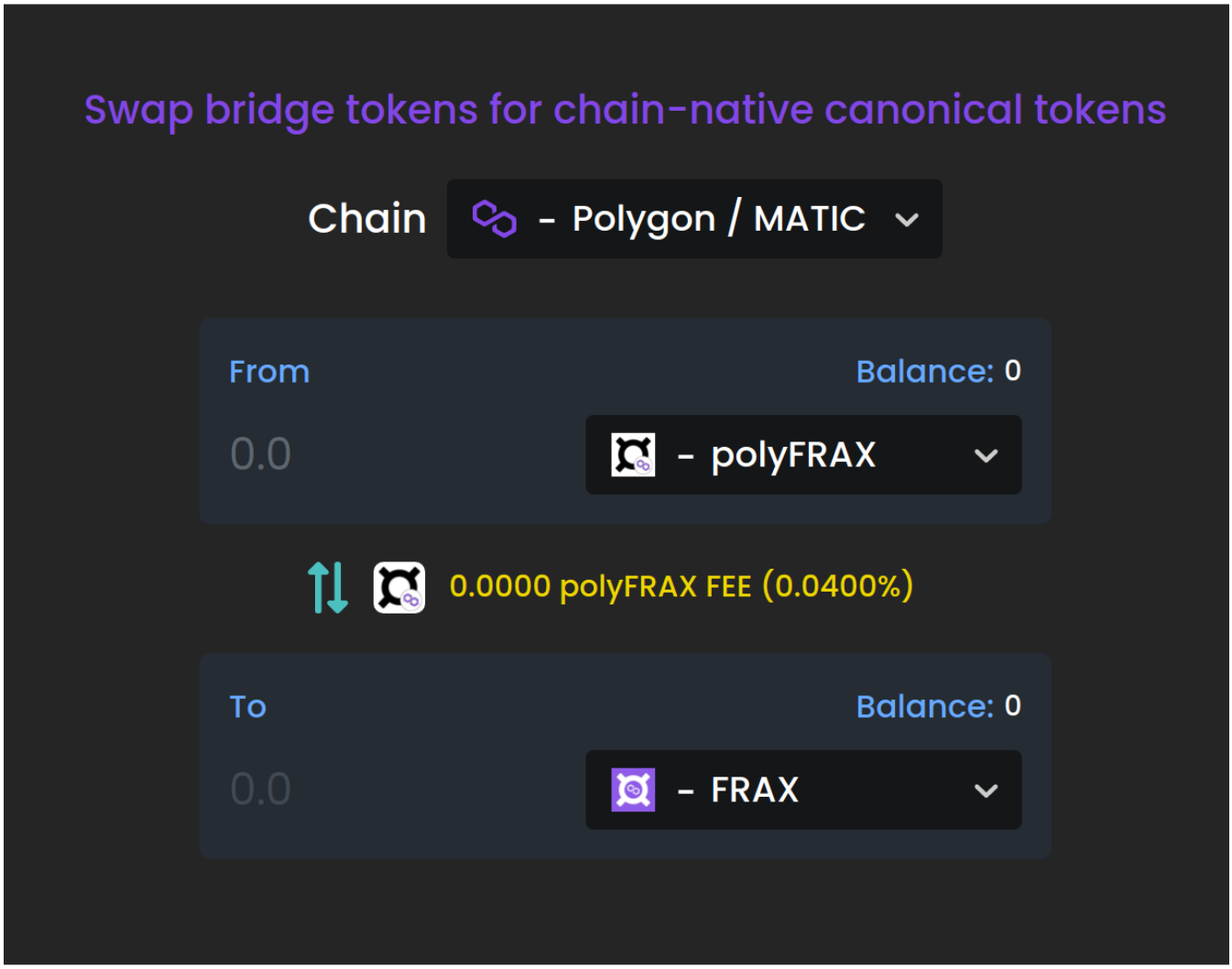
Matic / Polygon

How to bridge FRAX & FXS to Polygon (Matic) network and add

liquidity to Sushi.

NOTE: Once you have completed the Polygon wallet/bridge step, those tokens can be exchanged for native Polygon FRAX or FXS via https://app.frax.finance/crosschain

How to bridge FRAX & FXS to Polygon (Matic) network and add liquidity to Sushi.
Medium



https://app.frax.finance/crosschain

How to bridge FRAX to Polygon (Matic) network and add liquidity to mStable.

NOTE: The mStable farm uses anyFRAX (PoS FRAX, the bridge token), NOT Polygon native FRAX

Guide: How to bridge FRAX to Polygon (Matic) network and add liquidity to mStable.
Medium

Avalanche

How to get Avalanche (AVAX)

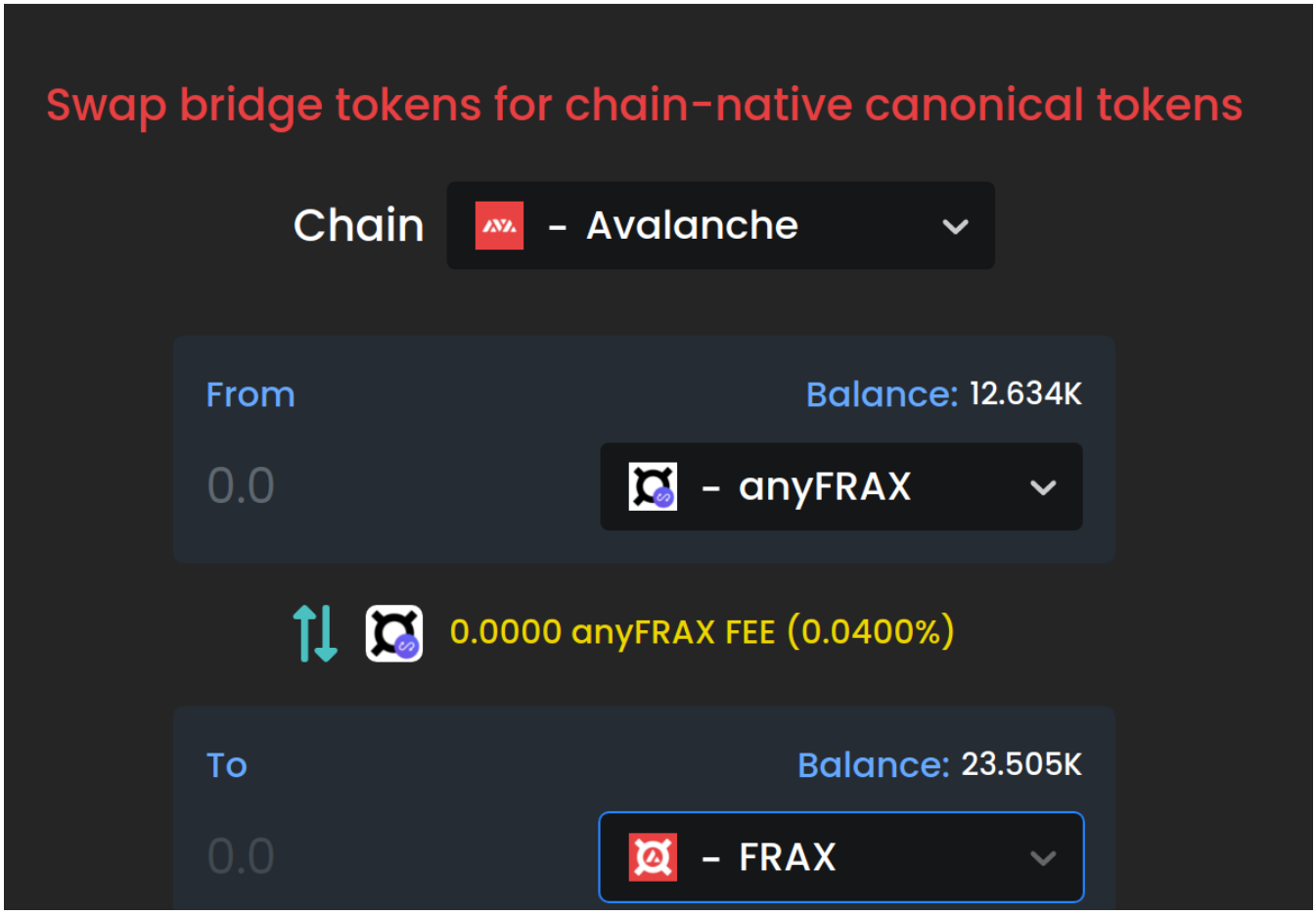
NOTE: Once you have completed the AnySwap bridge step, those tokens can be exchanged for native Avalanche FRAX or FXS via https://app.frax.finance/crosschain

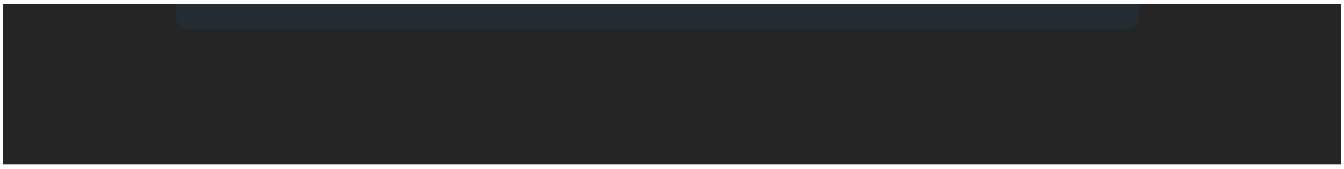
1) Go to a CEX and purchase AVAX. Here is a list:

> Avalanche (AVAX) price today, chart, market cap & news | CoinGecko
> CoinGecko

2) Follow these instructions to move the AVAX to the Avalanche C-Chain:

> Getting Started - Pangolin
> pangolindex





https://app.frax.finance/crosschain

How to bridge FRAX and FXS to Avalanche and add liquidity to Snowball

Guide: How to bridge FRAX to Avalanche network and add liquidity to Snowball.
Medium

---

how to swap your bridged FRAX to canonical native tokens on the Avalanche network and swap them on Axial

A guide on how to swap your bridged FRAX to canonical native tokens on the Avalanche net…
Medium

BSC

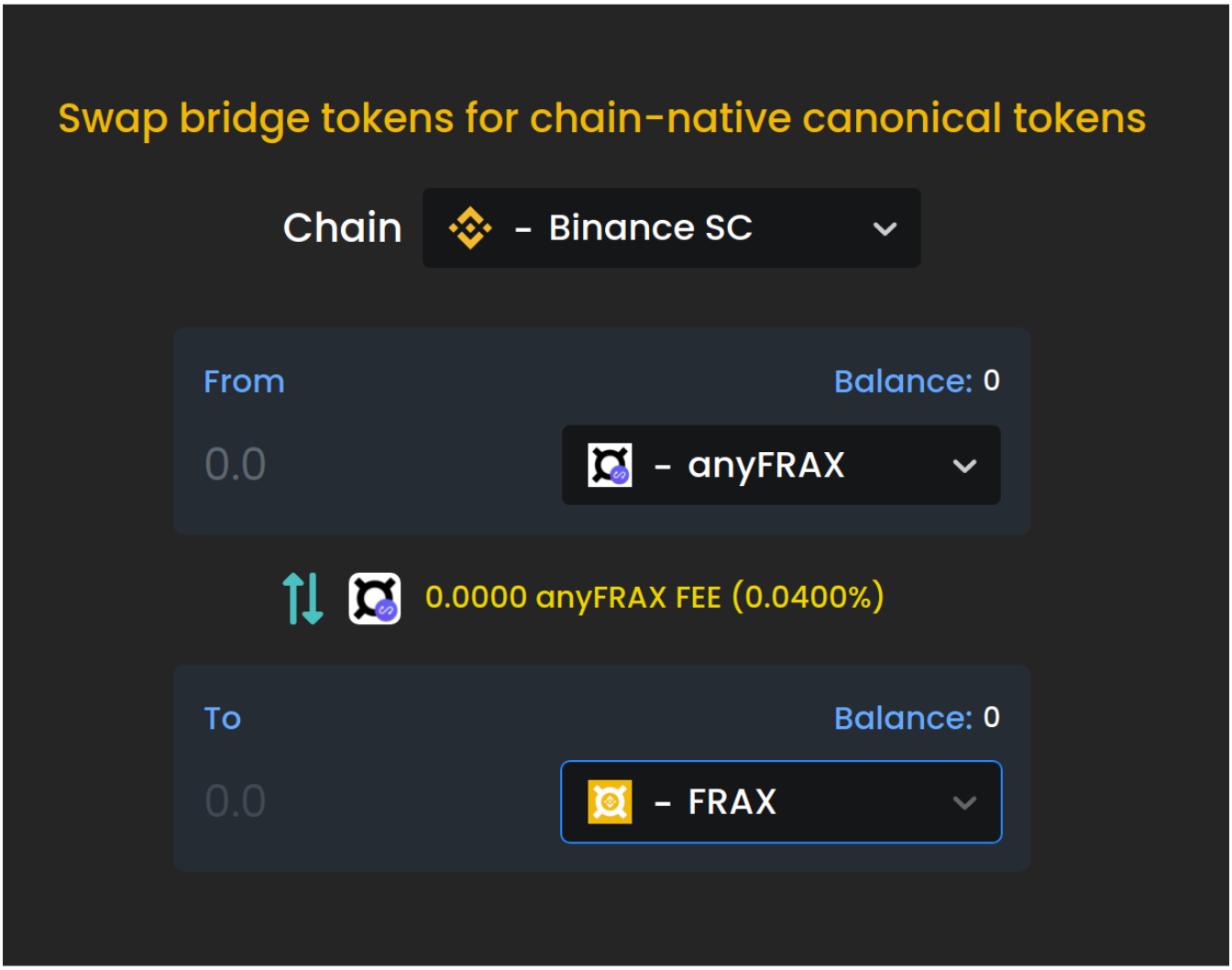
How to bridge FRAX & FXS to Binance smart chain (BSC) and add liquidity to Impossible Finance.

NOTE: Bridging to BSC can now be done via Binance's bridge or AnySwap. Those bridge tokens can be exchanged for native BSC FRAX or FXS via https://app.frax.finance/crosschain

THIS GUIDE BELOW IS FOR FANTOM, BUT THE ANYSWAP STEPS FOR BSC ARE ALMOST IDENTICAL.

Guide: How to bridge FRAX to Fantom network and add liquidity to SpiritSwap.

Medium



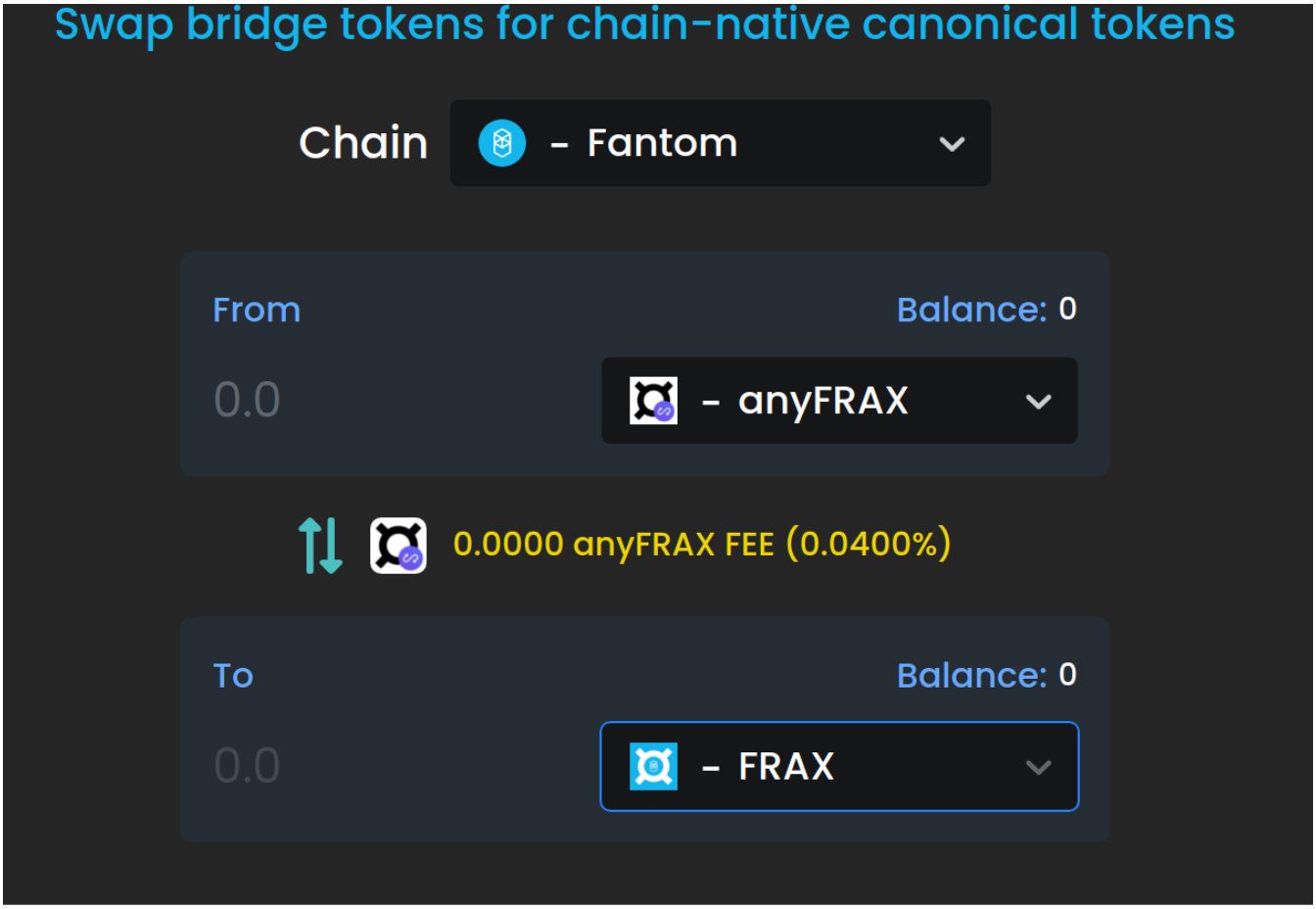
https://app.frax.finance/crosschain

Fantom

How to bridge FRAX to Fantom and add liquidity to SpiritSwap

NOTE: Once you have completed the AnySwap bridge step, those tokens can be exchanged for native Fantom FRAX or FXS via https://app.frax.finance/crosschain

Guide: How to bridge FRAX to Fantom network and add liquidity to SpiritSwap.
Medium



https://app.frax.finance/crosschain

Harmony

How to bridge FRAX & ETH to Harmony and add liquidity to Sushi

Guide: How to bridge FRAX & ETH to Harmony and add liquidity to Sushi.
Medium

Moonriver

Guide: How to swap your bridged FRAX-USDC to native tokens and add liquidity on Moonriver (SUSHI).

Guide: How to swap your bridged FRAX-USDC to native tokens and add liquidity on Moonrive…
Medium

Smart Contracts

Frax (FRAX)

Modified ERC-20 Contract representing the FRAX stablecoin.

Deployments

Chain	Address
Arbitrum	0x17FC002b466eEc40DaE837Fc4bE5c67993ddBd6F
Aurora	0xE4B9e004389d91e4134a28F19BD833cBA1d994B6
Avalanche	0xD24C2Ad096400B6FBcd2ad8B24E7acBc21A1da64
Boba	0x7562F525106F5d54E891e005867Bf489B5988CD9
BSC	0x90C97F71E18723b0Cf0dfa30ee176Ab653E89F40
Ethereum	0x853d955aCEf822Db058eb8505911ED77F175b99e
Fantom	0xdc301622e621166BD8E82f2cA0A26c13Ad0BE355
Harmony	0xFa7191D292d5633f702B0bd7E3E3BcCC0e633200
Moonbeam	0x322E86852e492a7Ee17f28a78c663da38FB33bfb
Moonriver	0x1A93B23281CC1CDE4C4741353F3064709A16197d
Optimism	0x2E3D870790dC77A83DD1d18184Acc7439A53f475
Polygon	0x45c32fA6DF82ead1e2EF74d17b76547EDdFaFF89
Solana	FR87nWEUxVgerFGhZM8Y4AggKGLnaXswr1Pd8wZ4kZcp

State Variables

ERC-20 (Inherited)

https://docs.openzeppelin.com/contracts/2.x/api/token/erc20#ERC20

AccessControl (Inherited)

https://docs.openzeppelin.com/contracts/3.x/api/access#AccessControl

NOTE: FRAX & FXS contracts have no pause or blacklist controls in any way (including system contracts).

FRAX-Specific

```
enum PriceChoice { FRAX, FXS }
```

An enum declaring FRAX and FXS. Used with oracles.

```
ChainlinkETHUSDPriceConsumer eth_usd_pricer
```

Instance for the Chainlink ETH / USD trading. Combined with FRAX / WETH, FXS / WETH, collateral / FRAX, and collateral / FXS trading pairs, can be used to calculate FRAX/FXS/Collateral prices in USD.

```
uint8 eth_usd_pricer_decimals
```

Decimals for the Chainlink ETH / USD trading pair price.

```
UniswapPairOracle fraxEthOracle
```

Instance for the FRAX / WETH Uniswap pair price oracle.

```
UniswapPairOracle fxsEthOracle
```

Instance for the FXS / WETH Uniswap pair price oracle.

```
address[] public owners
```

Array of owner address, who have privileged actions.

```
address governance_address
```

Address of the governance contract.

```
address public creator_address
```

Address of the contract creator.

```
address public timelock_address
```

Address of the timelock contract.

```
address public fxs_address
```

Address of the FXS contract

```
address public frax_eth_oracle_address
```

Address for the `fraxEthOracle` .

```
address public fxs_eth_oracle_address
```

Address for the `fxsEthOracle` .

```
address public weth_address
```

Address for the canonical wrapped-Ethereum (WETH) contract. Should be `0xc02aaa39b223fe8d0a0e5c4f27ead9083c756cc2` for the mainnet.

```
address public eth_usd_consumer_address
```

Address for the `ChainlinkETHUSDPriceConsumer` .

```
uint256 public genesis_supply
```

Genesis supply of FRAX. Should be a small nonzero amount. Most of the FRAX supply will come from minting, but a small amount is needed initially to prevent divide-by-zero errors in various functions.

```
address[] frax_pools_array
```

Array of all the `FraxPool` contract addresses.

```
mapping(address => bool) public frax_pools
```

Essentially the same as `frax_pools_array` , but in mapping form. Useful for gas savings in various functions like `globalCollateralValue()` .

```
```

```
1 uint256 public global_collateral_ratio
```

The current ratio of FRAX to collateral, over all `FraxPool` s.

```
1 uint256 public redemption_fee
```

The fee for redeeming FRAX for FXS and/or collateral. Also the fee for buying back excess collateral with FXS. See the `FraxPool` contract for usage.

```
1 uint256 public minting_fee
```

The fee for minting FRAX from FXS and/or collateral. See the `FraxPool` contract for usage.

```
1 address public DEFAULT_ADMIN_ADDRESS
```

Set in the constructor. Used in `AccessControl` .

```
1 bytes32 public constant COLLATERAL_RATIO_PAUSER
```

A constant used in the pausing of the collateral ratio.

```
1 bool public collateral_ratio_paused
```

Whether or not the collateral ratio is paused.

View Functions

oracle_price

```
1 oracle_price(PriceChoice choice) internal view returns (uint256)
```

Get the FRAX or FXS price, in USD.

frax_price

```
1 frax_price() public view returns (uint256)
```

Returns the price for FRAX from the FRAX-ETH Chainlink price oracle.

fxs_price

```
1 fxs_price() public view returns (uint256)
```

Returns the price for FXS from the FXS-ETH Chainlink price oracle.

frax_info

```
1 frax_info() public view returns (uint256, uint256, uint256, uint256, uint256, uint256, uin
```

Returns some commonly-used state variables and computed values. This is needed to avoid costly repeat calls to different getter functions. It is cheaper gas-wise to just dump everything and only use some of the info.

globalCollateralValue

```
1 globalCollateralValue() public view returns (uint256)
```

Iterate through all FRAX pools and calculate all value of collateral in all pools globally. This uses the oracle price of each collateral.

Public Functions

refreshCollateralRatio

```
1 refreshCollateralRatio() public
```

This function checks the price of FRAX and refreshes the collateral ratio if the price is not $1. If the price is above $1, then the ratio is lowered by .5%. If the price is below $1, then the ratio is increased by .5%. Anyone can poke this function to change the ratio. This function can only be called once every hour.

Restricted Functions

mint

```
1 mint(uint256 amount) public virtual onlyByOwnerOrGovernance
```

Public implementation of internal _mint().

pool_burn_from

```
1 pool_burn_from(address b_address, uint256 b_amount) public onlyPools
```

Used by pools when user redeems.

pool_mint

```
pool_mint(address m_address, uint256 m_amount) public onlyPools
```

This function is what other frax pools will call to mint new FRAX.

addPool

```
addPool(address pool_address) public onlyByOwnerOrGovernance
```

Adds collateral addresses supported, such as tether and busd, must be ERC20.

removePool

```
removePool(address pool_address) public onlyByOwnerOrGovernance
```

Remove a pool.

setOwner

```
setOwner(address owner_address) public onlyByOwnerOrGovernance
```

Sets the admin of the contract

setFraxStep

```
setFraxStep(uint256 _new_step) public onlyByOwnerOrGovernance
```

Sets the amount that the collateral ratio will change by upon an execution of refreshCollateralRatio(),

setPriceTarget

```
setPriceTarget(uint256 _new_price_target) public onlyByOwnerOrGovernance
```

Set the price target to be used for refreshCollateralRatio() (does not affect minting/redeeming).

setRefreshCooldown

```
setRefreshCooldown(uint256 _new_cooldown) public onlyByOwnerOrGovernance
```

Set refresh cooldown for refreshCollateralRatio().

setRedemptionFee

```
setRedemptionFee(uint256 red_fee) public onlyByOwnerOrGovernance
```

Set the redemption fee.

setMintingFee

```
setMintingFee(uint256 min_fee) public onlyByOwnerOrGovernance
```

Set the minting fee.

setFXSAddress

```
setFXSAddress(address _fxs_address) public onlyByOwnerOrGovernance
```

Set the FXS address.

setETHUSDOracle

```
setETHUSDOracle(address _eth_usd_consumer_address) public onlyByOwnerOrGovernance
```

Set the ETH / USD oracle address.

setFRAXEthOracle

```
setFRAXEthOracle(address _frax_addr, address _weth_address) public onlyByOwnerOrGovernance
```

Sets the FRAX / ETH Uniswap oracle address

setFXSEthOracle

```
setFXSEthOracle(address _fxs_addr, address _weth_address) public onlyByOwnerOrGovernance
```

Sets the FXS / ETH Uniswap oracle address

toggleCollateralRatio

```
toggleCollateralRatio() public onlyCollateralRatioPauser
```

Toggle pausing / unpausing the collateral ratio.

Events

FRAXBurned

```
FRAXBurned(address indexed from, address indexed to, uint256 amount)
```

Emitted when FRAX is burned, usually from a redemption by the pool.

Modifiers

onlyCollateralRatioPauser

```
onlyCollateralRatioPauser()
```

Restrict actions to the designated collateral ratio pauser.

onlyPools

```
onlyPools()
```

Restrict actions to pool contracts, e.g. minting new FRAX.

onlyByGovernance

```
onlyByGovernance()
```

Restrict actions to the governance contract, e.g. setting the minting and redemption fees, as well as the oracle and pool addresses.

onlyByOwnerOrGovernance

```
onlyByOwnerOrGovernance()
```

Restrict actions to the governance contract or owner account(s), e.g. setting the minting and redemption fees, as well as the oracle and pool addresses.

Frax Share (FXS)

Modified ERC-20 Contract representing the FXS token, which is used for staking and governance actions surrounding the FRAX stablecoin.

Deployments

Chain	Address
Arbitrum	0x9d2F299715D94d8A7E6F5eaa8E654E8c74a9 8A7
Aurora	0xBb8831701E68B99616bF940b7DafBeb4CDb2 e0b

Avalanche	0x214DB107654fF987AD859F34125307783fC8e387
Boba	0xae8871A949F255B12704A98c00C2293354a3013
BSC	0xe48A3d7d0Bc88d552f730B62c006bC925eadEeE
Ethereum	0x3432B6A60D23Ca0dFCa7761B7ab56459D9C64D0
Fantom	0x7d016eec9c25232b01F23EF992D98ca97fc2A5a
Harmony	0x0767D8E1b05eFA8d6A301a65b324B6b66A1CC14c
Moonbeam	0x2CC0A9D8047A5011dEfe85328a6f26968C8aA1C
Moonriver	0x6f1D1Ee50846Fcbc3de91723E61cb68CFa6DE98
Optimism	0x67CCEA5bb16181E7b4109c9c2143c24a1c225Be
Polygon	0x1a3acf6D19267E2d3e7f898f42803e90C921902
Solana	6LX8BhMQ4Sy2otmAWj7Y5sKd9YTVVUgfMsBz6B9W7ct

State Variables

ERC-20 (Inherited)

https://docs.openzeppelin.com/contracts/2.x/api/token/erc20#ERC20

AccessControl (Inherited)

https://docs.openzeppelin.com/contracts/3.x/api/access#AccessControl

FXS-Specific

```
address public FRAXStablecoinAdd
```

Address of the FRAX contract.

```
uint256 genesis_supply
```

Genesis supply of FXS.

```
uint256 public maximum_supply
```

Maximum supply of FXS.

```
uint256 public FXS_DAO_min
```

Minimum FXS required to join DAO groups.

```
address public owner_address
```

Address of the contract owner.

```
address public oracle_address
```

Address of the oracle.

```
address public timelock_address
```

Address of the timelock.

```
FRAXStablecoin private FRAX
```

The FRAX contract instance.

```
struct Checkpoint {
    uint32 fromBlock;
    uint96 votes;
}
```

From Compound Finance. Used for governance voting.

```
mapping (address => mapping (uint32 => Checkpoint)) public checkpoints
```

List of voting power for a given address, at a given block.

```
mapping (address => uint32) public numCheckpoints
```

Checkpoint count for an address.

Restricted Functions

setOracle

```
1 setOracle(address new_oracle) external onlyByOracle
```

Change the address of the price oracle.

setFRAXAddress

```
1 setFRAXAddress(address frax_contract_address) external onlyByOracle
```

Set the address of the FRAX contract.

setFXSMinDAO

```
1 setFXSMinDAO(uint256 min_FXS) external onlyByOracle
```

Set minimum FXS required to join DAO groups.

mint

```
1 mint(address to, uint256 amount) public onlyPools
```

Mint new FXS tokens.

pool_mint

```
1 pool_mint(address m_address, uint256 m_amount) external onlyPools
```

This function is what other FRAX pools will call to mint new FXS (similar to the FRAX mint).

pool_burn_from

```
1 pool_burn_from(address b_address, uint256 b_amount) external onlyPools
```

This function is what other FRAX pools will call to burn FXS.

Overridden Public Functions

transfer

```
1 transfer(address recipient, uint256 amount) public virtual override returns (bool)
```

Transfer FXS tokens.

transferFrom

```
1 transferFrom(address sender, address recipient, uint256 amount) public virtual override re
```

Transfer FXS tokens from another account. Must have an allowance set beforehand.

Public Functions

getCurrentVotes

```
1 getCurrentVotes(address account) external view returns (uint96)
```

Gets the current votes balance for `account`.

getPriorVotes

```
1 getPriorVotes(address account, uint blockNumber) public view returns (uint96)
```

Determine the prior number of votes for an account as of a block number. Block number must be a finalized block or else this function will revert to prevent misinformation.

Internal Functions

_moveDelegates

```
1 _moveDelegates(address srcRep, address dstRep, uint96 amount) internal
```

Misnomer, from Compound Finance's `_moveDelegates`. Helps keep track of available voting power for FXS holders.

_writeCheckpoint

```
1 _writeCheckpoint(address voter, uint32 nCheckpoints, uint96 oldVotes, uint96 newVotes) int
```

From Compound Finance's governance scheme. Helps keep track of available voting power for FXS

holders at a specific block. Called when a FXS token transfer, mint, or burn occurs.

safe32

```
safe32(uint n, string memory errorMessage) internal pure returns (uint32)
```

Make sure the provided int is 32 bits or less, and convert it to a uint32.

safe96

```
safe96(uint n, string memory errorMessage) internal pure returns (uint96)
```

Make sure the provided int is 96 bits or less, and convert it to a uint96.

add96

```
add96(uint96 a, uint96 b, string memory errorMessage) internal pure returns (uint96)
```

Add two uint96 integers safely.

sub96

```
sub96(uint96 a, uint96 b, string memory errorMessage) internal pure returns (uint96)
```

Subtract two uint96 integers safely.

getChainId

```
getChainId() internal pure returns (uint)
```

Return the Ethereum chain ID the contract is deployed on

Events

VoterVotesChanged

```
VoterVotesChanged(address indexed voter, uint previousBalance, uint newBalance)
```

Emitted when a voters account's vote balance changes

FXSBurned

```
FXSBurned(address indexed from, address indexed to, uint256 amount)
```

Emitted when FXS is burned, usually from a redemption by the pool

---

Modifiers

onlyPools

```
onlyPools()
```

Restrict actions to pool contracts, e.g. minting new FXS.

onlyByOracle

```
onlyByOracle()
```

Restrict actions to the oracle, such as setting the FRAX and oracle addresses

Multisigs

Name	Address
Community	0x63278bF9AcdFC9fA65CFa2940b89A34ADfbCb4A1
Team	0x8D4392F55bC76A046E443eb3bab99887F4366BB0
Investors	0xa95f86fE0409030136D6b82491822B3D70F890b3
Treasury	0x9AA7Db8E488eE3ffCC9CdFD4f2EaECC8ABeDCB48
Advisors	0x874a873e4891fB760EdFDae0D26cA2c00922C404
Comptroller	0xB1748C79709f4Ba2Dd82834B8c82D4a505003f27

Frax Pools

Contract used for minting and redeeming FRAX, as well as buying back excess collateral.

Deployment

Frax Pool contracts are deployed and permissioned from the governance system, meaning that a new type of collateral may be added at any time after a governance proposal succeeds and is executed. The current pool is USDC, with further collateral types open for future pools.

USDC: `0x3C2982CA260e870eee70c423818010DfeF212659`

Description

A Frax Pool is the smart contract that mints FRAX tokens to users for placing collateral or returns collateral by redeeming FRAX sent into the contract. Each Frax Pool has a different type of accepted collateral. Frax Pools can be in any kind of cryptocurrency, but stablecoins are easiest to implement due to their small fluctuations in price. Frax is designed to accept any type of cryptocurrency as collateral, but low volatility pools are preferred at inception since they do not change the collateral ratio erratically. There are promising new projects, such as Reflex Bonds, which dampen the volatility of their underlying crypto assets. Reflex Bonds could make for ideal FRAX collateral in the future. New Frax Pools can be added through FXS governance votes.

Each pool contract has a pool ceiling (the maximum allowable collateral that can be stored to mint FRAX) and a price feed for the asset. The initial Frax Pool at genesis will be USDC (USD Coin) and USDT (Tether) due to their large market capitalization, stability, and availability on Ethereum.

The pools operate through permissioned calls to the FRAXStablecoin (FRAX) and FRAXShare (FXS) contracts to mint and redeem the protocol tokens.

Minting and Redeeming FRAX

The contract has 3 minting functions: mint1t1FRAX(), mintFractionalFRAX(), and mintAlgorithmicFRAX(). The contract also has 3 redemption functions that mirror the minting functions: redeem1t1FRAX(), redeemFractionalFRAX(), redeemAlgorithmicFRAX().

The functions are separated into 1 to 1, fractional, and algorithmic phases to optimize gas usage. The 1 to 1 minting and redemption functions are only available when the collateral ratio is 100%. The fractional minting and redemption functions are only available between a collateral ratio of 99.99% and 0.01%. The algorithmic minting and redemption functions are only available at a ratio of 0%.

Slippage

Each of the minting and redeeming functions also has an `AMOUNT_out_min` parameter that specifies the minimum units of tokens expected from the transaction. This acts as a limit for slippage tolerance when submitting transactions, as the prices may update from the time a transaction is created to the time it is included in a block.

State Variables

AccessControl (Inherited)

https://docs.openzeppelin.com/contracts/3.x/api/access#AccessControl

FraxPool-Specific

```
ERC20 private collateral_token
```

Instance for the collateral token in the pool.

```
address private collateral_address
```

Address of the collateral token.

```
address[] private owners
```

List of the pool owners.

```
address private oracle_address
```

Address of the oracle contract.

```
address private frax_contract_address
```

Address of the FRAX contract.

```
address private fxs_contract_address
```

Address of the FXS contract.

```
address private timelock_address
```

Address of the timelock contract.

```
FRAXShares private FXS
```

Instance of the FXS contract.

```
FRAXStablecoin private FRAX
```

Instance of the FRAX contract.

```
UniswapPairOracle private oracle
```

Instance of the oracle contract.

```
mapping (address => uint256) private redeemFXSBalances
```

Keeps track of redemption balances for a given address. A redeemer cannot both request redemption and actually redeem their FRAX in the same block. This is to prevent flash loan exploits that could crash FRAX and/or FXS prices. They have to wait until the next block. This particular variable is for the FXS portion of the redemption.

```
mapping (address => uint256) private redeemCollateralBalances
```

Keeps track of redemption balances for a given address. A redeemer cannot both request redemption and actually redeem their FRAX in the same block. This is to prevent flash loan exploits that could crash FRAX and/or FXS prices. They have to wait until the next block. This particular variable is for the collateral portion of the redemption.

```
uint256 public unclaimedPoolCollateral
```

Sum of the `redeemCollateralBalances`.

```
uint256 public unclaimedPoolFXS
```

Sum of the `redeemFXSBalances`.

```
mapping (address => uint256) lastRedeemed
```

Keeps track of the last block a given address redeemed.

```
uint256 private pool_ceiling
```

Maximum amount of collateral the pool can take.

```
bytes32 private constant MINT_PAUSER
```

`AccessControl` role for the mint pauser.

```
bytes32 private constant REDEEM_PAUSER
```

`AccessControl` role for the redeem pauser.

```
1 bytes32 private constant BUYBACK_PAUSER
```

`AccessControl` role for the buyback pauser.

```
1 bool mintPaused = false
```

Whether or not minting is paused.

```
1 bool redeemPaused = false
```

Whether or not redeem is paused.

```
1 bool buyBackPaused = false
```

Whether or not buyback is paused.

View Functions

unclaimedFXS

```
1 unclaimedFXS(address _account) public view returns (uint256)
```

Return the total amount of unclaimed FXS.

unclaimedCollateral

```
1 unclaimedCollateral(address _account) public view returns (uint256)
```

Return the total amount of unclaimed collateral.

collatDollarBalance

```
1 collatDollarBalance() public view returns (uint256)
```

Return the pool's total balance of the collateral token, in USD.

availableExcessCollatDV

```
1 availableExcessCollatDV() public view returns (uint256)
```

Return the pool's excess balance of the collateral token (over that required by the collateral ratio), in USD.

getCollateralPrice

```
1 getCollateralPrice() public view returns (uint256)
```

Return the price of the pool's collateral in USD.

Public Functions

mint1t1FRAX

```
1 mint1t1FRAX(uint256 collateral_amount_d18) external notMintPaused
```

Mint FRAX from collateral. Valid only when the collateral ratio is 1.

mintFractionalFRAX

```
1 mintFractionalFRAX(uint256 collateral_amount, uint256 fxs_amount) external notMintPaused
```

Mint FRAX from collateral and FXS. Valid only when the collateral ratio is between 0 and 1.

mintAlgorithmicFRAX

```
1 mintAlgorithmicFRAX(uint256 fxs_amount_d18) external notMintPaused
```

Mint FRAX from FXS. Valid only when the collateral ratio is 0.

redeem1t1FRAX

```
1 redeem1t1FRAX(uint256 FRAX_amount) external notRedeemPaused
```

Redeem collateral from FRAX. Valid only when the collateral ratio is 1. Must call `collectionRedemption()` later to collect.

redeemFractionalFRAX

```
1 redeemFractionalFRAX(uint256 FRAX_amount) external notRedeemPaused
```

Redeem collateral and FXS from FRAX. Valid only when the collateral ratio is between 0 and 1. Must call `collectionRedemption()` later to collect.

redeemAlgorithmicFRAX

```
redeemAlgorithmicFRAX(uint256 FRAX_amount) external notRedeemPaused
```

Redeem FXS from FRAX. Valid only when the collateral ratio is 0. Must call `collectionRedemption()` later to collect.

collectRedemption

```
collectRedemption() public
```

After a redemption happens, transfer the newly minted FXS and owed collateral from this pool contract to the user. Redemption is split into two functions to prevent flash loans from being able to take out FRAX / collateral from the system, use an AMM to trade the new price, and then mint back into the system.

buyBackFXS

```
buyBackFXS(uint256 FXS_amount) external
```

Function can be called by an FXS holder to have the protocol buy back FXS with excess collateral value from a desired collateral pool. This can also happen if the collateral ratio > 1

recollateralizeAmount

```
recollateralizeAmount() public view returns (uint256 recollateralization_left)
```

When the protocol is recollateralizing, we need to give a discount of FXS to hit the new CR target. Returns value of collateral that must increase to reach recollateralization target (if 0 means no recollateralization)

recollateralizeFrax

```
recollateralizeFrax(uint256 collateral_amount_d18) public
```

Thus, if the target collateral ratio is higher than the actual value of collateral, minters get FXS for adding collateral. This function simply rewards anyone that sends collateral to a pool with the same amount of FXS + .75%. Anyone can call this function to recollateralize the protocol and take the hardcoded .75% arb opportunity

---

Restricted Functions

toggleMinting

```
toggleMinting() external onlyMintPauser
```

Toggle the ability to mint.

toggleRedeeming

```
1 toggleRedeeming() external onlyRedeemPauser
```

Toggle the ability to redeem.

toggleBuyBack

```
1 toggleBuyBack() external onlyBuyBackPauser
```

Toggle the ability to buyback.

setPoolCeiling

```
1 setPoolCeiling(uint256 new_ceiling) external onlyByOwnerOrGovernance
```

Set the `pool_ceiling`, which is the total units of collateral that the pool contract can hold.

setOracle

```
1 setOracle(address new_oracle) external onlyByOwnerOrGovernance
```

Set the `oracle_address`.

setCollateralAdd

```
1 setCollateralAdd(address _collateral_address) external onlyByOwnerOrGovernance
```

Set the `collateral_address`.

addOwner

```
1 addOwner(address owner_address) external onlyByOwnerOrGovernance
```

Add an address to the array of owners.

removeOwner

```
1 removeOwner(address owner_address) external onlyByOwnerOrGovernance
```

Remove an owner from the owners array.

Modifiers

onlyByOwnerOrGovernance

```
onlyByOwnerOrGovernance()
```

Restrict actions to the governance contract or the owner(s).

notRedeemPaused

```
notRedeemPaused()
```

Ensure redemption is not paused.

notMintPaused

```
notMintPaused()
```

Ensure minting is not paused.

Governance

Used for governance actions on the FRAX, FXS, staking, and pool contracts.

Deployment

The AlphaGovernor contract is deployed at `0xd74034C6109A23B6c7657144cAcBbBB82BDCB00E`

The Timelock contract is deployed at `0x8412ebf45bAC1B340BbE8F318b928C466c4E39CA`

Description

The Frax governance module is forked from Compound, with FXS acting as the voting token in the system.

Users may propose new changes to the protocol if they are holding a certain threshold of FXS (1% of the total votes, equivalent to `1,000,000 FXS`), or may combine their votes together to submit a proposal.

Once a proposal has been submitted, it begins an active voting period of `3 days`, where it may be voted for or against. If a majority is in support of the proposal at the end of the period, and the proposal has a minimum of `4,000,000 FXS` in favor, the change is queued into the Timelock where it can be implemented after `2 days`.

Access Control

Frax uses OpenZepplin's Access Control module for certain permissions. Each role in Frax that is permission through `hasRole` may also be decentralized by a proposal in the governance module, which can then call `grantRole` on other addresses.

Timelock (48 hours)

The Timelock contract is responsible for executing proposals that have succeeded in their voting phase, and has permissions over the other system contracts like `FRAXStablecoin`, `FRAXShares`, and `FRAXPool` through Access Control. The Timelock contract provides transparency to changes that will affect the Frax protocol, as the queued proposals may only be activated after the waiting period is satisfied within the contract logic.

Contracts

The Frax governance contract controls the Timelock contract, which receives accepted proposals and uses its authority over the system contracts to update them as needed. The Frax admin address is the initial governor of the system to which roles are granted to upon deployment. Over time, the roles will be granted to the Timelock contract which gives governance control to the community.

Oracle

Used to get FRAX-collateral and FXS-collateral price data.

Deployment

The Chainlink price consumer contract is deployed at:
`0xBa6C6EaC41a24F9D39032513f66D738B3559f15a`

The FRAX-ETH Uniswap pair oracle is deployed at:
`0xD18660Ab8d4eF5bE062652133fe4348e0cB996DA`

The FRAX-USDC Uniswap pair oracle is deployed at:
`0x2AD064cEBA948A2B062ba9AfF91c98B9F0a1f608`

The FRAX-USDT Uniswap pair oracle is deployed at:
`0x97587c990617f65A83CAb4f08b23F78472a0413b`

The FRAX-FXS Uniswap pair oracle is deployed at:
`0xD0435BF68dF2B516C6382caE8847Ab5cdC5c3Ea7`

The FXS-ETH Uniswap pair oracle is deployed at:

`0x9e483C76D7a66F7E1feeBEAb54c349Df2F00eBdE`

The FXS-USDC Uniswap pair oracle is deployed at:
`0x28fdA30a6Cf71d5fC7Ce17D6d20c788D98Ff2c46`

The FXS-USDT Uniswap pair oracle is deployed at:
`0x4FCb1759BD13950E7e73eEd650eb5bB355bC1CBC`

Description

The Frax system takes price feeds from two external systems: Chainlink and Uniswap. The system records the ETH-USD price from Chainlink and applies it to the FRAX-wETH and FXS-wETH pool balances from Uniswap in order to get an accurate FRAX-USD and FXS-USD price. This allows FRAX to follow the true price of USD and not a basket of onchain stablecoins (which could deviate significantly).

The Chainlink oracle is a time weighted average of the ETH-USD price updated every hour.

Chainlink

The `ChainlinkETHUSDPriceConsumer` contract is responsible for getting the price of ETH in terms of USD. To get the price of ETH in USD from this contract, call `getLatestPrice()` and divide by `getDecimals()`.

Uniswap

The Uniswap V2 system includes price oracles that use a time-weighted average price in order to robustly calculate an accurate price for tokens within the Uniswap pools. Frax uses these oracles over a `1 hour` time-weighted average price on its Uniswap pools to get price information for FRAX, FXS, and the collateral tokens in the system. The period of the time-weighted average price is changeable as a system parameter through a governance proposal.

The `UniswapPairOracle` contract allows one to get the price of a token within the system from its pool balance. To get the price of a token from a pair, call `consult(address token, uint amountIn)` on the pair's UniswapPairOracle instance with the token's address and requested quantity.

Staking

Allows staking Uniswap trading pair liquidity pool tokens in exchange for FXS rewards.

Based on Synthetix's staking contract:

https://docs.synthetix.io/incentives/

Description

Frax users are able to stake in select Uniswap liquidity pool tokens in exchange for FXS rewards. Future pools and incentives can be added by governance.

Deployment

Liquidity Pool Tokens (LP)

Uniswap FRAX/WETH LP: `0xFD0A40Bc83C5faE4203DEc7e5929B446b07d1C76`

Uniswap FRAX/USDC LP: `0x97C4adc5d28A86f9470C70DD91Dc6CC2f20d2d4D`

Uniswap FRAX/FXS LP: `0xE1573B9D29e2183B1AF0e743Dc2754979A40D237`

Uniswap FXS/WETH LP: `0xecBa967D84fCF0405F6b32Bc45F4d36BfDBB2E81`

Staking Contracts

Uniswap FRAX/WETH staking: `0xD875628B942f8970De3CcEaf6417005F68540d4f`

Uniswap FRAX/USDC staking: `0xa29367a3f057F3191b62bd4055845a33411892b6`

Uniswap FRAX/FXS staking: `0xda2c338350a0E59Ce71CDCED9679A3A590Dd9BEC`

Uniswap FXS/WETH staking (deprecated): `0xDc65f3514725206Dd83A8843AAE2aC3D99771C88`

State Variables

```
1 FRAXStablecoin private FRAX
```

Instance of the FRAX contract.

```
1 ERC20 public rewardsToken
```

Instance for the reward token.

```
1 ERC20 public stakingToken
```

Instance for the staking token.

```
uint256 public periodFinish
```

Block when the staking period will finish.

```
uint256 public rewardRate
```

Maximum reward per second.

```
uint256 public rewardsDuration
```

Reward period, in seconds.

```
uint256 public lastUpdateTime
```

Last timestamp where the contract was updated / state change.

```
uint256 public rewardPerTokenStored
```

Actual reward per token in the current period.

```
uint256 public locked_stake_max_multiplier
```

Maximum boost / weight multiplier for locked stakes.

```
uint256 public locked_stake_time_for_max_multiplier
```

The time, in seconds, to reach `locked_stake_max_multiplier` .

```
uint256 public locked_stake_min_time
```

Minimum staking time for a locked staked, in seconds.

```
string public locked_stake_min_time_str
```

String version is `locked_stake_min_time_str` .

```
uint256 public cr_boost_max_multiplier
```

Maximum boost / weight multiplier from the collateral ratio (CR). This is applied to both locked and unlocked stakes.

```
mapping(address => uint256) public userRewardPerTokenPaid
```

Keeps track of when an address last collected a reward. If they collect it some time later, they will get the correct amount because `rewardPerTokenStored` is constantly varying.

```
mapping(address => uint256) public rewards
```

Current rewards balance for a given address.

```
uint256 private _staking_token_supply
```

Total amount of pool tokens staked .

```
uint256 private _staking_token_boosted_supply
```

`_staking_token_supply` with the time and CR boosts accounted for. This is not an actual amount of pool tokens, but rather a 'weighed denominator'.

```
mapping(address => uint256) private _balances
```

Balance of pool tokens staked for a given address.

```
mapping(address => uint256) private _boosted_balances
```

`_balances` , but with the time and CR boosts accounted for, like `_staking_token_boosted_supply` .

```
mapping(address => LockedStake[]) private lockedStakes
```

Gives a list of locked stake lots for a given address.

```
struct LockedStake {
        bytes32 kek_id;
        uint256 start_timestamp;
        uint256 amount;
        uint256 ending_timestamp;
        uint256 multiplier; // 6 decimals of precision. 1x = 1000000
    }
```

A locked stake 'lot'.

View Functions

totalSupply

```
totalSupply() external override view returns (uint256)
```

Get the total number of staked liquidity pool tokens.

stakingMultiplier

```
stakingMultiplier(uint256 secs) public view returns (uint256)
```

Get the time-based staking multiplier, given the `secs` length of the stake.

crBoostMultiplier

```
crBoostMultiplier() public view returns (uint256)
```

Get the collateral ratio (CR) - based staking multiplier.

stakingTokenSupply

```
stakingTokenSupply() external view returns (uint256)
```

same as totalSupply().

balanceOf

```
balanceOf(address account) external override view returns (uint256)
```

Get the amount of staked liquidity pool tokens for a given `account`.

boostedBalanceOf

```
boostedBalanceOf(address account) external view returns (uint256)
```

Get the boosted amount of staked liquidity pool tokens for a given `account`. Boosted accounts for the CR and time-based multipliers.

lockedBalanceOf

```
lockedBalanceOf(address account) public view returns (uint256)
```

Get the amount of locked staked liquidity pool tokens for a given `account`.

unlockedBalanceOf

```
unlockedBalanceOf(address account) external view returns (uint256)
```

Get the amount of unlocked / free staked liquidity pool tokens for a given `account` .

lockedStakesOf

```
1 lockedStakesOf(address account) external view returns (LockedStake[] memory)
```

Return an array of all the locked stake 'lots' for

stakingDecimals

```
1 stakingDecimals() external view returns (uint256)
```

Returns the `decimals()` for `stakingToken` .

rewardsFor

```
1 rewardsFor(address account) external view returns (uint256)
```

Get the amount of FXS rewards for a given `account` .

lastTimeRewardApplicable

```
1 lastTimeRewardApplicable() public override view returns (uint256)
```

Used internally to keep track of `rewardPerTokenStored` .

rewardPerToken

```
1 rewardPerToken() public override view returns (uint256)
```

The current amount of FXS rewards for staking a liquidity pool token.

earned

```
1 earned(address account) public override view returns (uint256)
```

Returns the amount of unclaimed FXS rewards for a given `account` .

getRewardForDuration

```
1 getRewardForDuration() external override view returns (uint256)
```

Calculates the FXS reward for a given `rewardsDuration` period.

Mutative Functions

stake

```
stake(uint256 amount) external override nonReentrant notPaused updateReward(msg.sender)
```

Stakes some Uniswap liquidity pool tokens. These tokens are freely withdrawable and are only boosted by the `crBoostMultiplier()`.

stakeLocked

```
stakeLocked(uint256 amount, uint256 secs) external nonReentrant notPaused updateReward(msg
```

Stakes some Uniswap liquidity pool tokens and also locks them for the specified `secs` . In return for having their tokens locked, the staker's base `amount` will be multiplied by a linear time-based multiplier, which ranges from 1 at `secs` = 0 to `locked_stake_max_multiplier` at `locked_stake_time_for_max_multiplier`. The staked value is also multiplied by the `crBoostMultiplier()` . This multiplied value is added to `_boosted_balances` and acts as a weighted amount when calculating the staker's share of a given period reward.

withdraw

```
withdraw(uint256 amount) public override nonReentrant updateReward(msg.sender)
```

Withdraw unlocked Uniswap liquidity pool tokens.

withdrawLocked

```
withdrawLocked(bytes32 kek_id) public nonReentrant updateReward(msg.sender)
```

Withdraw locked Uniswap liquidity pool tokens. Will fail if the staking time for the specific `kek_id` staking lot has not elapsed yet.

getReward

```
getReward() public override nonReentrant updateReward(msg.sender)
```

Claim FXS rewards.

exit

```
exit() external override
```

Withdraw all unlocked pool tokens and also collect rewards.

renewIfApplicable

```
renewIfApplicable() external
```

Renew a reward period if the period's finish time has completed. Calls `retroCatchUp()` .

retroCatchUp

```
retroCatchUp() internal
```

Renews the period and updates `periodFinish` , `rewardPerTokenStored` , and `lastUpdateTime` .

Restricted Functions

notifyRewardAmount

```
notifyRewardAmount(uint256 reward) external override onlyRewardsDistribution updateReward(
```

This notifies people (via the event `RewardAdded`) that the reward is being changed.

recoverERC20

```
recoverERC20(address tokenAddress, uint256 tokenAmount) external onlyOwner
```

Added to support recovering LP Rewards from other systems to be distributed to holders.

setRewardsDuration

```
setRewardsDuration(uint256 _rewardsDuration) external onlyOwner
```

Set the duration of the rewards period.

setLockedStakeMaxMultiplierUpdated

```
setLockedStakeMaxMultiplierUpdated(uint256 _locked_stake_max_multiplier) external onlyOwne
```

Set the maximum multiplier for locked stakes.

setLockedStakeTimeForMaxMultiplier

```
setLockedStakeTimeForMaxMultiplier(uint256 _locked_stake_time_for_max_multiplier) external
```

Set the time, in seconds, when the locked stake multiplier reaches `locked_stake_max_multiplier` .

setLockedStakeMinTime

```
setLockedStakeMinTime(uint256 _locked_stake_min_time) external onlyOwner
```

Set the minimum time, in seconds, of a locked stake.

setMaxCRBoostMultiplier

```
setMaxCRBoostMultiplier(uint256 _max_boost_multiplier) external onlyOwner
```

aaa

initializeDefault

```
initializeDefault() external onlyOwner
```

Intended to only be called once in the lifetime of the contract. Initializes `lastUpdateTime` and `periodFinish` .

Modifiers

updateReward

```
updateReward(address account)
```

Calls `retroCatchUp()` , if applicable, and otherwise syncs `rewardPerTokenStored` and `lastUpdateTime` . Also, syncs the `rewards` and `userRewardPerTokenPaid` for the provided `account` .

Curve AMO

A description of the Curve AMO smart contract

`iterate()` The iterate function calculates the balances of FRAX and 3CRV in the metapool in the hypothetical worst-case assumption of FRAX price falling to the CR. To start, the function takes the current live balances of the metapool and simulates an external arbitrageur swapping 10% of the current FRAX in the metapool until the price given is equal (or close to) the CR, swapping out the corresponding amount of 3CRV. This simulates the situation wherein the open-market price of FRAX falls to the CR, and the resulting 1-to-1 swap normally offered by the metapool is picked off by arbitrageurs until there is no more profit to be had by buying FRAX elsewhere for the CR price and selling it into the metapool. Line 282 is the specific location where the price of FRAX is checked.

Then, the metapool checks how much its LP tokens would withdraw in that worst-case scenarios in terms of the underlying FRAX and 3CRV. The ratio between the two is normally tilted roughly 10-to-1 in terms of FRAX withdrawable to 3CRV withdrawable. For the protocol's accounting of how much collateral it has, it values each 3CRV withdrawable at the underlying collateral value (i.e. how much USDC it can redeem for it) and each FRAX at the collateral ratio. Since the protocol never actually sends this much FRAX into circulation under normal circumstances, this is a highly conservative estimate on the amount of collateral it is actually entitled to in terms of USDC.

To check scenarios of how much reserves would be indebted to the Curve AMO at other prices, one may simply adjust the `fraxFloor()` value in local testing through setting a `custom_floor`.

AMOs

Investor AMO: `0xEE5825d5185a1D512706f9068E69146A54B6e076`

Curve AMO: `0xbd061885260F176e05699fED9C5a4604fc7F2BDC`

Lending AMO: `0x9507189f5B6D820cd93d970d67893006968825ef`

AMO Minter

Frax's updated structure for minting FRAX and processing mints & redeems

In September and October of 2021, the system moved to an updated model using a FraxPoolV3 system contract (`0x2fE065e6FFEf9ac95ab39E5042744d695F560729`) that handles responsive mints & redeems for the protocol with a lower attack surface. Through this new pool, the Frax AMO Minter contract was designed to do algorithmic minting according to the specifications of new AMOs attached to the FraxPoolV3.

A consequence of this was that the old AMOs that were built on the FraxPoolV2 (`0x1864Ca3d47AaB98Ee78D11fc9DCC5E7bADdA1c0d`) were upgraded to new versions using the FraxPoolV3 collateral and minting system, all controlled through the comptroller msig & timelock system.

The new system allows for automated collection of yields & return of collateral to the FraxPoolV3, of which the profit may be distributed to FXS holders by the FXS1559 AMO.

Contract Addresses

The AMO Minter is deployed at the following address on the Ethereum mainnet: `0xcf37B62109b537fa0Cb9A90Af4CA72f6fb85E241`.

Miscellaneous & Bug Bounty

All addresses: https://github.com/FraxFinance/frax-solidity/blob/master/src/types/constants.ts#L626

Oracle updater bot: `0xBB437059584e30598b3AF0154472E47E6e2a45B9`

Utility / helper contract deployer: `0x36a87d1e3200225f881488e4aeedf25303febcae`

Front Running Mitigation & Testing Environments

Frax Protocol testing suite uses Hardhat+Truffle (with Ganache support) on all testing scripts.

Front running of smart contracts are mitigated on system contracts since swaps have ceiling sizes. Thus, Frax Protocol front running is dependent on protocols that AMOs mint into rather than endogenous system contracts.

Frax Bug Bounty

Frax Finance provides one of the largest bounties in the industry for exploits where user funds are at risk or protocol controlled funds/collateral are at risk.

The bounty is simply calculated as the lower value of 10% of the total possible exploit or $10m worth paid in FRAX+FXS (evenly split). Both tokens are immediately liquid. The bounty will be delivered immediately or a maximum turnaround time of 5 days due to timelock+mitigation. Slow arbitrage opportunities or value exchange over a prolonged period is not applicable to this bounty and will receive a base compensation bounty of 50,000 FRAX.

Note: This bounty does not cover any front-end bugs/visual bugs or any type of server-side code of any web application that interacts with the Frax Protocol. The above bug bounty is **only** for smart contract code. Smart contract code on any chain that manages Frax Protocol value and/or user deposited value is included in this bounty.

Contacts: you can reach out anonymously through any communication channel including Twitter, Telegram, Discord, or Signal.

API

Combined Data

https://api.frax.finance/combineddata/

Staking / APRs

https://api.frax.finance/pools

Other

Media Kit / Logos

FRAX

FRAX_Logos.zip 3MB
Binary

FXS

FXS_Logos.zip 1MB
Binary