English

Frax: Fractional Stablecoin Protocol

A new category of decentralized stablecoin with a novel mechanism

Frax is the first fractional stablecoin protocol. Frax is open-source, permissionless, and entirely on-chain – currently implemented on Ethereum and 12 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.

The Frax ecosystem has 2 stablecoins: FRAX (pegged to the US dollar) & FPI (pegged to the US Consumer Price Index). The Frax Finance economy is composed primarily of the two stablecoins, a native AMM (Fraxswap), and a lending facility (Fraxlend).

Core concepts to understand the unified Frax Finance ecosystem include:

- **Fractional** – Frax is the first and only stablecoin with parts of its supply backed by collateral and parts of the supply algorithmically stabilized. The stablecoin (FRAX) is named after this hybrid fractional-reserve system.
- **Fraxswap, a native AMM** – Fraxswap is the first AMM with time weighted average market maker orders used by the Frax Protocol for rebalancing collateral, mints/redemptions, expanding/contracting FRAX supply, and deploying protocol owned liquidity onchain.
- **Fraxlend, permissionless lending markets** – Fraxlend is the lending facility for the FRAX & FPI stablecoins allowing debt origination, customized non-custodial loans, and onboarding collateral assets to the Frax Finance economy.
- **Crypto Native CPI Stablecoin** – Frax's end vision is to build the most important decentralized stablecoins in the world. The Frax Price Index (FPI) stablecoin is the first stablecoin pegged to a basket of consumer goods creating its own unit of account separate from any nation state denominated money.
- **Four Tokens** – FRAX is the stablecoin targeting a tight band around $1/coin. Frax Share (FXS) is the governance token of the entire Frax ecosystem of smart contracts which accrues fees, seigniorage revenue, and excess collateral value. FPI is the inflation resistant, CPI pegged stablecoin. FPIS is the governance token of the Frax Price Index and splits its value capture with FXS holders.
- **Gauge Rewards System** – The community can propose new gauge rewards for strategies that integrate FRAX stablecoins. FXS emissions are fixed, halve each year, and entirely flow to different gauges based on the votes of veFXS stakers.

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.

Frax Stablecoin (FRAX)

Achieves stability with less fiat collateral needed (capital efficiency)

No custodial risk / fully onchain

Monetary policy decided by community (FXS Holders)

Optional anonymity of minted FRAX via zk-SNARKs (planned)

$1

Stabilized via multiple mechanisms (bonds, minting, redeeming, Curve metapool, AMOs)

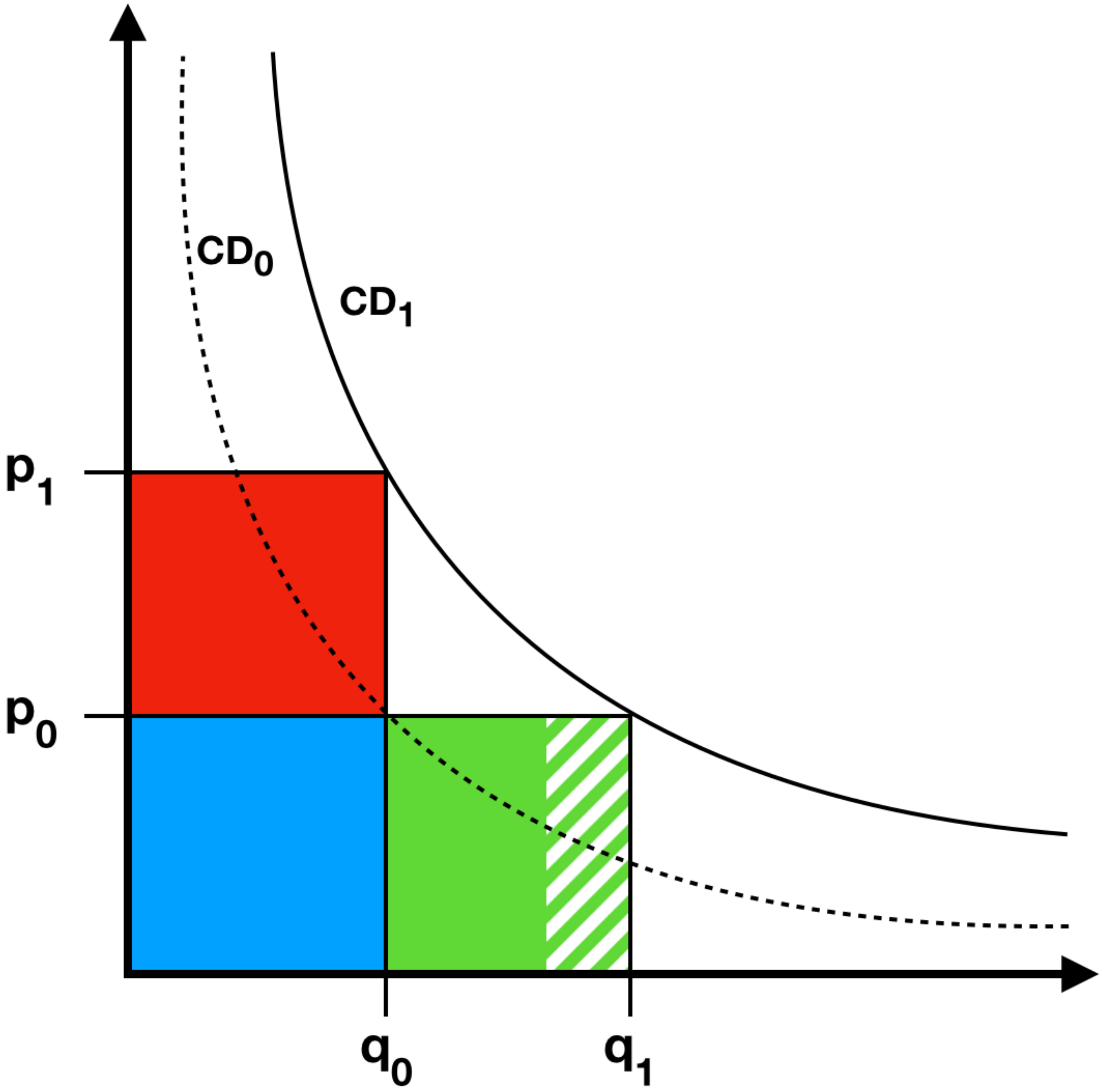
Price Stability

How arbitrage keeps FRAX price-stable

FRAX can be minted and redeemed from the system for $1 of value, allowing arbitragers to balance the demand and supply of FRAX in the open market. At all times in order to mint new FRAX a user must place $1 worth of value into the system. If the market price is above the price target of $1, then there is an arbitrage opportunity to mint tokens by placing $1 of value into the system per FRAX and sell the minted FRAX for above $1 in the open market. The difference is simply the proportion of FXS and collateral comprising the $1 of value. When FRAX is in the 100% collateral phase, all of the value that is used to mint FRAX is collateral. As the protocol moves into the fractional state, some of the value that enters into the system during minting becomes FXS (which is then burned). For example, in a 96% collateral ratio, every FRAX minted requires $.96 of collateral and burning $.04 of FXS. In a 95% collateral ratio, every FRAX minted requires $.95 of collateral and burning $.05 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 easy to understand and economically sound. During the 100% phase, it is trivially simple. During the fractional/algorithmic phase, FXS is burned as FRAX is minted. On the other hand, when FRAX is redeemed, minting of FXS occurs. When there is demand for FRAX, redeeming it for FXS plus collateral initiates minting of a similar amount of FRAX into circulation on the other end (which burns a similar amount of FXS). The value that accrues to the FXS market cap is the sum of the non-collateralized value of FRAX's market cap. FXS token's value is therefore partially determined by the demand for FRAX. This is the summation of all past and future shaded areas under the curve displayed as follows.



The supply/demand curve illustrates how minting and redeeming FRAX stabilizes the price (q is quantity, p is price). At CD_0 the FRAX's price 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. For instance, 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 a demand increase, market cap increases either by an increase in quantity (at a stable price) or through an increase in price. The green square and red square have the same area and thus would have added the same amount of value in market cap. On a side note, the half-shaded portion in the green square indicates 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 FRAX demand.

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.

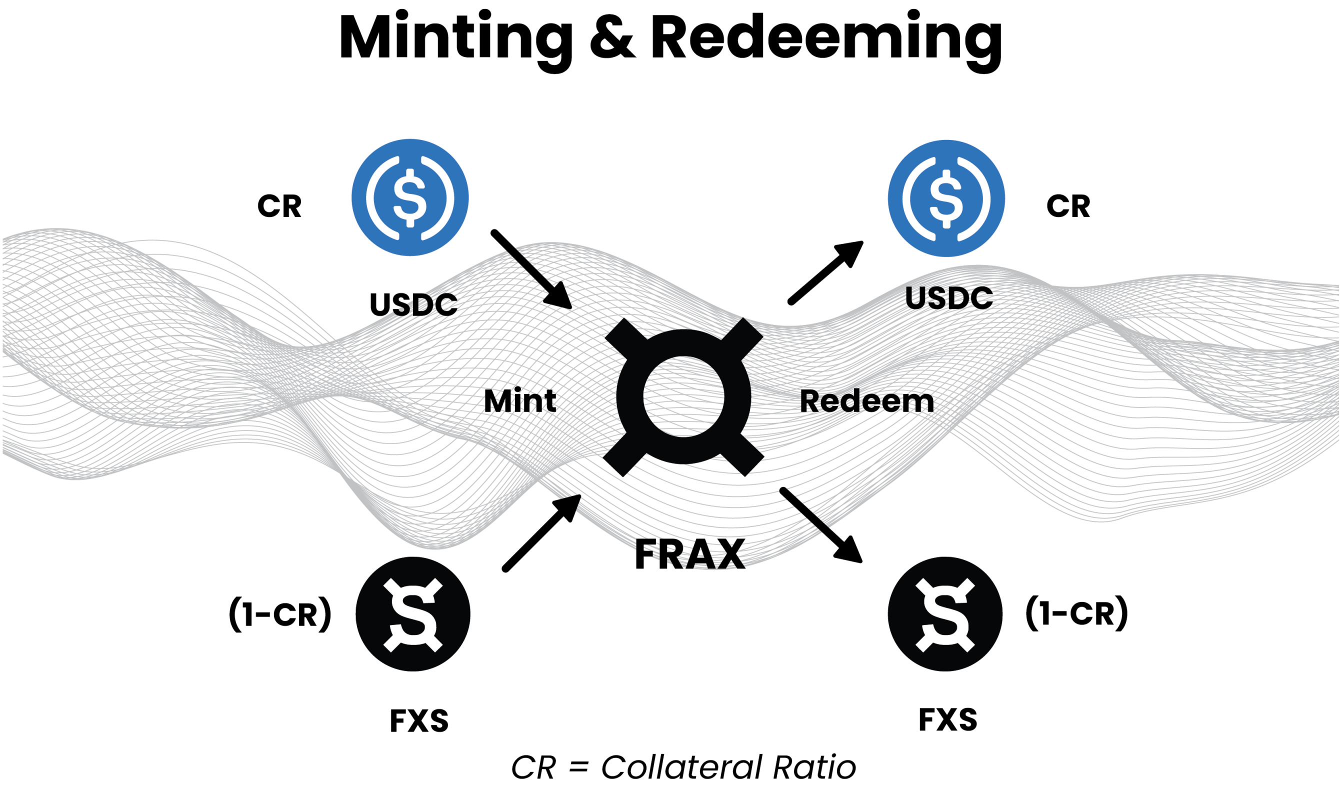
At genesis, the protocol adjusts the collateral ratio once every hour by a step of .25%. When FRAX is at or above $1, the function lowers the collateral ratio by one step per hour and when the price of FRAX is below $1, the function increases the collateral ratio by one step per hour. This means that 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.

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.

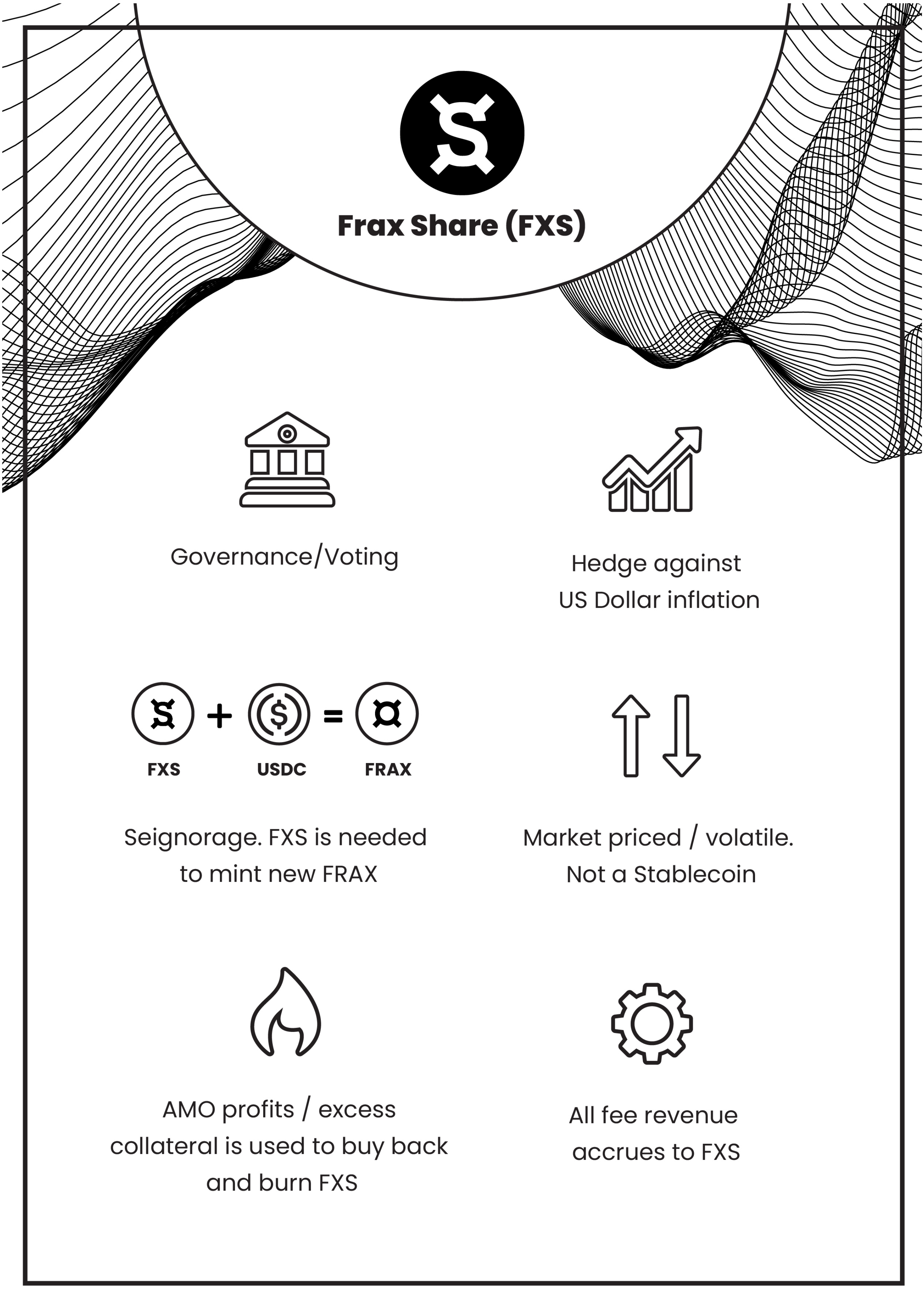
The Frax Share token (FXS) is the non-stable, utility token in the protocol. It is meant to be volatile and hold rights to governance and all utility of the system. It is important to note that we take a highly governance-minimized approach to designing trustless money in the same ethos as Bitcoin. We 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.



Frax Share (FXS)

Governance/Voting

Hedge against
US Dollar inflation

FXS + USDC = FRAX

Seignorage. FXS is needed
to mint new FRAX

Market priced / volatile.
Not a Stablecoin

AMO profits / excess
collateral is used to buy back
and burn FXS

All fee revenue
accrues to FXS

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

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

NOTE: Governance has voted to move 100% of protocol profits to veFXS yield after this vote. This documentation page below describes the original FXS1559 specification which burnt all FXS bought back. The current mechanism sends that amount of FXS to the veFXS yield contract.

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.

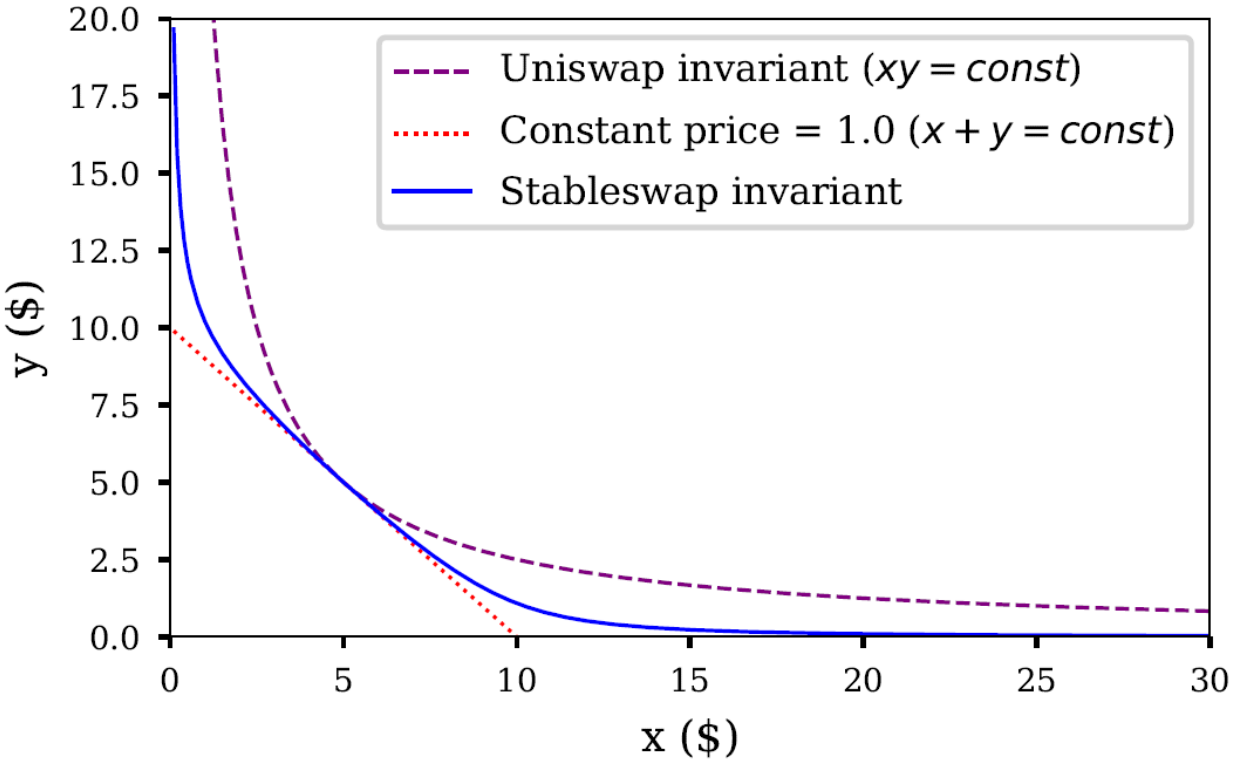
Curve

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 XD^{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.

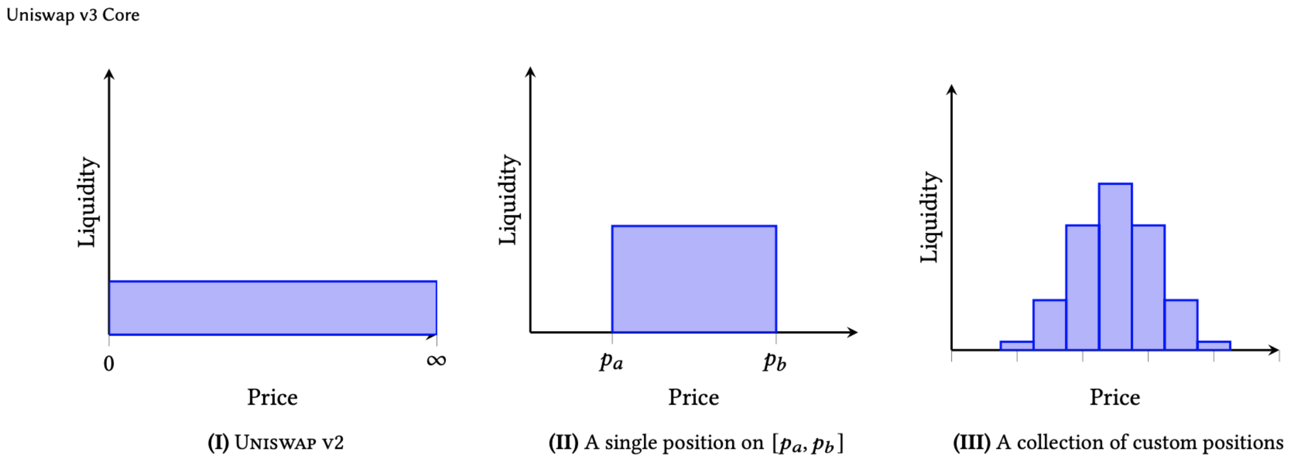
Additionally, Curve allocates CRV tokens as rewards for liquidity providers to select pools (called gauge rewards). Since the Frax protocol will likely be the largest liquidity provider of the FRAX3CRV pool, it can allocate all its FRAX3CRV LP tokens into Curve gauges to earn a significant return. The 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

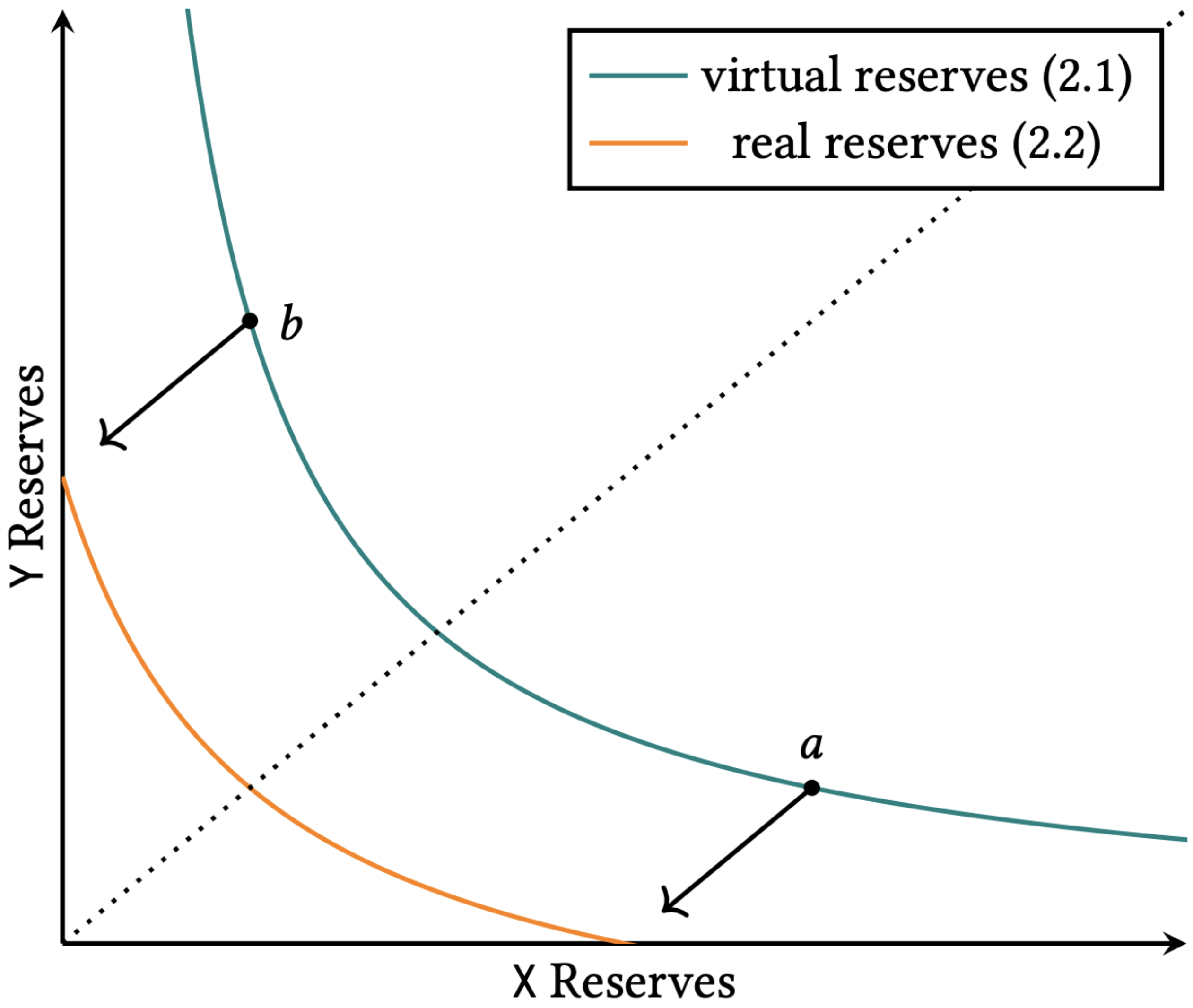
Taken from the Uniswap v3 whitepaper

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`

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.

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.

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 has 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.

Gauge 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. Other protocols can choose to distribute their rewards through Frax's gauge farming contracts to acquire the veFXS boost functionality. FXS gauge farming contracts support up to 4 different token rewards per gauge.

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 based on partnership agreements and governance votes. Each gauge will display the exact terms of the boosts available.

Farming boosts are given in ratios of veFXS per 1 FRAX in the LP deposit token. For example, a FRAX-IQ gauge 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 that contains 5,000 FRAX (total value of $10,000).

Most gauges currently offer a 2x boost in yield with a requirements of 4 veFXS to 1 FRAX.

veFXS Yield

The primary cash flow distribution mechanism of the Frax Protocol is to veFXS holders. Cash flow earned from AMOs, Fraxlend loans, and Fraxswap fees are typically used to buy back FXS from the market then distributed to veFXS stakers as yield. The emission rate varies depending on protocol profitability, sources of cash flow, market price of FXS, and governance actions.

Historical view of veFXS yield can be viewed here: https://app.frax.finance/vefxs

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 help create long term alignment for the Frax Finance economy.

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

- Allocate voting power to long-term holders of FXS through veFXS
- Incentivizing gauge farmers to stake FXS
- Allowing DAOs and other projects to build a large and long term veFXS position and participate in Frax governance.
- 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

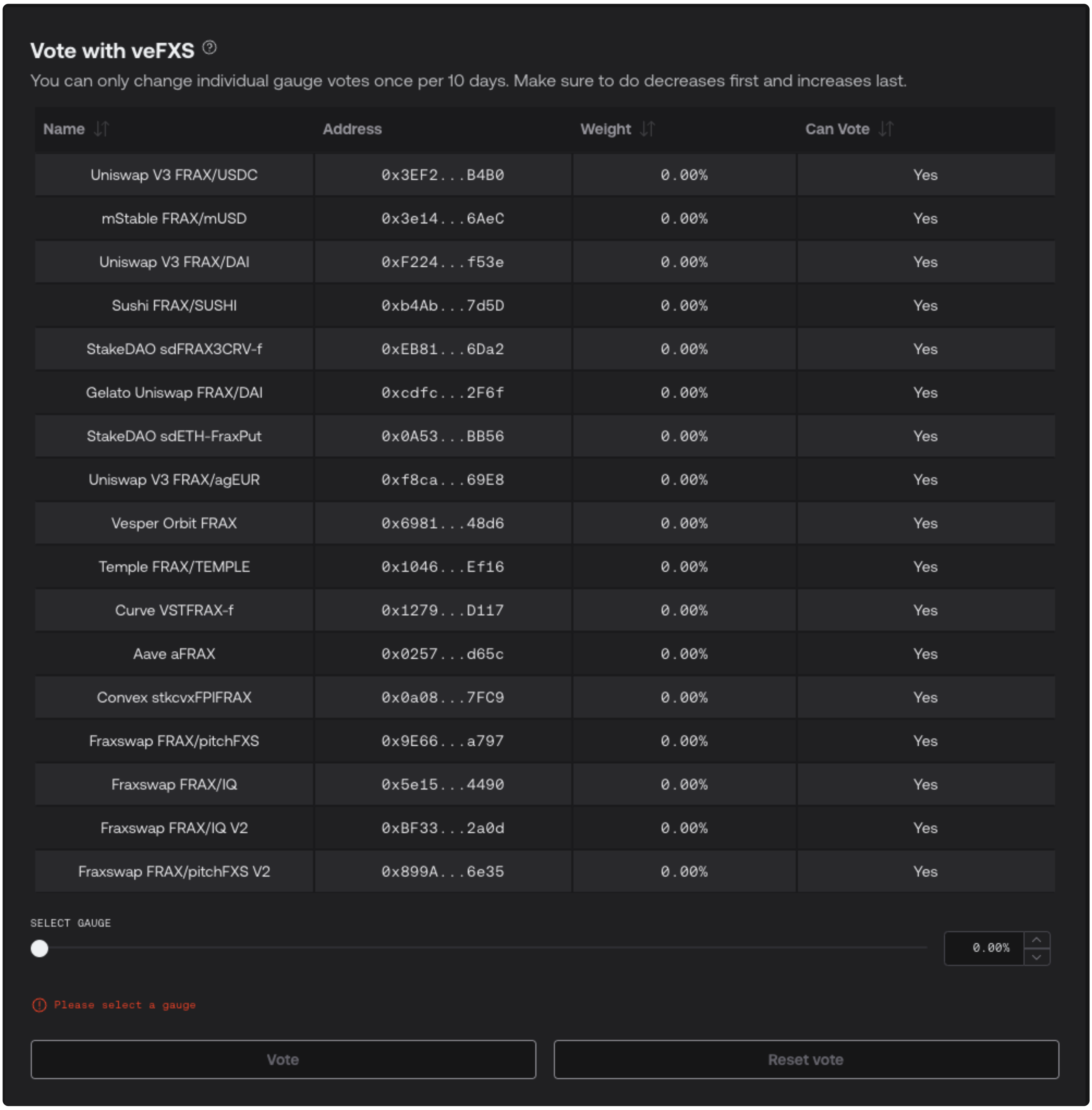
Gauge weighted system for controlling FXS emissions

A "gauge" is a farming smart contract that takes deposits in one asset (typically an LP token, a vault token, NFT position etc) and rewards the depositor yield in FXS (and potentially other) tokens. Gauge contracts can take many forms of deposits such as FRAX lending deposits (aFRAX in Aave, cFRAX in Compound, fFRAX in Fraxlend), LP tokens (Curve LP tokens or Fraxswap LP tokens), or even NFTs (such as Uniswap v3 NFT positions). Gauges are used to incentivize particular "strategies" and behaviors that are advantageous to the protocol such as increasing FRAX lending, deepening liquidity of certain pairs, or growing a partnership/integration between another project. The full list of current gauges can be found here: https://app.frax.finance/gauge

The FXS allocated to each gauge strategy is referred to to as its "gauge weight." 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 control over the future FXS emission rate. Each veFXS is equal to 1 vote.

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. **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 each Wednesday. Any user can change their weight allocation every 10 days.**

Since FXS gauge emissions are fixed and halve every year, governance can decide whether to allocate part of protocol cash flows or newly minted FRAX to gauge rewards after a few years. Thus, 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. Gauge strategies are currently a vital part of incentivizing the right behavior for the growth of the Frax ecosystem.



Vote with veFXS

You can only change individual gauge votes once per 10 days. Make sure to do decreases first and increases last.

Name	Address	Weight	Can Vote
Uniswap V3 FRAX/USDC	0x3EF2...B4B0	0.00%	Yes
mStable FRAX/mUSD	0x3e14...6AeC	0.00%	Yes
Uniswap V3 FRAX/DAI	0xF224...f53e	0.00%	Yes
Sushi FRAX/SUSHI	0xb4Ab...7d5D	0.00%	Yes
StakeDAO sdFRAX3CRV-f	0xEB81...6Da2	0.00%	Yes
Gelato Uniswap FRAX/DAI	0xcdfc...2F6f	0.00%	Yes
StakeDAO sdETH-FraxPut	0x0A53...BB56	0.00%	Yes
Uniswap V3 FRAX/agEUR	0xf8ca...69E8	0.00%	Yes
Vesper Orbit FRAX	0x6981...48d6	0.00%	Yes
Temple FRAX/TEMPLE	0x1046...Ef16	0.00%	Yes
Curve VSTFRAX-f	0x1279...D117	0.00%	Yes
Aave aFRAX	0x0257...d65c	0.00%	Yes
Convex stkcvxFPIFRAX	0x0a08...7FC9	0.00%	Yes
Fraxswap FRAX/pitchFXS	0x9E66...a797	0.00%	Yes
Fraxswap FRAX/IQ	0x5e15...4490	0.00%	Yes
Fraxswap FRAX/IQ V2	0xBF33...2a0d	0.00%	Yes
Fraxswap FRAX/pitchFXS V2	0x899A...6e35	0.00%	Yes

SELECT GAUGE

0.00%

Please select a gauge

The gauge voting list on app.frax.finance/gauge used for allocating all FXS emissions. The aggregate weights of all votes decide which gauge strategy contract FXS tokens are sent to.

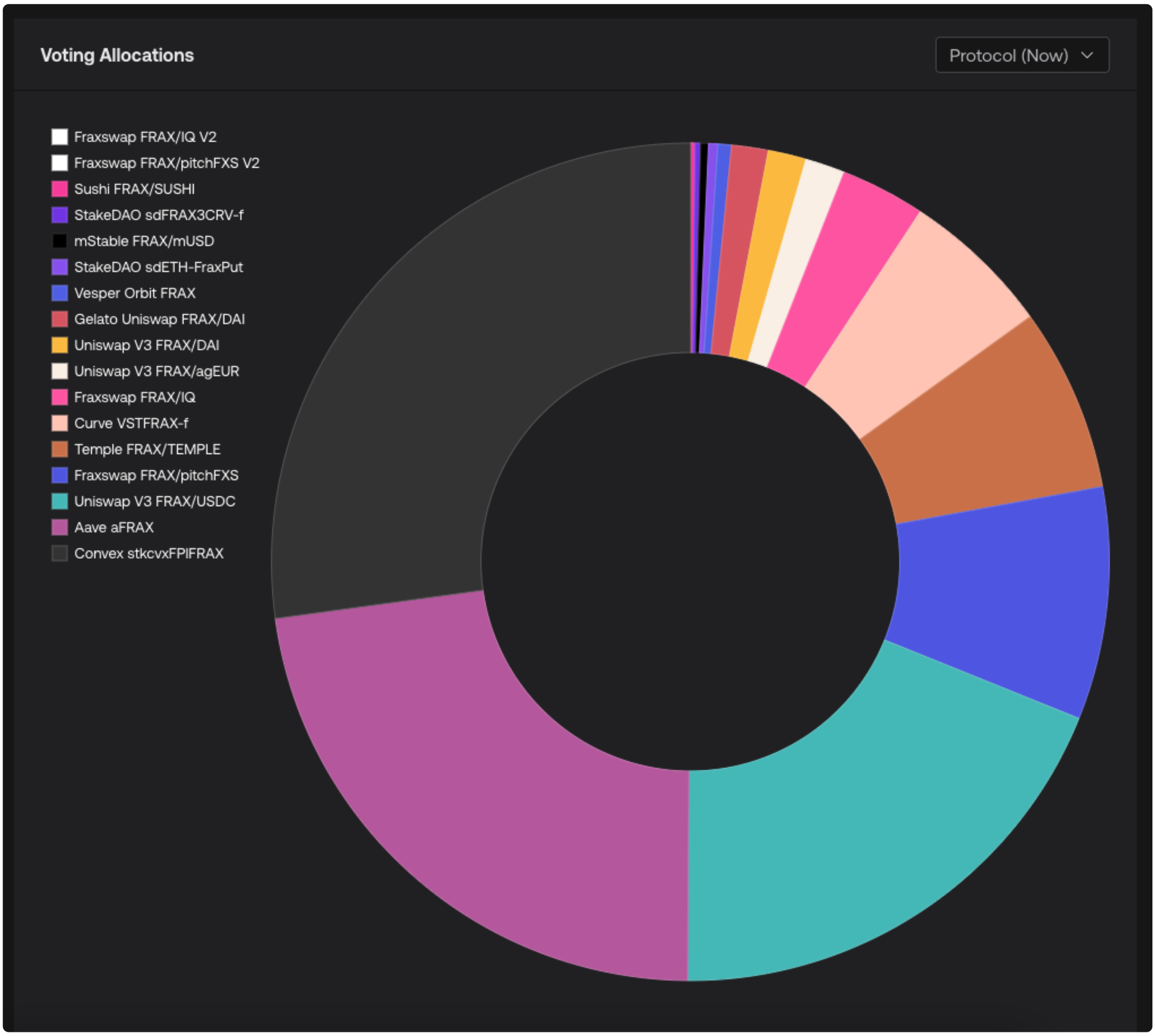
Current Types of Gauges

LP Gauge: an LP gauge is the most common type of gauge contract taking an ERC20 LP token as a deposit. Most gauges incentivize LP positions from Fraxswap, Curve, Uniswap v2, etc. Typically, these gauges offer a 2x veFXS boost of 4 veFXS per 1 FRAX in the LP position and an additional 1 year 2x timelock boost (unless otherwise specified).

Lending Gauge: a lending gauge is typically deployed to incentivize FRAX lending activity in a money market such as Aave, Fraxlend, Compound etc. The deposit token is aFRAX, fFRAX, cFRAX, etc. Lending gauges typically do not offer timelock boosts but offer up to 2x veFXS boost for 1 FRAX lent out per 4 veFXS.

Uniswap V3 Gauge: Uniswap v3 gauges take an NFT LP position as a deposit. These gauges are pre-configured at launch to accept NFT LPs at a specific tick range to incentivize only the exact concentrated liquidity position that governance approved for the pair. These gauges offer a 2x veFXS boost of 4 veFXS per 1 FRAX in the LP position and an additional timelock boost of 2x-3x for 1-3 year locks (specified for each gauge on its corresponding staking page).

Vault Gauge: a vault gauge takes a vault strategy token as a deposit such as a Stake DAO or Yearn Finance vault token. Vault gauges typically offer a 2x veFXS boost of 4 veFXS per 1 FRAX in the vault position and a timelock boost of 2-3x for 1-3 year (specified for each gauge).



Pie chart displaying the total weights for a particular weekly gauge period. Periods change on Wednesdays at 11:59:59 UTC. Votes can be changed by veFXS stakers once every 10 days.

veFXS Boosts & Timelock Boosts

Users who stake tokens in a gauge contract earn boosts to their APR based on the amount of veFXS they have. Additionally, users that lock their deposited tokens within the gauge contract for a specific period of time will earn a further additive boost thus enabling stacking of both boosts for maximal APR. Since gauge weights change weekly, locked LPs in gauges do not get their deposits unlocked if the gauge weight changes. See the veFXS spec page for an explanation of how boosts are calculated. Each gauge strategy can have different boost terms depending on the end behavior it is incentivizing. For example, a lending gauge might have a very low timelock boost since lending FRAX does not provide peg stability and thus not need extra rewards. However, a Curve, Uniswap, or Fraxswap liquidity gauge might provide a high timelock boost because locking liquidity helps FRAX peg strength during volatility.

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. Any FRAX pool (including cross-chain pools) can be added as a gauge in the future. The veFXS gauge system is completely agnostic to the deposit token within a gauge as long as the FRAX stablecoin is being used within the strategy. Essentially, veFXS gauges are the money layer gauge weights of DeFi while other gauges (such as 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.

Fraxbridge & Cross Chain FRAX & FXS

Bridge Mechanism

Multichain FRAX+FXS that is fungible across many networks

This method is deprecated for the newer Fraxferry mechanism.
Please see Fraxferry documentation here: https://docs.frax.finance/fraxferry/overview

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.

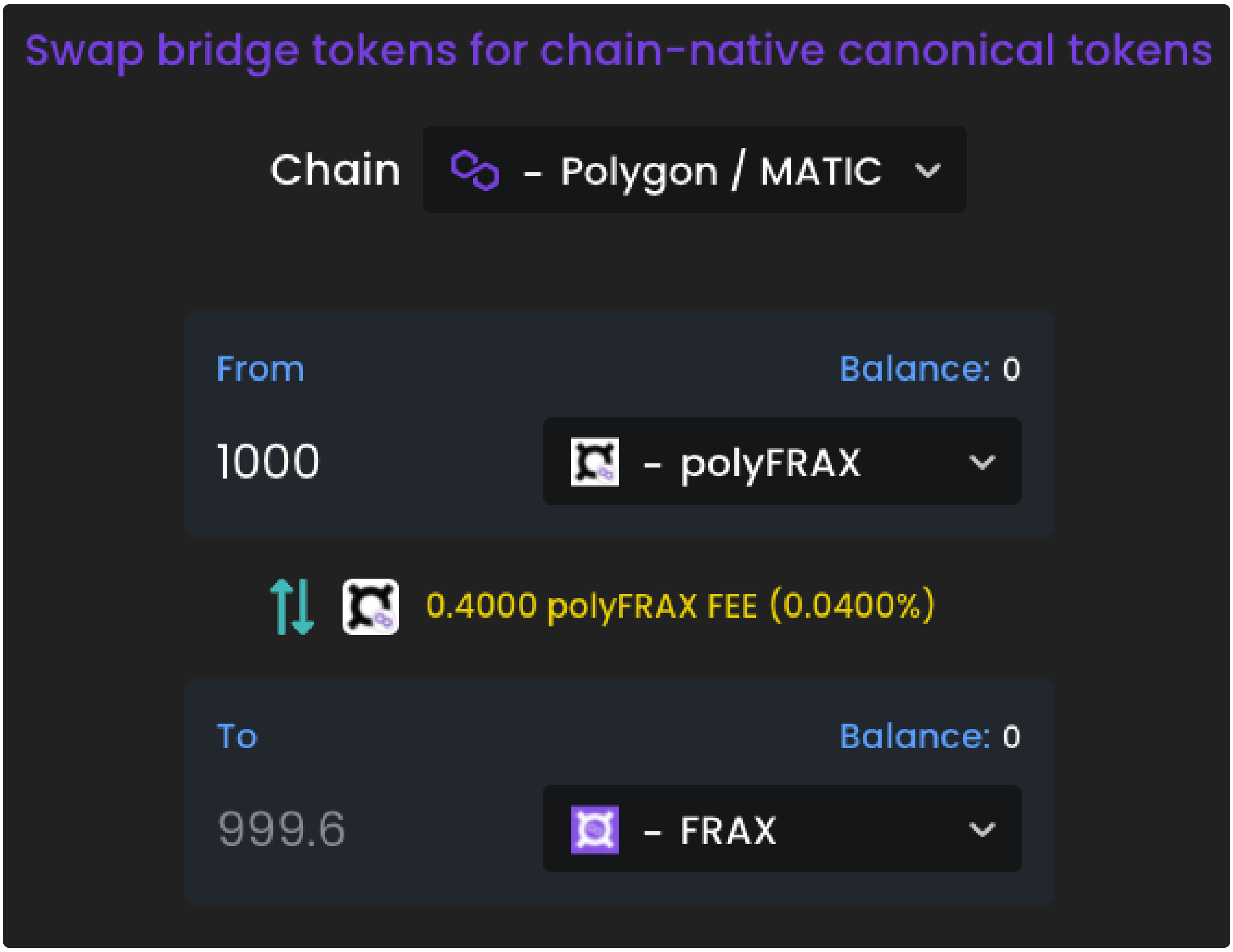
Bridged FRAX/FXS: Tokens that are brought to the current chain from another network using a supported bridge protocol. The naming convention for bridged tokens maintains a prefix designation for the bridge used to bring them to the current network. Ex: AnySwap bridged FRAX from ETH to AVAX is known as 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 and arbitraged.

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).



Swap bridge tokens for chain-native canonical tokens

Chain - Polygon / MATIC

From — Balance: 0

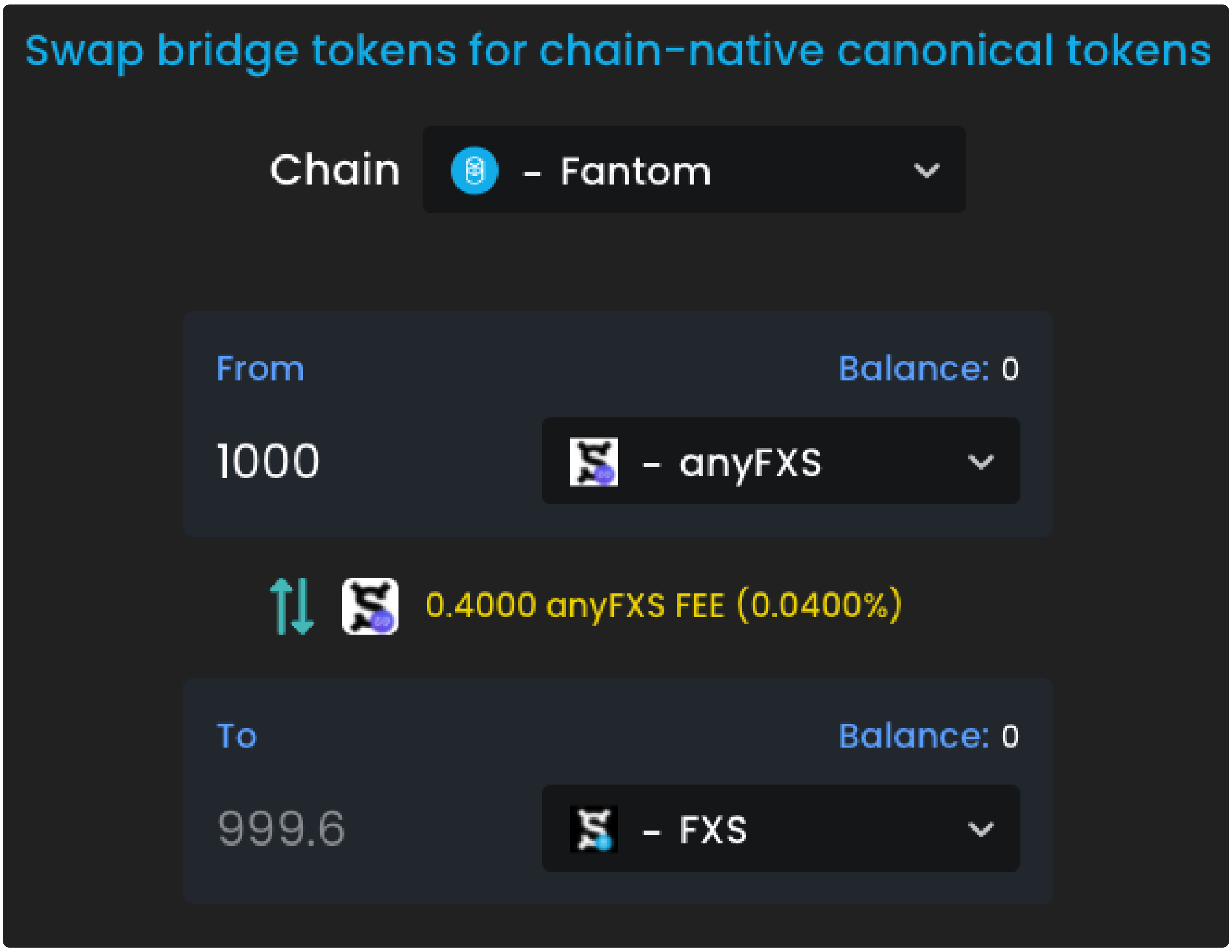
1000 - polyFRAX

0.4000 polyFRAX FEE (0.0400%)

To — Balance: 0

999.6 - FRAX

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.



Swap bridge tokens for chain-native canonical tokens

Chain - Fantom

From Balance: 0

1000 - anyFXS

0.4000 anyFXS FEE (0.0400%)

To Balance: 0

999.6 - FXS

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

Canonical Token Addresses

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

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

Frax Price Index

Overview (CPI Peg & Mechanics)

A novel stablecoin pegged to a basket of consumer goods

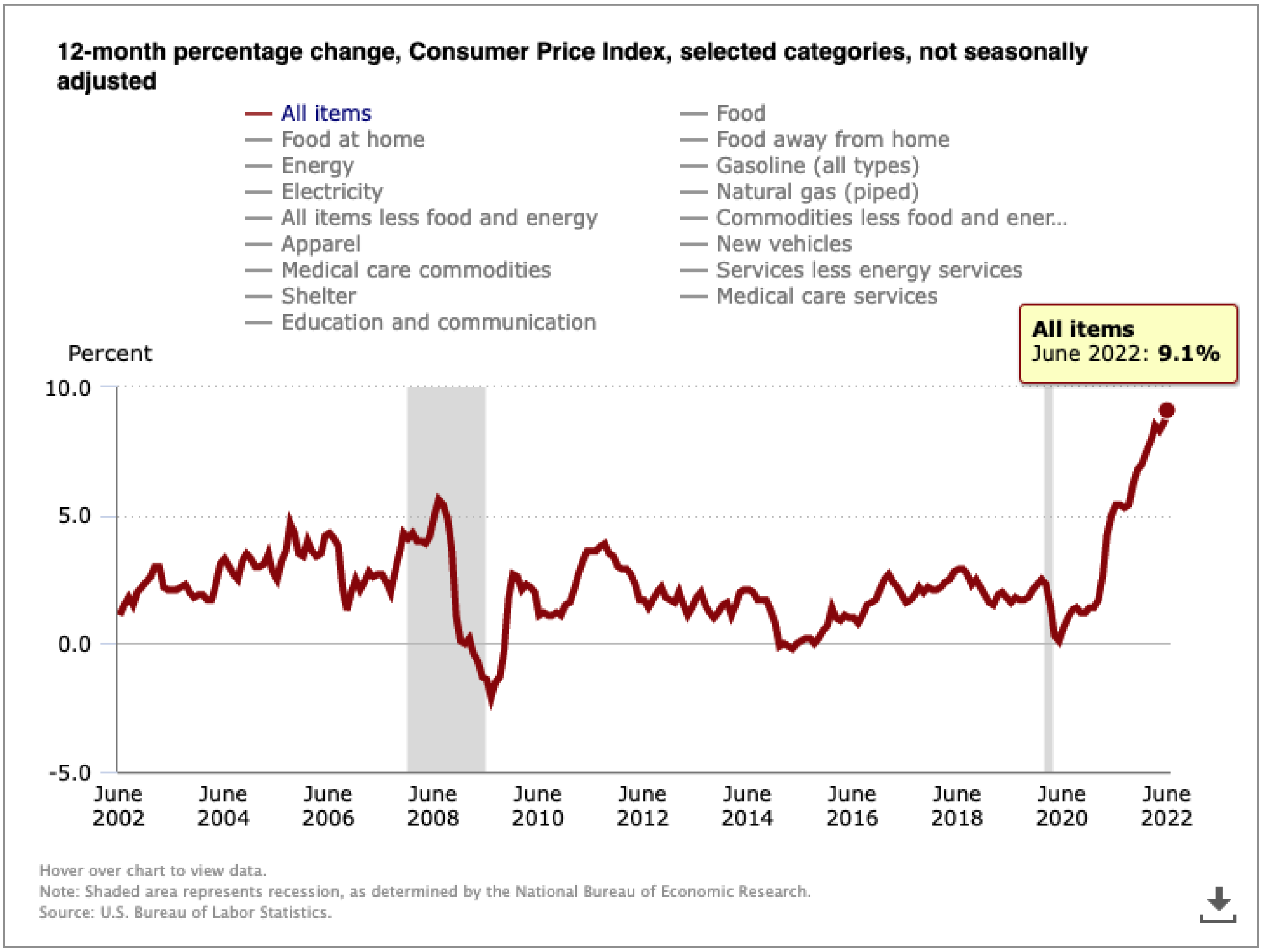
The Frax Price Index (FPI) is the second stablecoin of the Frax Finance ecosystem. FPI is the first stablecoin pegged to a basket of real-world consumer items as defined by the US CPI-U average. The FPI stablecoin is intended to keep its price constant to the price of all items within the CPI basket and thus hold its purchasing power with on-chain stability mechanisms. Like the FRAX stablecoin, all FPI assets and market operations are on-chain and use AMO contracts.

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.

Inflation & Peg Calculation

The FPI uses the CPI-U unadjusted 12 month inflation rate reported by the US Federal Government: https://www.bls.gov/news.release/cpi.nr0.htm A specialized Chainlink oracle commits this data on-chain immediately after it is publicly released. The oracle's reported inflation rate is then applied to the redemption price of FPI stablecoins in the system contract. This redemption price grows per second on-chain (or declines in rare cases of deflation). The peg calculation rate is updated once every 30 days when bls.gov releases their monthly CPI price data.

Thus, the FPI peg tracks the above 12 month inflation rate and pegs to it at all times from the FPI redemption contract. Thus, when buying FPI stablecoins for another asset (such as ETH) the trader is taking the position that CPI purchasing power is growing faster over time than the sold ETH. If selling FPI for ETH, then the trader is taking the position that ETH growth is outpacing the CPI inflation rate of the US Dollar.



The 12 month inflation rate is used as the peg target of FPI. At an inflation rate of 9.1% in June 2022, the FPI peg would grow at a rate of 9.1% against USD for the next 30 days. Note that during deflationary periods (June 2009) where the rate of inflation is negative, the FPI peg would decrease against USD.

FPI as a Unit of Account

FPI 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. While FPI can be considered an inflation resistant yield asset, its primary motivation is to create a new stablecoin to denominate transactions, value, and debt. Denominating DAO treasuries and measuring revenue in FPI as well as benchmarking performance against an FPI trading pair helps better gauge if value accrual is actively growing against inflation in real terms. It also helps ground on-chain economics to baskets of real world assets.

At first, the treasury will be comprised solely of $FRAX, but will expand to include other crypto-native assets such as bridged BTC, ETH, and non-crypto consumer goods and services.

FPI Stability Mechanism

FPI uses the same type of AMOs as the FRAX stablecoin however it is modeled to keep a 100% collateral ratio (CR) at all times. This means that for the collateral ratio to stay at 100% the protocol balance sheet must be growing at least at the rate of CPI inflation. Thus, AMO strategy contracts must earn a yield proportional to CPI otherwise the CR will decrease to below 100%. During times that AMO yield is under the CPI rate, a TWAMM AMO will sell FPIS tokens for FRAX stablecoins to keep the CR at 100% at all times. The FPIS TWAMMs will be removed when the CR returns to 100%.

Frax Price Index Share (FPIS)

The Frax Price Index Share (FPIS) is the governance token of the Frax Price Index (FPI) stablecoin. FPIS is interconnected to the Frax Share (FXS) token thus it is referred to as a "linked governance token." The FXS & FPIS are economically linked programmatically in a similar way that a layer 1 token is connected to a dapp token on its network.

FXS is the base token of the Frax Ecosystem thus FXS will always accrue value as both FRAX & FPI stablecoins grow no matter what. FXS accrues value proportional to the aggregate growth of the entire Frax economy as a whole similar to how ETH captures value from the sum total of all dapp economic activity that pays gas for access to Ethereum blockspace. FPIS tracks FPI growth specifically similar to a specific ERC20 token tracks growth of its own protocol rather than the entire L1 economy. If you think the Frax economy is undervalued as a whole, you should want to own more FXS. If you think people are undervaluing FPI growth specifically, you should increase more exposure to FPIS. It's the exact same dynamic of whether you would invest in a specific project or you invest in ETH instead. If you think the ETH economy as a whole is undervalued, you'd buy more ETH. But if you think a specific project will grow faster than the sum total of the economy, you'll want to own that specific token rather than the L1 token.*

The Frax Collateral Ratio (FCR) is the ratio of FRAX stablecoins directly backing FPI tokens. The FCR is directly calculated before value distribution to FPIS & veFPIS token holders. The FCR specifically is used for FXS value capture of the system.

Whenever excess FPI balance sheet value is distributed back to FPIS token holders it will pass through an "FCR contract" or function call that calculates how much "FRAX collateral FPI uses."

Essentially, any economic productivity above the inflation rate goes to FPIS holders. Since the FPI is pegged to a basket of consumer items, it represents a claim on the value of the basket. The value the protocol generates in excess beyond that basket is captured by FPIS holders.

FPIS Token Distribution (100,000,000 FPIS total supply)

No FPIS tokens can be minted over the 100m genesis supply except to keep the FPI peg to the CPI rate and keep the CR constant at 100%.

35% Frax Finance Treasury 35,000,000
FXS voters have total control in voting how to distribute these tokens through governance.

30% FPI Protocol Treasury 30,000,000
FPIS voters themselves have total control in how to distribute these tokens.

25% Core Developers & Contributors Treasury 25,000,000
4 year back-vested to start from February 20th 2022 at the same time as airdrop genesis with a 6 month cliff. Distribution occurs on/around 20th of each month. This treasury will be staked in veFPIS system in 4 week intervals between monthly distributions when veFPIS is released.

10% February 2022 Airdrop to FXS Holders 10,000,000
Snapshot on February 20th 11:59:59 UTC 2022 and claimable until August 20th 11:59:59 UTC 2022

***No discussion of value capture in this documentation is investment advice. Governance token mechanics are merely meant to describe how the Frax/FPI system functions.**

Oracle

Description

The oracle uses Chainlink's December 2021 CPI-U data point (provided by Fiews) as the 'base' index for determining the peg price. Each month, the change / delta percent of the index is applied to the previous month's data point to determine the peg price. December 2021 was chosen because the oracle contract requires two initial 'historical' data points.

Example:

December 2021 CPI-U: 280.126

January 2022 CPI-U: 281.933

Delta is (281.933 / 280.126) - 1 = 0.64506686%

Assuming December 2021 is $1, applying the delta percentage gives

$1 x (1.0064506686) = $1.0064506686 as the peg price.

If February 2022 CPI-U data was 284.182, the delta would be (284.182 / 281.933) - 1 = 0.79770726%.

Applying this to the previous peg price would give $1.0064506686 * (1 + 0.0079770726) = $1.0144791987 as the new peg price. In other words, you would need this many Feb 2022 dollars to buy one Dec 2021 dollar.

Raw data source: https://data.bls.gov/timeseries/CUSR0000SA0

Chainlink / Fiews Job: https://market.link/jobs/44964ac4-d302-4141-8f94-67e58e34b88d

Code

frax-solidity/CPITrackerOracle.sol at master · FraxFinance/frax-solidity

GitHub

FPI Controller Pool

Description

The FPI Controller Pool has various helper functions related to FPI and its peg.

Mint / Redeeming

Users can mint FPI with FRAX or redeem FPI for FRAX. There is a small fee associated with this, initially 0.30%.

twammToPeg

The twammToPeg function is used by the protocol to introduce market pressure to bring the market price of FPI up (or down) to the target peg price.

giveFRAXToAMO / receiveFRAXFromAMO

The contract can lend FRAX collateral to various AMOs that earn yield (among other things).

Code

frax-solidity/FPIControllerPool.sol at master · FraxFinance/frax-solidity
GitHub

veFPIS

An updated and modular veFPIS

veFPIS is an updated and vesting+yield system for the FPIS governance token. Similar to veFXS, users may lock their FPIS for up to 4 years for four times the amount of veFPIS (e.g. 100 FPIS locked for 4 years returns 400 veFPIS). veFPIS is not a transferable token nor does it trade on liquid markets. It's akin to an account based point system that signifies the vesting duration of the wallet's locked FXS tokens within the protocol.

The veFPIS balance linearly decreases as tokens approach their lock expiry, approaching 1 veFPIS per 1 FPIS at zero lock time remaining.

Double Whitelist & Modular Functionality

veFPIS has an additional "DeFi whitelist" for smart contracts that add modular functionality to the staking system. Governance can approve each new whitelisted DeFi feature. For example, a liquidation contract can be whitelisted by governance which allows a staker's underlying FPIS tokens to be liquidated if they borrow against their veFPIS balance. Users will have to approve each DeFi whitelisted contract to spend their FPIS tokens before the new functionality can be unlocked for each user. This allows the staking system to remain fully trustless so that no extra logic can access a staker's veFPIS balance without a staker's approval thus keeping modules opt-in per wallet address. This system allows governance to add new iterative functionality to veFPIS staking such as "slashing conditions" and new ways to earn higher yield (and potentially be slashed if users opt-in) by adding smart contracts which allow veFPIS holders to vote on CPI gauge weights, borrow FPI, or control liquidity deployment.

Fraxswap

Technical Specifications

A Unique Time Weighted Average Market Maker for Trustless Monetary Policy

Overview

Fraxswap is the first constant product automated market maker with an embedded time-weighted average market maker (TWAMM) for conducting large trades over long periods of time trustlessly. It is fully permissionless and the core AMM is based on Uniswap V2. This new AMM helps traders execute large orders efficiently and will be heavily used by the Frax Protocol to increase the stability of the pegs for the FRAX & FPI stablecoins as well as return protocol excess profits to FXS holders through TWAMM purchases.
The motivation for building Fraxswap was to create a unique AMM with specialized features for algorithmic stablecoin monetary policy, forward guidance, and large sustained market orders to stabilize the price of one asset by contracting its supply or acquiring a specific collateral over a prolonged period. Specifically, Frax Protocol will use Fraxswap for: buying back and burning FXS with AMO profits, minting new FXS to buy back and burn FRAX stablecoins to stabilize the price peg, minting FRAX to purchase hard assets through seigniorage, and many more market operations in development.

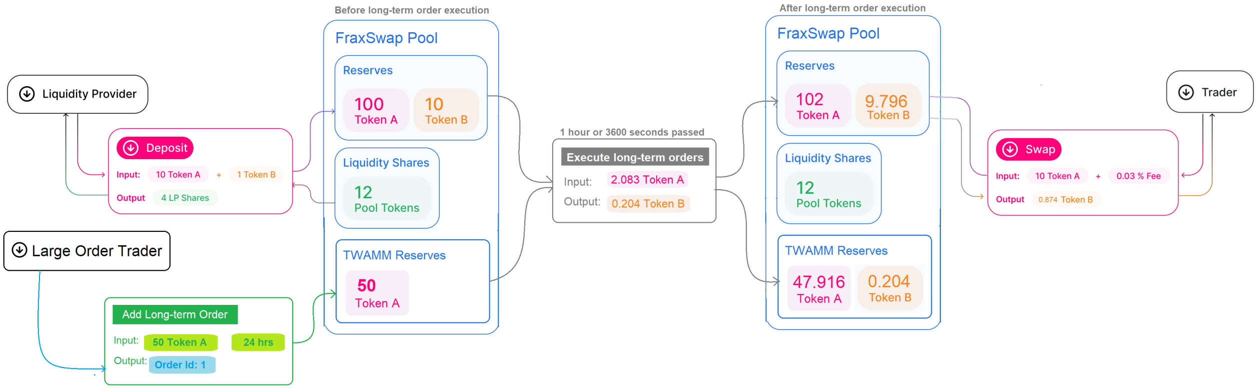


Diagram of order flow on a Fraxswap pair with a TWAMM order active.

Core AMM

The Core AMM for Fraxswap is Uniswap V2 and based on the full range xy=k constant product design. Fraxswap has adhered to many of the design decisions of Uniswap V2 as it extended the codebase to support TWAMMs. Fraxswap v2 plans to support concentrated liquidity & correlated asset liquidity in a unique & novel way to allow TWAMM functionality across such pairs as well.

Extensive documentation on Uniswap core functionality can be found here: https://docs.uniswap.org/protocol/V2/concepts/protocol-overview/how-uniswap-works

Time-weighted Average Market Maker (TWAMM)

Fraxswap is the first live TWAMM implementation. The embedded TWAMM adheres to Paradigm's original whitepaper specifications. Features mentioned in the whitepaper are used to optimize the execution of long-term orders which include: order pooling and aligning order expiries (hourly). Long-term orders are executed prior to any interaction with Fraxswap, this means the long-term orders are executed first before the AMM interaction and once per block (see above schematic). Fraxswap implements an approximation formula of Paradigm's original formula and allows for a simplified, more gas-efficient TWAMM.

The TWAMM whitepaper describes the fundamental mechanics:
"Each TWAMM instance facilitates trading between a particular pair of assets, such as ETH and [FRAX].
The TWAMM contains an embedded AMM, a standard constant-product market maker for [...] two assets. Anyone may trade with this embedded AMM at any time, just as if it were a normal AMM.
Traders can submit long-term orders to the TWAMM, which are orders to sell a fixed amount of one of the assets over a fixed number of blocks — say, an order to sell 100 ETH over the next 2,000 blocks.
The TWAMM breaks these long-term orders into infinitely many infinitely small virtual sub-orders, which trade against the embedded AMM at an even rate over time. Processing transactions for each of these virtual sub-orders individually would cost infinite gas, but a closed-form mathematical formula allows us to calculate their cumulative effect only when needed.
The execution of long-term orders will push the embedded AMM's price away from prices on other markets over time. When this happens, arbitrageurs will trade against the embedded AMM's price to bring it back in line, ensuring good execution for long-term orders.
For example, if long-term sells have made ETH cheaper on the embedded AMM than it is on a particular centralized exchange, arbitrageurs will buy ETH from the embedded AMM, bringing its price back up, and sell it on the centralized exchange for a profit."

Interactive comparison of Paradigm's TWAMM formula and Fraxswap's TWAMM formula can be found at:
https://www.desmos.com/calculator/wp4zrkh6uj

Protocol Uses & DAO to DAO Swaps

Using Fraxswap for critical system functionality and market operations

Fraxswap is specifically designed for use in the Frax Protocol’s critical system functions through TWAMM orders. The motivation for building Fraxswap was to create a unique AMM with specialized features for algorithmic stablecoin monetary policy, forward guidance, and large sustained market orders to stabilize the price of one asset by contracting its supply or acquiring a specific collateral over a prolonged period. Specifically, Frax Protocol will use Fraxswap for: buying back and burning FXS with AMO profits, minting new FXS to buy back and burn FRAX stablecoins to stabilize the price peg, minting FRAX to purchase hard assets through seigniorage, and many more market operations in development.

Fraxswap is a fully permissionless AMM which means other protocols can create their own LP pairs in any token and use Fraxswap for the same (or other novel) use cases.

Ideal Fraxswap use cases by other protocols, stablecoin issuers, & DAOs include:

1.) Accumulation of a treasury asset (such as stablecoins) over time by slowly selling governance tokens.
2.) Buying back governance tokens slowly over time with DAO revenues & reserves.
3.) Acquire another protocol's governance tokens slowly over time with the DAO's own governance tokens (similar to a corporate acquisition/merger but in a permissionless manner).
4.) Defending "risk free value" (RFV) for treasury based DAOs such as Olympus, Temple, and various projects where the backing of the governance token is socially or programmatically guaranteed.

To use Fraxswap for monetary policy, the best method is to create a token pair and add protocol controlled liquidity. Then, TWAMM orders can be placed in any size in either direction as desired for forward guidance and rebalancing of the DAO's net assets. See the technical specifications section to understand slippage calculations and liquidity optimizations for TWAMMs.

Fraxlend

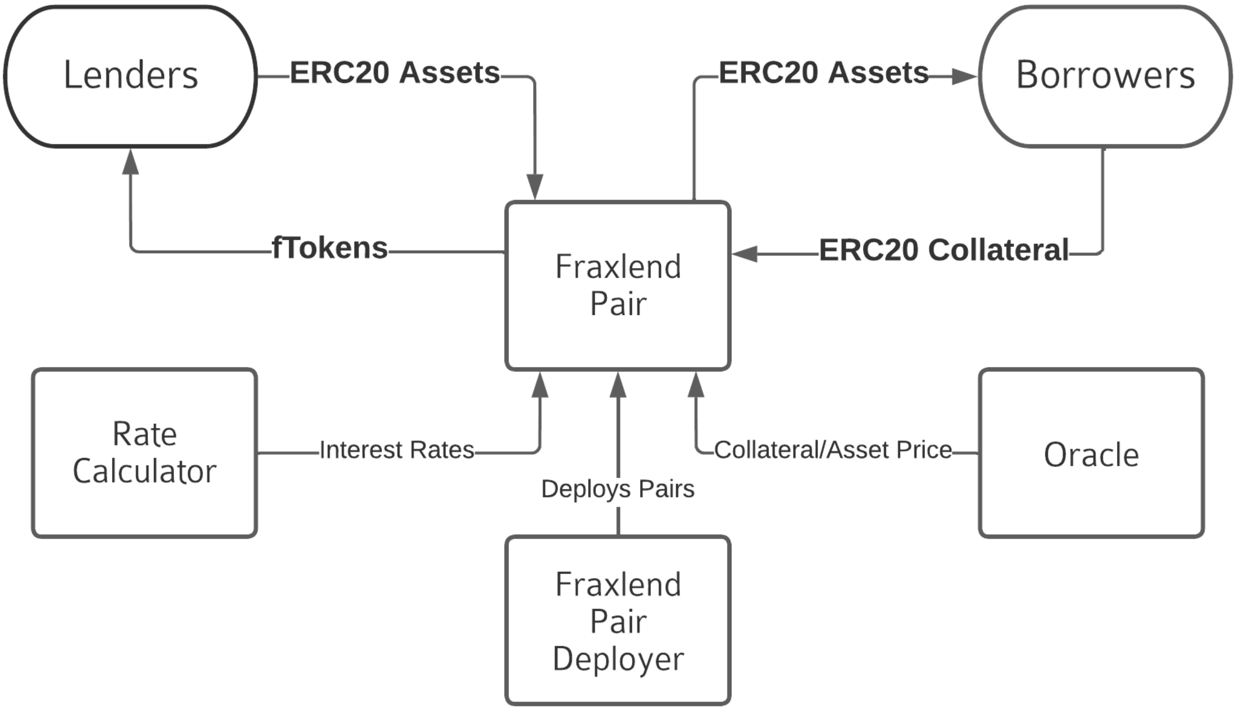
Fraxlend Overview

Fraxlend is a lending platform that provides lending markets between a pair of ERC-20 assets. Each pair is an isolated market which allows anyone to participate in lending and borrowing activities.

Lenders are able to deposit ERC-20 assets into the pair and receive yield-bearing fTokens. As interest is earned, fTokens can be redeemed for ever-increasing amounts of the underlying asset.

Fraxlend also supports the ability to create custom Term Sheets for over-the-counter debt structuring. Fraxlend Pairs can be created with features like: maturity dates, restricted borrowers & lenders, under-collateralized loans, and limited liquidations.

Ecosystem Participants



The primary participants in Fraxlend are **Lenders** and **Borrowers**, these participants interact with individual **Pairs**.

- Lenders provide Asset Tokens to the pair in exchange for fTokens
- Borrowers provide Collateral Tokens to the pair and in exchange receive Asset Tokens. Borrowing incurs an interest rate which is capitalized and paid to lenders upon redemption of fTokens.

Beyond **Pairs**, the rest of the ecosystem includes: **Oracles**, **Rate Calculators**, and the **Fraxlend Pair Deployer**

- Each pair relies on one (or two) **ChainLink Oracles** to determine the market rate for both the Asset Token and the Collateral Token. Other oracle implementations (like TWAPs) are supported in limited cases.

- Each pair can be deployed with a different **Rate Calculator**. These contracts calculate the interest rate based on the amount of available capital to borrow. Typically less borrowing will lead to lower rates, with more borrowing leading to higher rates.
- Each pair is deployed by a **Deployer** contract
- The pair **Registry** keeps track of all Fraxlend Pairs deployed

Key Concepts

Pairs

Each pair is an isolated market to borrow a single ERC-20 token (known as the Asset Token) by depositing a different ERC-20 token (known as the Collateral Token).

When Lenders deposit Asset Tokens into the Pair, they receive fTokens. fTokens are ERC-20 tokens and are redeemable for Asset Tokens (sometimes referred to as the underlying asset). As interest is accrued, fTokens are redeemable for increasing amounts of the underlying asset.

Borrowers deposit Collateral Tokens into the Pair and in exchange receive the right to borrow Asset Tokens

Loan-To-Value

Each borrower's position has a Loan-To-Value (LTV). This represents the ratio between the value of the assets borrowed and the value of the collateral deposited. The LTV changes when the exchange rate between Asset and Collateral Tokens move or when interest is capitalized.

If a borrower's LTV rises above the Maximum LTV, their position is considered unhealthy. In order to remediate this, a borrower can add collateral or repay debt in order to move the LTV back into a healthy range.

The Maximum LTV value is immutable and configured at deployment. Typically the value is set to 75%. Custom Term Sheet deployments can set this value manually, even creating under-collateralized loans by setting the value above 100%. Maximum LTV configured above 100% must be accompanied by a borrower whitelist to protect against malicious actors. The configured value can be found by calling the `maxLTV()` function on the pair.

Rate Calculator

Each pair is configured to use a specific Rate Calculator contract to determine interest rates. At launch, Fraxlend supports two types of Rate Calculators.

- **Time-Weighted Variable Rate Calculator -** allows the interest rate to change based on the amount of assets borrowed, known as the utilization. When utilization is below the target, interest rates will adjust downward, when utilization is above the target, interest rates will adjust upward. For more information see Advanced Concepts: Time-Weighted Variable Interest Rate
- **Linear Rate Calculator** - calculates the interest rate purely as a function of utilization. Lower utilization results in lower borrowing rate, while higher utilization results in higher borrowing rates. For more information see Advanced Concepts: Linear Rate
- **Variable Rate V2** - calculates the interest rate as a function of utilization. The interest rate responds immediately to changes in utilization along a rate function f(Utilization) = rate. However, the slope of the function increases when utilization is above the target and decreases when utilization is below the target. This means that rates will respond instantly to changes in utilization. Extended periods of low or high utilization will change the shape of the rate curve. For more information see ********************LINK HERE*************************

Liquidations

When a borrowers LTV rises above the Maximum LTV, any user can repay all or a portion of the debt on the borrower's behalf and receive an equal value of collateral plus a liquidation fee. The liquidation fee is immutable and defined at deployment. By default the value is set to 10% and can be accessed by calling the `liquidationFee()` view function on the pair.

fToken Share Price

When lenders deposit Asset Tokens they receive fTokens at the current fToken Share Price. fTokens represent a lender's share of the total amount of underlying assets deposited in the pair, plus the capitalized interest from borrowers. As interest accrues, the Share Price increases. Upon redemption, the fTokens are redeemable for an ever-increasing amount of Asset Tokens which includes capitalized interest. To check the current fToken Share Price, call the `totalAsset()` view function and compare the value of amount/shares.

Vault Account

The Vault Account is core concept to all accounting in the Pair. A Vault Account is a struct which contains two values:

1. The total **Amount** of tokens in the Vault Account.

2. the total number of **Shares** in the Vault Account.

```
struct VaultAccount {
    // Represents the total amount of tokens in the vault
    uint128 amount; // Analogous to market capitalization

    // Represents the total number of shares or claims to the vault
    uint128 shares; // Analogous to shares outstanding
}
```

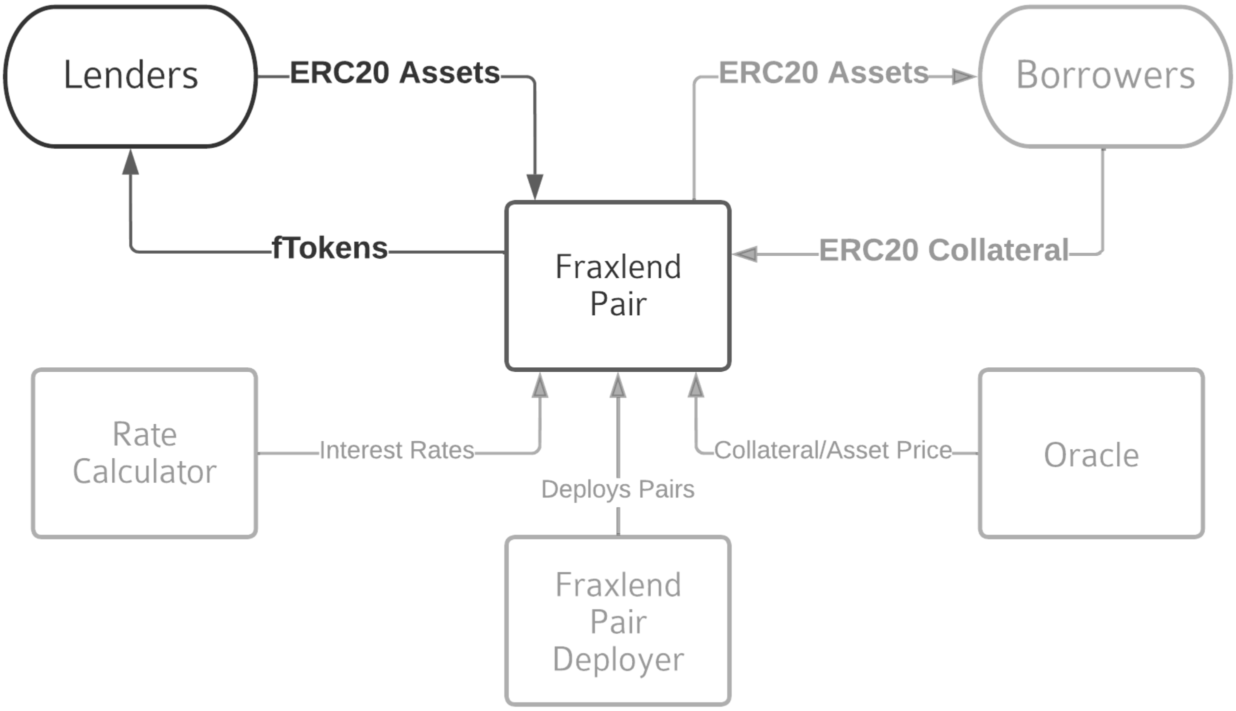
The Shares represent the total number of claims to the amount. Shares can be redeemed for Asset Tokens. The rate of redemption, known as the **Share Price** is determined by dividing the Amount value by the Shares value. It is essentially the exchange rate between Shares and the underlying Asset Token.

To convert between a value represented as an Amount into the corresponding Shares, divide the Amount by the Share Price. To convert from Shares to Amount, multiply by Share Price.

Shares = Amount / Share Price
Amount = Shares x Share Price

Lending

Lenders deposit Asset Tokens into the Pair and in return receive the corresponding number of Asset Shares (fTokens) depending on the current Share Price.



At any time a Lender can exchange their fTokens for the underlying Asset Tokens at the rate given by the current price. The fToken Share Price increases as more interest is accrued.

Accruing interest is the only operation which can change the Share Price. Because interest accrual is always positive, the number of Asset Tokens that each fToken is redeemable for cannot decrease.

A Lending Example

Alice has deposited 100 FRAX tokens to be lent out to borrowers, the initial fToken Share Price was 1.00 and she received 100 fTokens. Since her initial deposit, the pair has earned 10 FRAX of interest. So the Amount shows 110 (100 deposited FRAX + 10 FRAX earned as interest). The current fToken Share Price is 1.10 (110 FRAX / 100 fTokens)

Asset Vault Account	Total
Amount	110
Shares	100
Alice Share Balance (fToken)	100
Bob Share Balance (fToken)	0

If Bob now deposits 100 FRAX for lending we would see the following changes. First the Amount in the pair will increase by 100. The current Share Price for fTokens is 1.10. Therefore Bob will receive 90.91 (100 / 1.10) fTokens for his deposit. The Asset Vault Account now looks like this:

Asset Vault Account	Total
Amount	210
Shares	190.91
Alice Share Balance (fToken)	100
Bob Share Balance (fToken)	90.91

As interest accrues, the Amount increases. If an additional 20 FRAX are accrued as interest the Asset Vault Account looks like this:

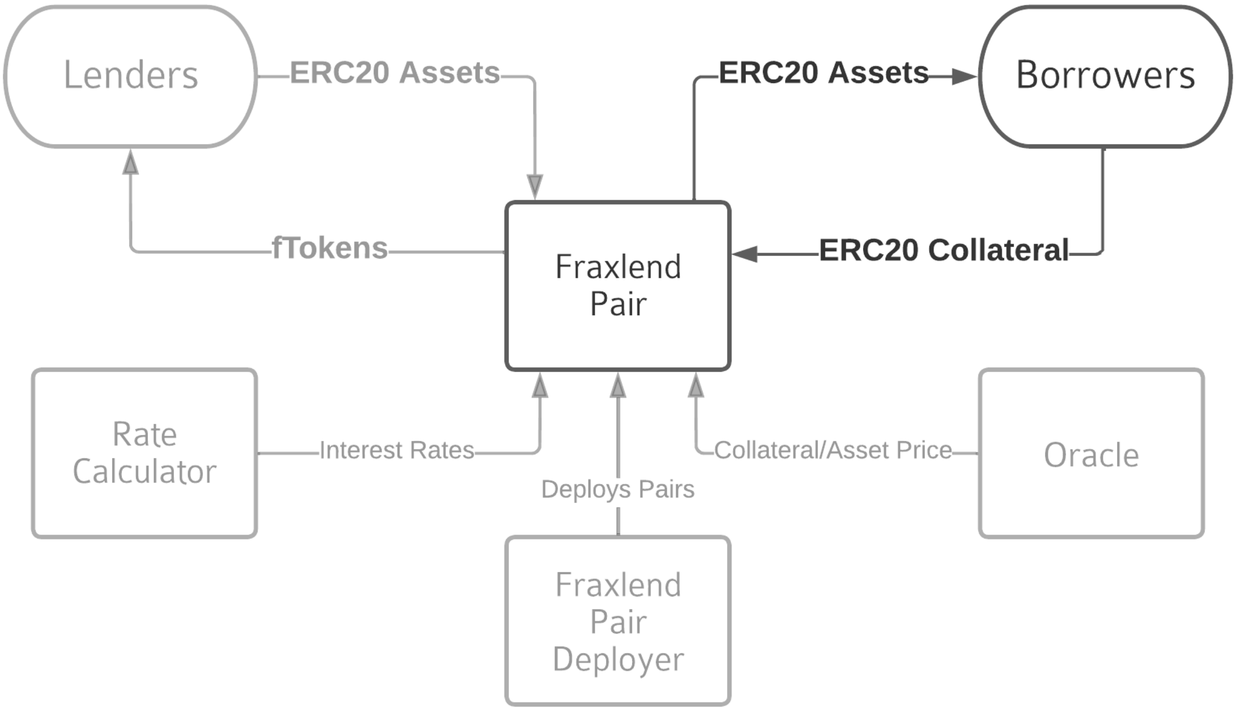
Asset Vault Account	Total
Amount	230
Shares	190.91
Alice Share Balance (fToken)	100
Bob Share Balance (fToken)	90.91

The Share Price has now increased to 1.2048 (230 FRAX / 190.91 fToken). This means that Bob can redeem his 90.91 fTokens for 109.52 FRAX. Likewise, Alice can redeem her 100 fTokens for 120.48 FRAX.

Over time, as more interest accrues, Alice and Bob can redeem their fTokens for an ever-increasing amount of the underlying asset.

Borrowing

Each pair gives the opportunity for users to borrow Asset Tokens, in return borrowers must supply the Pair with the appropriate amount of Collateral Tokens



As long as borrowers have an open position, interest accrues and is capitalized. This means that over time the amount a borrower owes increases by an amount equal to the interest they owe. In order for a borrower to receive their collateral back, they must return the original loan amount plus all accrued interest.

A Borrowing Example

Just like we used the Asset Vault Account to keep track of the total asset amounts and the corresponding number of shares, we use the Borrow Vault Account to keep track of the total amount borrowed, the capitalized interest, and the number of outstanding borrow shares.

Suppose that Alice has borrowed 100 FRAX ($100 of value) using $150 worth of ETH. Since her initial borrow she has accumulated 10 FRAX of interest. The Borrow Vault Account would appear as follows:

Borrow Vault Account	Total
Amount	110 FRAX
Shares	100
Alice Collateral Amount	$150 ETH
Alice Borrow Shares Balance	100

Remember that borrower's positions must remain below the Maximum Loan-To-Value (LTV). Because Alice's LTV is 73.33% she is below the max value of 75% and her position is considered healthy.

We calculate Alice's LTV in the following way. First we calculate the value of her loan to be $110 by multiplying her Borrow Share Balance (100 shares) by the Borrow Share Price (1.10), then multiply by the FRAX price given in USD (1.00). The value of her loan is $100 (100x1.10x1.00). The value of her collateral is $150. This gives $110 / $150 = 73.33%

Bob now borrows 100 FRAX, using $175 worth of ETH. Given the current Borrow Share Price of 1.10, his Borrows Shares Balance would be approximately 90.91. Unlike lenders, borrowers do not receive an ERC20 token representing their debt, instead the Share Balances are simply stored in the Pair. Now the Borrow Vault Account looks like this:

Borrow Vault Account	Total
Amount	210 FRAX
Shares	190.91
Alice Collateral Amount	$150 ETH
Alice Borrow Shares Balance	100
Bob Collateral Amount	$175 ETH
Bob Borrow Shares Balance	90.91

Suppose that the Pair accrues another 20 FRAX of interest. The Borrow Vault Account now looks like this:

Borrow Vault Account	Total
Amount	230 FRAX
Shares	190.91
Alice Collateral Amount	$150 ETH
Alice Borrow Shares Balance	100
Bob Collateral Amount	$175 ETH
Bob Borrow Shares Balance	90.91

The new Borrow Share Price is 1.2048 (230 / 190.91). This means that in order for Alice to repay her debt she will need to repay 120.48 FRAX (Alice Shares (100) x Share Price (1.2048)). Likewise, Bob would need to repay 109.52 FRAX (Bob Shares (90.91) x Share Price (1.2048)).

As interest accrues the amount required to repay the loan increases and the LTV of each position changes.

Alice's LTV: 80.32% (120.48 / 150)

Bob's LTV: 62.58% (109.52 / 175)

As the interest accrued and was capitalized, Alice's position has entered an unhealthy state as her LTV is above the Maximum LTV of 75%. This puts her at high risk of having her position liquidated.

Advanced Concepts

□Position Health & Liquidations

□Interest Rates

Position Health & Liquidations

Position Health

Each pair has a configured Maximum Loan-To-Value (LTV). Over time, as interest is capitalized, borrowers must add more collateral or repay a portion of their debt. Otherwise they risk having their position become unhealthy. To determine a borrower's LTV we use the value of the collateral and the value of the fTokens.

$$LTV = \frac{BorrowShares \times SharePrice}{CollateralBalance/ExchangeRate}$$

Share Price is the price of 1 fToken in Asset Token Units (i.e. AssetToken:fToken ratio)
Exchange Rate is the price of 1 Asset Token in Collateral Units (i.e. Collateral:Asset ratio)

Liquidations

When a borrowers LTV rises above the Maximum LTV, any user can repay the debt on the borrower's behalf and receive an equal value of collateral plus a liquidation fee. The liquidation fee is immutable and defined at deployment. By default the value is set to 10% and can be accessed by calling the `liquidationFee()` view function on the pair. The configured Maximum LTV can be found by calling the `maxLTV()` function on the pair.

Dynamic Debt Restructuring

Liquidators can close a borrower's position as soon as LTV exceeds the Maximum LTV (typically 75%). However, in cases of extreme volatility, it is possible that liquidators cannot close the unhealthy position before the LTV exceeds 100%. In this unlikely scenario bad debt is accumulated. In this case, the liquidator repays the maximum amount of the borrower's position covered by the borrower's collateral and the remaining debt is reduced from the total claims that all lenders have on underlying capital. This prevents the situation wherein lenders rush to withdraw liquidity, leaving the last lender holding worthless fTokens (commonly known as "bad debt" in other lending markets) and ensures the pair is able to resume operating normally immediately after adverse events.

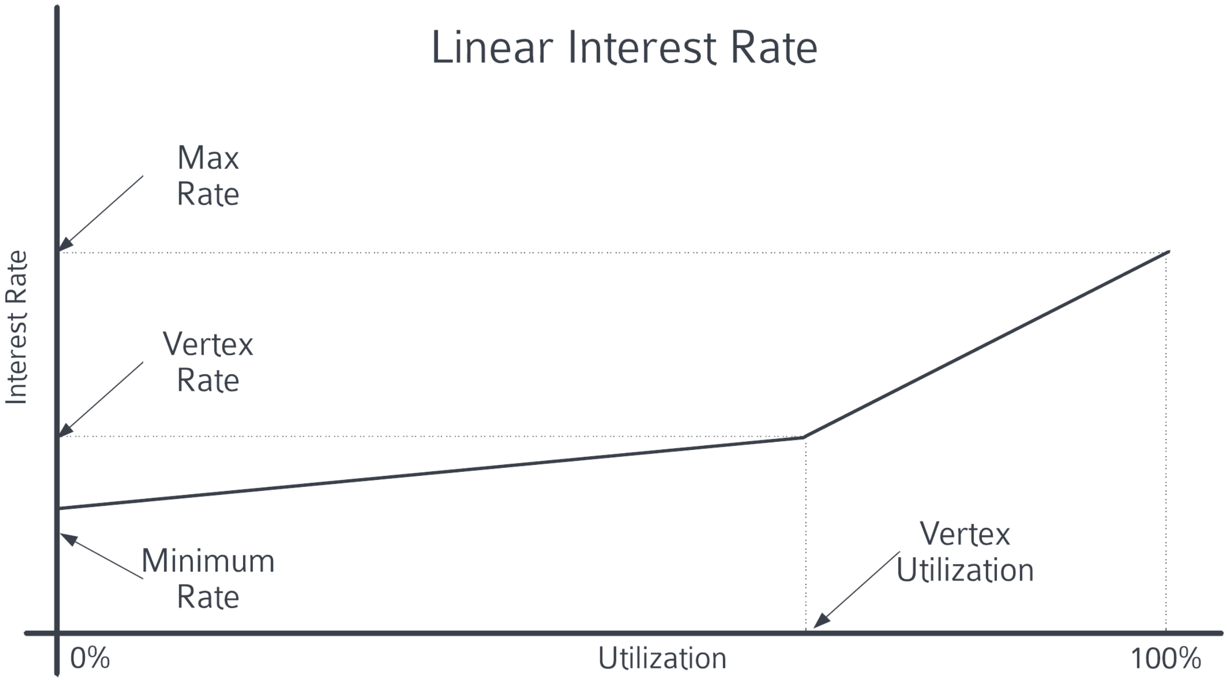
Interest Rates

Each pair is configured to change interest rates as a function of Utilization. Utilization is the total amount of deposited assets which have been lent to borrowers. Fraxlend currently has two interest rate models available for use:

1. The Linear Rate
2. Time-Weighted Variable Rate
3. Variable Rate V2

Linear Rate

The Linear rate is a configurable function that allows for two linear functions of the form `y = mx +b` . The function takes parameters which are defined at the time the Pair is created.



Minimum Rate: Rate when Utilization is 0%
Vertex Rate: Rate when utilization is equal to vertex utilization (i.e when the two slopes meet)
Vertex Utilization: The Utilization % where the two slopes meet
Maximum Rate: Rate when Utilization is 100%

These configuration values are immutable and fixed at time of Pair creation.

The Interest Rate is calculated using the following formulae:

If **Utilization Rate is equal to Vertex Utilization** then:

$$InterestRate(U = U_{vertex}) = Rate_{vertex}$$

If **Utilization Rate is less than Vertex Utilization** then:

$$InterestRate(U < U_{vertex}) = Rate_{min} + \left(U \times \frac{(Rate_{vertex} - Rate_{min})}{U_{vertex}}\right)$$

If **Utilization Rate is greater than Vertex Utilization** then:

$$InterestRate(U > U_{vertex}) = Rate_{vertex} + \left((U - U_{vertex}) \times \left(\frac{Rate_{max} - Rate_{vertex}}{1 - U_{vertex}}\right)\right)$$

Time-Weighted Variable Interest Rate

The Time-Weighted Variable Interest Rate adjusts the current rate over time. The variable interest rate is configured with a half-life value, given in seconds, which determines how quickly the interest rate adjusts.

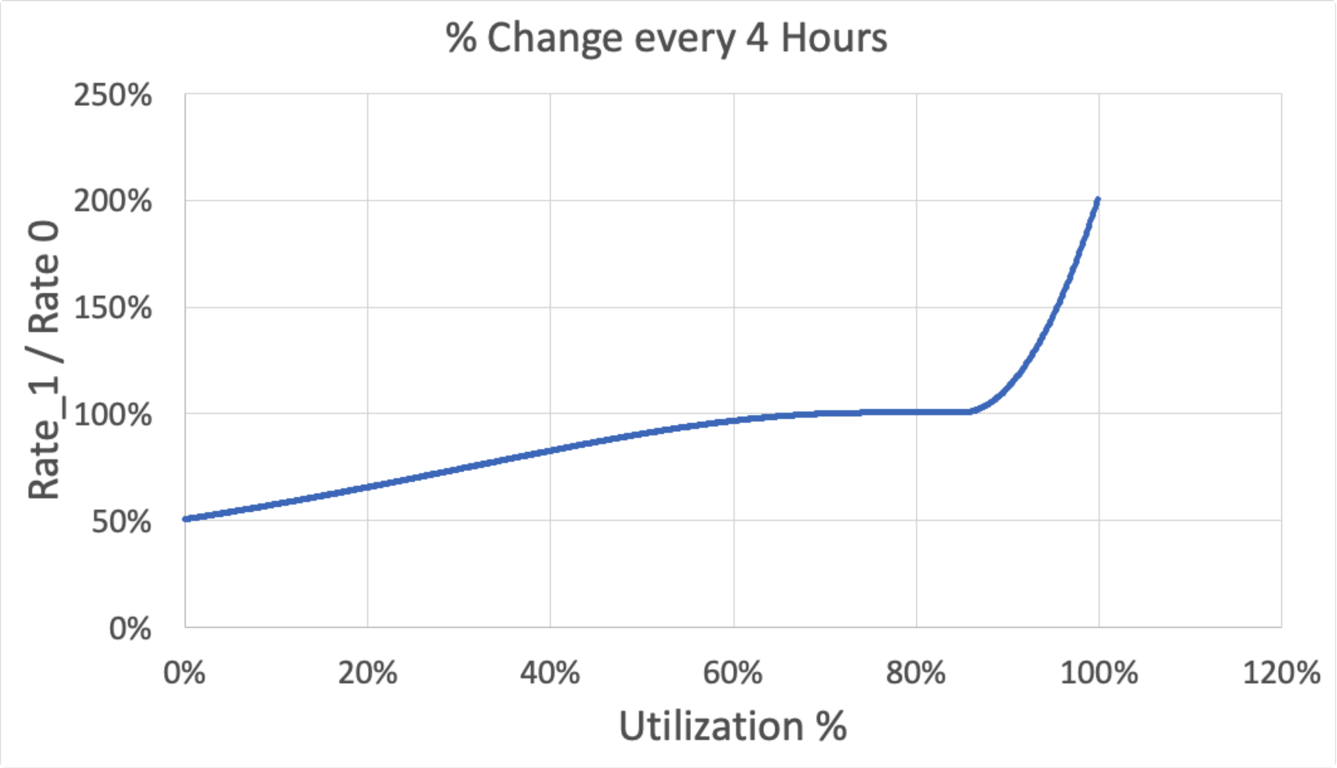
Minimum Rate: Minimum Rate to which interest can fall
Target Utilization Range: The Utilization Range where the interest rate does not adjust, it is considered in equilibrium with the market expectations.
Maximum Rate: Maximum Rate to which interest can rise
Interest Rate Half-Life: The time it takes for the interest to halve when Utilization is 0%. This is the speed at which the interest rate adjusts. ***In the currently available Rate Calculator, the Interest Rate Half-Life is 12 hours.***

The Time-Weighted Variable Interest Rate allows the market to signal the appropriate interest rate.

When **Utilization is below the target range**, the interest rate lowers, this encourages more borrowing and lenders to pull their capital, both of which push the Utilization Rate back into the target range.

When **Utilization is above the target range**, the interest rate increases which encourages more lending and less borrowing, bringing the Utilization back towards the target range.   Encouraging participants to borrow or lend as a function of both time and utilization.

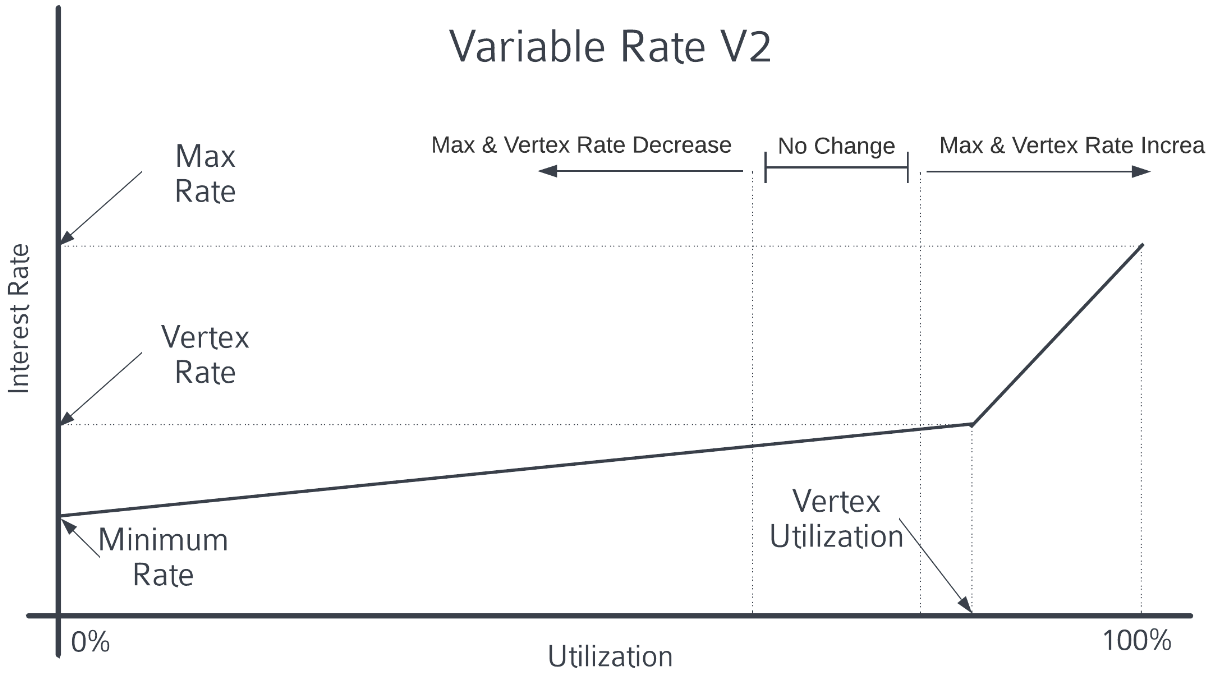
The following graph shows how the interest rate changes when the Interest Rate Half-Life is 4 hours, with a Target Utilization Range of 75% - 85%:



This allows the market, not the pair creator, to decide the appropriate interest rate of a given asset-collateral pair, the pair creator only needs to provide a target utilization.

Variable Rate V2 Interest Rate

The Variable Rate V2 Interest Rate combines the concepts from the Linear Interest Rate and the Time-Weighted Variable Interest Rate. Specifically, it utilizes the linear function from the Linear Interest Rate to determine the current rate, but adjusts the vertex and max rate utilizing the formula from the Time-Weighted Variable Interest Rate.



Just like the Time-Weighted Variable Interest Rate, the Variable Rate V2 takes a half-life and target utilization parameters. When utilization is low, the Vertex and Max Rate will decrease. If utilization is high, the Vertex and Max Rate will increase. The rate of the decrease/increase is determined by both the utilization and half-life. If utilization is 0% the Vertex and Max Rates will decrease by 50% each half-life. If the utilization is 100%, the increase will be 100% per half-life.

This means that interest rates will immediately respond to changes in utilization along the linear rate curve while at the same time adjusting to market conditions over the long term by scaling the slope of the linear rate curve.

Vault Account

The **Vault Account** is a struct used for accounting in the Fraxlend Pair:

```
struct VaultAccount {
    uint128 amount;
    uint128 shares;
}
```

The **Vault Account** contains two elements:

1. `amount` - represents a total amount
2. `shares` - represents claims against the amount

Lending Accounting

In lending, `amount` represents the total amount of assets deposited and the interest accrued.

When lenders deposit assets, the amount of assets deposited increases `amount` **,** the `shares` value is also increased by an amount such that the ratio between `amount` / `shares` remains unchanged.

When interest is accrued, `amount` is increased and `shares` remains the same. Each lender's share of the underlying assets are measured in shares.

When lenders remove liquidity, they redeem shares for the underlying asset. The `shares` amount will be decreased by the number of shares redeemed, the `amount` will be decreased such that the ratio between `amount` / `shares` remains unchanged.

Borrow Accounting

In borrowing, `amount` represents the total amount of assets borrowed and the interest accrued.

When borrowers receive assets, the number of tokens received increases `amount` **,** the `shares` value is also increased by an amount such that the ratio between `amount` / `shares` remains unchanged.

When interest is accrued, `amount` is increased and `shares` remains the same. Each borrower's debt is measured in shares.

When borrower's repay debt, `amount` is decremented by the amount of assets returned, `shares` is decreased by an amount such that the ratio of `amount` / `shares` remain unchanged. An individual borrowers shares balance is decremented by this number of shares.

API

FraxlendPairCore

An abstract contract which contains the core logic and storage for the FraxlendPair

constructor

```
constructor(bytes _configData, bytes _immutables, uint256 _maxLTV, uint256 _liquidationFee, u
```

The `constructor` function is called on deployment

Param	Type	Description
_configData	bytes	abi.encode(address _asset, address _collateral, address _oracleMultiply, address _oracleDivide, uint256 _oracleNormalization, address _rateContract, bytes memory _rateInitData)
_immutables	bytes	
_maxLTV	uint256	The Maximum Loan-To-Value f a borrower to be considered solvent (1e5 precision)
_liquidationFee	uint256	The fee paid to liquidators give as a % of the repayment (1e5 precision)
_maturityDate	uint256	The maturityDate date of the Pair
_penaltyRate	uint256	The interest rate after maturity date
_isBorrowerWhitelistActive	bool	Enables borrower whitelist
_isLenderWhitelistActive	bool	Enables lender whitelist

initialize

```
function initialize(string _name, address[] _approvedBorrowers, address[] _approvedLenders, by
```

The `initialize` function is called immediately after deployment

This function can only be called by the deployer

Param	Type	Description
_name	string	The name of the contract
_approvedBorrowers	address[]	An array of approved borrower addresses
_approvedLenders	address[]	An array of approved lender addresses
_rateInitCallData	bytes	The configuration data for the Rate Calculator contract

_totalAssetAvailable

```
function _totalAssetAvailable(struct VaultAccount _totalAsset, struct VaultAccount _totalBorr
```

The `_totalAssetAvailable` function returns the total balance of Asset Tokens in the contract

Param	Type	Description
_totalAsset	struct VaultAccount	VaultAccount struct which store total amount and shares for assets
_totalBorrow	struct VaultAccount	VaultAccount struct which store total amount and shares for borrows

Return	Type	Description
[0]	uint256	The balance of Asset Tokens held by contract

_isSolvent

```
function _isSolvent(address _borrower, uint256 _exchangeRate) internal view returns (bool)
```

The `_isSolvent` function determines if a given borrower is solvent given an exchange rate

Param	Type	Description
_borrower	address	The borrower address to check
_exchangeRate	uint256	The exchange rate, i.e. the amount of collateral to buy 1e1 asset

Return	Type	Description
[0]	bool	Whether borrower is solvent

_isPastMaturity

```
function _isPastMaturity() internal view returns (bool)
```

The `_isPastMaturity` function determines if the current block timestamp is past the maturityDate date

Return	Type	Description
[0]	bool	Whether or not the debt is past maturity

isSolvent

```
modifier isSolvent(address _borrower)
```

Checks for solvency AFTER executing contract code

Param	Type	Description
_borrower	address	The borrower whose solvency we will check

approvedBorrower

```
modifier approvedBorrower()
```

Checks if msg.sender is an approved Borrower

approvedLender

```
modifier approvedLender(address _receiver)
```

Checks if msg.sender and _receiver are both an approved Lender

Param	Type	Description
_receiver	address	An additional receiver address to check

isNotPastMaturity

```
modifier isNotPastMaturity()
```

Ensure function is not called when passed maturity

AddInterest

```
event AddInterest(uint256 _interestEarned, uint256 _rate, uint256 _deltaTime, uint256 _feesAm
```

The `AddInterest` event is emitted when interest is accrued by borrowers

Param	Type	Description
_interestEarned	uint256	The total interest accrued by al borrowers
_rate	uint256	The interest rate used to calculate accrued interest
_deltaTime	uint256	The time elapsed since last interest accrual
_feesAmount	uint256	The amount of fees paid to protocol
_feesShare	uint256	The amount of shares distributed to protocol

UpdateRate

```
event UpdateRate(uint256 _ratePerSec, uint256 _deltaTime, uint256 _utilizationRate, uint256 _
```

The `UpdateRate` event is emitted when the interest rate is updated

Param	Type	Description
_ratePerSec	uint256	The old interest rate (per second)
_deltaTime	uint256	The time elapsed since last update
_utilizationRate	uint256	The utilization of assets in the Pair
_newRatePerSec	uint256	The new interest rate (per second)

addInterest

```
function addInterest() external returns (uint256 _interestEarned, uint256 _feesAmount, uint25
```

The `addInterest` function is a public implementation of _addInterest and allows 3rd parties to trigger interest accrual

Return	Type	Description
_interestEarned	uint256	The amount of interest accrued by all borrowers
_feesAmount	uint256	
_feesShare	uint256	
_newRate	uint64	

_addInterest

```
function _addInterest() internal returns (uint256 _interestEarned, uint256 _feesAmount, uint2
```

The `_addInterest` function is invoked prior to every external function and is used to accrue interest and update interest rate

Can only called once per block

Return	Type	Description
_interestEarned	uint256	The amount of interest accrued by all borrowers
_feesAmount	uint256	
_feesShare	uint256	
_newRate	uint64	

UpdateExchangeRate

```
event UpdateExchangeRate(uint256 _rate)
```

The `UpdateExchangeRate` event is emitted when the Collateral:Asset exchange rate is updated

Param	Type	Description
_rate	uint256	The new rate given as the amount of Collateral Token to buy 1e18 Asset Token

updateExchangeRate

```
function updateExchangeRate() external returns (uint256 _exchangeRate)
```

The `updateExchangeRate` function is the external implementation of _updateExchangeRate.

This function is invoked at most once per block as these queries can be expensive

Return	Type	Description
_exchangeRate	uint256	The new exchange rate

_updateExchangeRate

```
function _updateExchangeRate() internal returns (uint256 _exchangeRate)
```

The `_updateExchangeRate` function retrieves the latest exchange rate. i.e how much collateral to buy 1e18 asset.

This function is invoked at most once per block as these queries can be expensive

Return	Type	Description
_exchangeRate	uint256	The new exchange rate

_deposit

```
function _deposit(struct VaultAccount _totalAsset, uint128 _amount, uint128 _shares, address _
```

The `_deposit` function is the internal implementation for lending assets

Caller must invoke `ERC20.approve` on the Asset Token contract prior to calling function

Param	Type	Description
_totalAsset	struct VaultAccount	An in memory VaultAccount struct representing the total amounts and shares for the Asset Token
_amount	uint128	The amount of Asset Token to be transferred
_shares	uint128	The amount of Asset Shares (fTokens) to be minted
_receiver	address	The address to receive the Asset Shares (fTokens)

deposit

```
function deposit(uint256 _amount, address _receiver) external returns (uint256 _sharesReceive
```

The `deposit` function allows a user to Lend Assets by specifying the amount of Asset Tokens to lend

Caller must invoke `ERC20.approve` on the Asset Token contract prior to calling function

Param	Type	Description
_amount	uint256	The amount of Asset Token to transfer to Pair
_receiver	address	The address to receive the Asset Shares (fTokens)

Return	Type	Description
_sharesReceived	uint256	The number of fTokens receive for the deposit

mint

```
function mint(uint256 _shares, address _receiver) external returns (uint256 _amountReceived)
```

The `mint` function allows a user to Lend assets by specifying the number of Assets Shares (fTokens) to mint

Caller must invoke `ERC20.approve` on the Asset Token contract prior to calling function

Param	Type	Description
_shares	uint256	The number of Asset Shares (fTokens) that a user wants to mint
_receiver	address	The address to receive the Asset Shares (fTokens)

Return	Type	Description
_amountReceived	uint256	The amount of Asset Tokens transferred to the Pair

_redeem

```
function _redeem(struct VaultAccount _totalAsset, uint128 _amountToReturn, uint128 _shares, a
```

The `_redeem` function is an internal implementation which allows a Lender to pull their Asset Tokens out of the Pair

Caller must invoke `ERC20.approve` on the Asset Token contract prior to calling function

Param	Type	Description
_totalAsset	struct VaultAccount	An in-memory VaultAccount struct which holds the total amount of Asset Tokens and th total number of Asset Shares (fTokens)
_amountToReturn	uint128	The number of Asset Tokens tc return
_shares	uint128	The number of Asset Shares (fTokens) to burn
_receiver	address	The address to which the Asse Tokens will be transferred
_owner	address	The owner of the Asset Shares (fTokens)

redeem

```
function redeem(uint256 _shares, address _receiver, address _owner) external returns (uint256
```

The `redeem` function allows the caller to redeem their Asset Shares for Asset Tokens

Param	Type	Description
_shares	uint256	The number of Asset Shares (fTokens) to burn for Asset Tokens
_receiver	address	The address to which the Asse Tokens will be transferred
_owner	address	The owner of the Asset Shares (fTokens)

Return	Type	Description
_amountToReturn	uint256	The amount of Asset Tokens to be transferred

withdraw

```
function withdraw(uint256 _amount, address _receiver, address _owner) external returns (uint25
```

The `withdraw` function allows a user to redeem their Asset Shares for a specified amount of Asset Tokens

Param	Type	Description
_amount	uint256	The amount of Asset Tokens to be transferred in exchange for burning Asset Shares
_receiver	address	The address to which the Asse Tokens will be transferred
_owner	address	The owner of the Asset Shares (fTokens)

Return	Type	Description
_shares	uint256	The number of Asset Shares (fTokens) burned

BorrowAsset

```
event BorrowAsset(address _borrower, address _receiver, uint256 _borrowAmount, uint256 _share
```

The `BorrowAsset` event is emitted when a borrower increases their position

Param	Type	Description
_borrower	address	The borrower whose account was debited
_receiver	address	The address to which the Asse Tokens were transferred
_borrowAmount	uint256	The amount of Asset Tokens transferred
_sharesAdded	uint256	The number of Borrow Shares the borrower was debited

_borrowAsset

```
function _borrowAsset(uint128 _borrowAmount, address _receiver) internal returns (uint256 _sha
```

The `_borrowAsset` function is the internal implementation for borrowing assets

Param	Type	Description
_borrowAmount	uint128	The amount of the Asset Toker to borrow
_receiver	address	The address to receive the Asset Tokens

Return	Type	Description
_sharesAdded	uint256	The amount of borrow shares the msg.sender will be debited

borrowAsset

```
function borrowAsset(uint256 _borrowAmount, uint256 _collateralAmount, address _receiver) exte
```

The `borrowAsset` function allows a user to open/increase a borrow position

Borrower must call `ERC20.approve` on the Collateral Token contract if applicable

Param	Type	Description
_borrowAmount	uint256	The amount of Asset Token to borrow
_collateralAmount	uint256	The amount of Collateral Toker to transfer to Pair
_receiver	address	The address which will receive the Asset Tokens

Return	Type	Description
_shares	uint256	The number of borrow Shares the msg.sender will be debited

_addCollateral

```
function _addCollateral(address _sender, uint256 _collateralAmount, address _borrower) interna
```

The `_addCollateral` function is an internal implementation for adding collateral to a borrowers position

Param	Type	Description
_sender	address	The source of funds for the nev collateral
_collateralAmount	uint256	The amount of Collateral Toker to be transferred
_borrower	address	The borrower account for whicl the collateral should be credite

addCollateral

```
function addCollateral(uint256 _collateralAmount, address _borrower) external
```

The `addCollateral` function allows the caller to add Collateral Token to a borrowers position

msg.sender must call ERC20.approve() on the Collateral Token contract prior to invocation

Param	Type	Description
_collateralAmount	uint256	The amount of Collateral Toker to be added to borrower's position
_borrower	address	The account to be credited

RemoveCollateral

```
event RemoveCollateral(address _sender, uint256 _collateralAmount, address _receiver, address
```

The `RemoveCollateral` event is emitted when collateral is removed from a borrower's position

Param	Type	Description
_sender	address	The account from which funds are transferred
_collateralAmount	uint256	The amount of Collateral Token to be transferred
_receiver	address	The address to which Collatera Tokens will be transferred
_borrower	address	

_removeCollateral

```
function _removeCollateral(uint256 _collateralAmount, address _receiver, address _borrower) i
```

The `_removeCollateral` function is the internal implementation for removing collateral from a borrower's position

Param	Type	Description
_collateralAmount	uint256	The amount of Collateral Token to remove from the borrower's position
_receiver	address	The address to receive the Collateral Token transferred
_borrower	address	The borrower whose account will be debited the Collateral amount

removeCollateral

```
function removeCollateral(uint256 _collateralAmount, address _receiver) external
```

The `removeCollateral` function is used to remove collateral from msg.sender's borrow position

msg.sender must be solvent after invocation or transaction will revert

Param	Type	Description
_collateralAmount	uint256	The amount of Collateral Toker to transfer
_receiver	address	The address to receive the transferred funds

RepayAsset

```
event RepayAsset(address _sender, address _borrower, uint256 _amountToRepay, uint256 _shares)
```

The `RepayAsset` event is emitted whenever a debt position is repaid

Param	Type	Description
_sender	address	The msg.sender of the transaction
_borrower	address	The borrower whose account will be credited
_amountToRepay	uint256	The amount of Asset token to b transferred
_shares	uint256	The amount of Borrow Shares which will be debited from the borrower after repayment

_repayAsset

```
function _repayAsset(struct VaultAccount _totalBorrow, uint128 _amountToRepay, uint128 _share
```

The `_repayAsset` function is the internal implementation for repaying a borrow position

The payer must have called ERC20.approve() on the Asset Token contract prior to invocation

Param	Type	Description
_totalBorrow	struct VaultAccount	An in memory copy of the totalBorrow VaultAccount struc
_amountToRepay	uint128	The amount of Asset Token to transfer
_shares	uint128	The number of Borrow Shares the sender is repaying
_payer	address	The address from which funds will be transferred
_borrower	address	The borrower account which w be credited

repayAsset

```
function repayAsset(uint256 _shares, address _borrower) external returns (uint256 _amountToRe
```

The `repayAsset` function allows the caller to pay down the debt for a given borrower.

Caller must first invoke `ERC20.approve()` for the Asset Token contract

Param	Type	Description
_shares	uint256	The number of Borrow Shares which will be repaid by the call
_borrower	address	The account for which the debt will be reduced

Return	Type	Description
_amountToRepay	uint256	The amount of Asset Tokens which were transferred in orde to repay the Borrow Shares

Liquidate

```
event Liquidate(address _borrower, uint256 _collateralForLiquidator, uint256 _shares)
```

The `Liquidate` event is emitted when a liquidation occurs

Param	Type	Description
_borrower	address	The borrower account for whic the liquidation occured
_collateralForLiquidator	uint256	The amount of Collateral Toke transferred to the liquidator
_shares	uint256	The number of Borrow Shares the liquidator repaid on behalf the borrower

liquidate

```
function liquidate(uint256 _shares, address _borrower) external returns (uint256 _collateralF
```

The `liquidate` function allows a third party to repay a borrower's debt if they have become insolvent

Caller must invoke `ERC20.approve` *on the Asset Token contract prior to calling* `Liquidate()`

Param	Type	Description
_shares	uint256	The number of Borrow Shares repaid by the liquidator
_borrower	address	The account for which the repayment is credited and from whom collateral will be taken

Return	Type	Description
_collateralForLiquidator	uint256	The amount of Collateral Toke transferred to the liquidator

LeveragedPosition

```
event LeveragedPosition(address _borrower, address _swapperAddress, uint256 _borrowAmount, ui
```

The `LeveragedPosition` event is emitted when a borrower takes out a new leveraged position

Param	Type	Description
_borrower	address	The account for which the debt is debited
_swapperAddress	address	The address of the swapper which conforms the FraxSwap interface
_borrowAmount	uint256	The amount of Asset Token to be borrowed to be borrowed
_borrowShares	uint256	The number of Borrow Shares the borrower is credited
_initialCollateralAmount	uint256	The amount of initial Collateral Tokens supplied by the borrow
_amountCollateralOut	uint256	The amount of Collateral Toker which was received for the Asset Tokens

leveragedPosition

```
function leveragedPosition(address _swapperAddress, uint256 _borrowAmount, uint256 _initialCo
```

The `leveragedPosition` function allows a user to enter a leveraged borrow position with minimal upfront Collateral

Caller must invoke `ERC20.approve()` on the Collateral Token contract prior to calling function

Param	Type	Description
_swapperAddress	address	The address of the whitelisted swapper to use to swap borrowed Asset Tokens for Collateral Tokens
_borrowAmount	uint256	The amount of Asset Tokens borrowed
_initialCollateralAmount	uint256	The initial amount of Collateral Tokens supplied by the borrow
_amountCollateralOutMin	uint256	The minimum amount of Collateral Tokens to be receive in exchange for the borrowed Asset Tokens
_path	address[]	An array containing the addresses of ERC20 tokens to swap. Adheres to UniV2 style path params.

Return	Type	Description
_totalCollateralBalance	uint256	The total amount of Collateral Tokens added to a users account (initial + swap)

RepayAssetWithCollateral

```
event RepayAssetWithCollateral(address _borrower, address _swapperAddress, uint256 _collatera
```

The `RepayAssetWithCollateral` event is emitted whenever `repayAssetWithCollateral()` is invoked

Param	Type	Description
_borrower	address	The borrower account for whicl the repayment is taking place
_swapperAddress	address	The address of the whitelisted swapper to use for token swap
_collateralToSwap	uint256	The amount of Collateral Toker to swap and use for repayment
_amountAssetOut	uint256	The amount of Asset Token which was repaid
_sharesRepaid	uint256	The number of Borrow Shares which were repaid

repayAssetWithCollateral

```
function repayAssetWithCollateral(address _swapperAddress, uint256 _collateralToSwap, uint256
```

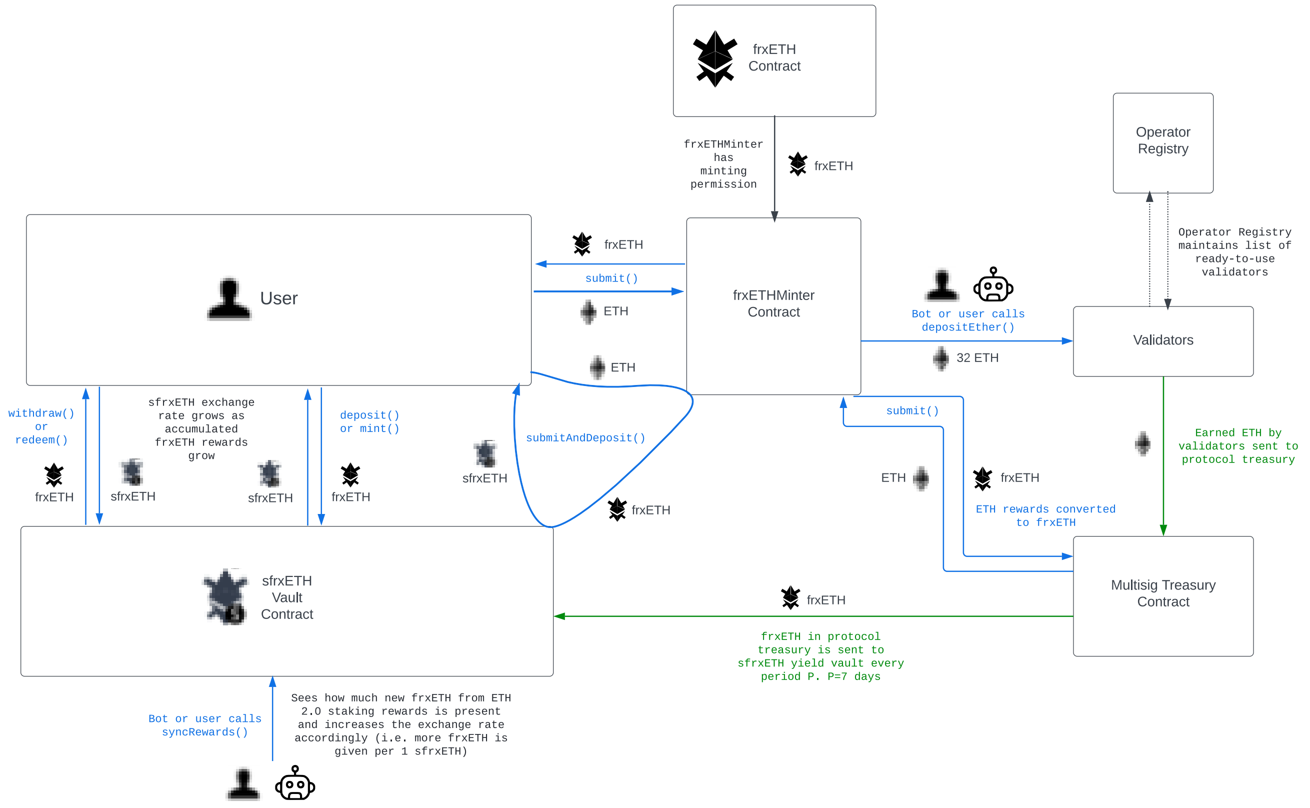
The `repayAssetWithCollateral` function allows a borrower to repay their debt using existing collateral in contract

Param	Type	Description
_swapperAddress	address	The address of the whitelisted swapper to use for token swap
_collateralToSwap	uint256	The amount of Collateral Toker to swap for Asset Tokens
_amountAssetOutMin	uint256	The minimum amount of Asset Tokens to receive during the swap
_path	address[]	An array containing the addresses of ERC20 tokens to swap. Adheres to UniV2 style path params.

Return	Type	Description
_amountAssetOut	uint256	The amount of Asset Tokens received for the Collateral Tokens, the amount the borrowers account was credite

Frax Ether

Overview


frxETH Contract
frxETHMinter has minting permission
frxETH
Operator Registry
Operator Registry maintains list of ready-to-use validators
User
frxETH
submit()
ETH
frxETHMinter Contract
Bot or user calls depositEther()
32 ETH
Validators
ETH
submitAndDeposit()
withdraw() or redeem()
sfrxETH exchange rate grows as accumulated frxETH rewards grow
deposit() or mint()
frxETH
sfrxETH
sfrxETH
frxETH
sfrxETH
frxETH
submit()
ETH
frxETH
ETH rewards converted to frxETH
Earned ETH by validators sent to protocol treasury
sfrxETH Vault Contract
frxETH
frxETH in protocol treasury is sent to sfrxETH yield vault every period P. P=7 days
Multisig Treasury Contract
Bot or user calls syncRewards()
Sees how much new frxETH from ETH 2.0 staking rewards is present and increases the exchange rate accordingly (i.e. more frxETH is given per 1 sfrxETH)

Flowchart

Frax Ether is a liquid ETH staking derivative designed to uniquely leverage the Frax Finance ecosystem to maximize staking yield and smoothen the Ethereum staking process for a simplified, secure, and DeFi-native way to earn interest on ETH.

The Frax Ether system comprises three primary components, Frax Ether (frxETH), Staked Frax Ether (sfrxETH), and the Frax ETH Minter:

1. **frxETH** acts as a stablecoin loosely pegged to ETH, leveraging Frax's winning playbook on stablecoins and onboarding ETH into the Frax ecosystem.
2. **sfrxETH** is the version of frxETH which accrues staking yield. All profit generated from Frax Ether validators is distributed to sfrxETH holders. By exchanging frxETH for sfrxETH, one become's eligible for staking yield, which is redeemed upon converting sfrxETH back to frxETH.
3. **Frax ETH Minter** (**frxETHMinter**) allows the exchange of ETH for frxETH, bringing ETH into the Frax ecosystem, spinning up new validator nodes when able, and minting new frxETH equal to the amount of ETH sent.

Liquid Staking

Solo ETH staking requires the technical knowledge and initial setup associated with running a validator node, and also that deposits be made 32 ETH at a time. By opting to use a liquid ETH staking derivative instead of staking ETH in another form, staking yield can be accrued much more simply, abstracting the need to run validators, allowing yield to be earned on any amount of ETH, allowing withdrawals at any time and of any size, and allowing far greater composability throughout DeFi.

frxETH flywheel interview with Jack Corddry

frxETH and sfrxETH

ETH in the Frax ecosystem comes in two forms, frxETH (Frax Ether), and sfrxETH (Staked Frax Ether).

frxETH

frxETH acts as a stablecoin loosely pegged to ETH, so that 1 frxETH always represents 1 ETH and the amount of frxETH in circulation matches the amount of ETH in the Frax ETH system. When ETH is sent to the frxETHMinter, an equivalent amount of frxETH is minted. Holding frxETH on its own is not eligible for staking yield and should be thought of as analogous as holding ETH.

sfrxETH

sfrxETH is a ERC-4626 vault designed to accrue the staking yield of the Frax ETH validators. At any time, frxETH can be exchanged for sfrxETH by depositing it into the sfrxETH vault, which allows users to earn staking yield on their frxETH. Over time, as validators accrue staking yield, an equivalent amount of frxETH is minted and added to the vault, allowing users to redeem their sfrxETH for a greater amount of frxETH than they deposited.

The exhange rate of frxETH per sfrxETH increases over time as staking rewards are added to the vault. By holding sfrxETH you hold a % claim on an increasing amount of the vault's frxETH, splitting staking rewards up among sfrxETH holders proportional to their share of the total sfrxETH. This is similar to other autocompounding tokens like Aave's aUSDC and Compound's cUSDC.

Technical Specifications

frxETH

frxETH shares much of its code with both the Frax and FPI stablecoins, and implements the ERC-2612 standard, allowing spender approvals to be made via ERC-712 signatures passed to the permit() function.

sfrxETH

sfrxETH is a ERC-4626 compliant vault. sfrxETH is obtained by first approving the sfrxETH contract as a frxETH spender, and then calling mint() (mints a specific number of sfrxETH) or deposit() (deposits a specific amount of frxETH). The approval step and the minting step can be combined with depositWithSignature() or mintWithSignature(), removing the need for two seperate transactions.

As validators generate staking yield, an equivalent amount of frxETH is minted and sent to the sfrxETH contract. This means that once rewards are synced, one's sfrxETH may be redeemed for a greater amount of frxETH than it took to mint.

To prevent malicious users from stealing a validator yield distribution to the vault, reward distributions are smoothed over time cycles. Whenever syncRewards() is called on the sfrxETH contract, any additional frxETH added to the contract over the contract's internal balance is queued to be distributed linearly over the remainder of a cycle window.

sfrxETH is also ERC-2612 compliant, allowing the use of signature permits.

frxETHMinter

The frxETHMinter mints frxETH when it receives ETH either through the submit() or receive() function. Whenever a submission pushes the minter balance over 32 ETH, the contract pops a validator's deposit credentials off of a stack and passes the 32 eth deposit along with the credentials to the ETH 2.0 deposit contract, automatically spinning up a new validator.

As needed, new credentials are added to the stack to ensure that there are always validators ready to take deposits. If at any time the contract runs out of validators, frxETH will continue to be minted as normal (unless paused) but no new validators will be spun up until more are added to the stack.

The withdrawal credential is shared by all the validators on the stack, meaning all validators share the same withdrawal address. This address is set to the Frax Multisig at launch, so that withdrawals may be safely handled once live.

In addition, when adding validators it is necessary to pass the DepositDataRoot as provided when generating the deposit data, this is to provide redundancy in ensuring a validator with misinputted parameters will not be accepted when ETH is deposited.

The protocol may set a ratio of funds to withold when ETH is submitted. These funds are not counted when gathering 32 ETH deposits to spin up validators and are instead used to market make across DeFi, ensuring liquid markets for frxETH.

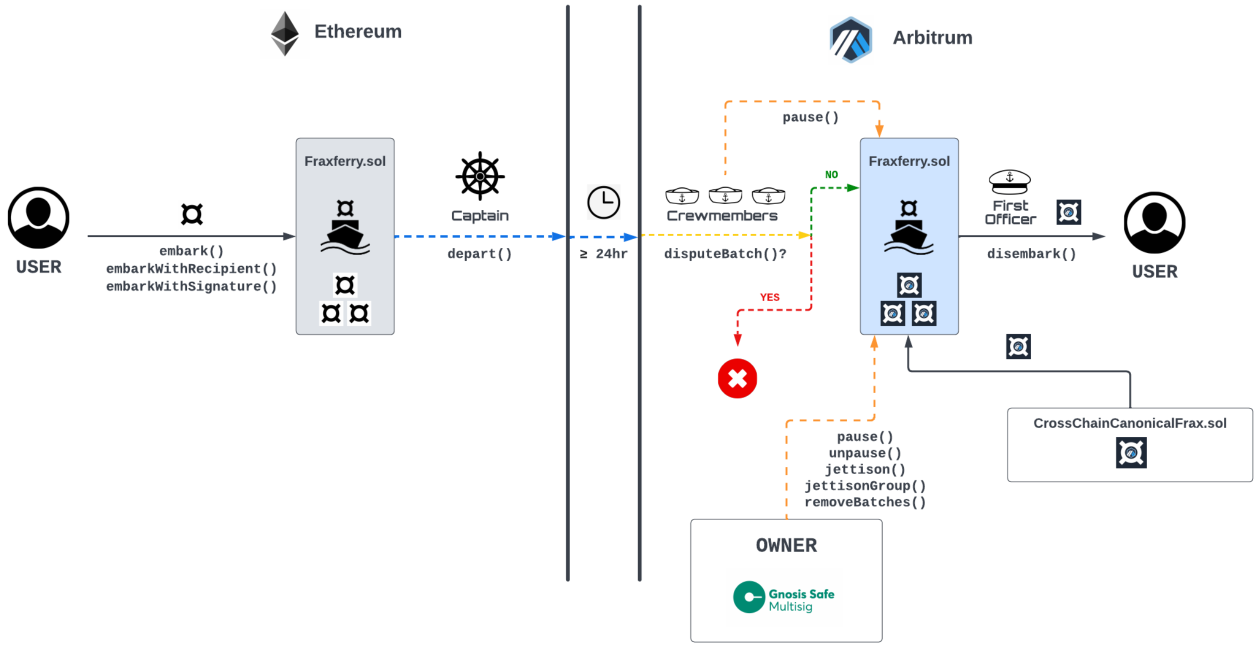
Frax in-House Validators

(Documentation coming soon)

Fraxferry

Overview

Ferry that can be used to ship tokens between chains



Overview

Summary

A slower, simpler, but more secure method of bridging tokens.

Motivation

- Too many bridge hacks from bugs, team rugs, anon devs, etc.
- Risks of infinite mints.
- Some chains have slow bridges (Arbitrum, Optimism, etc).

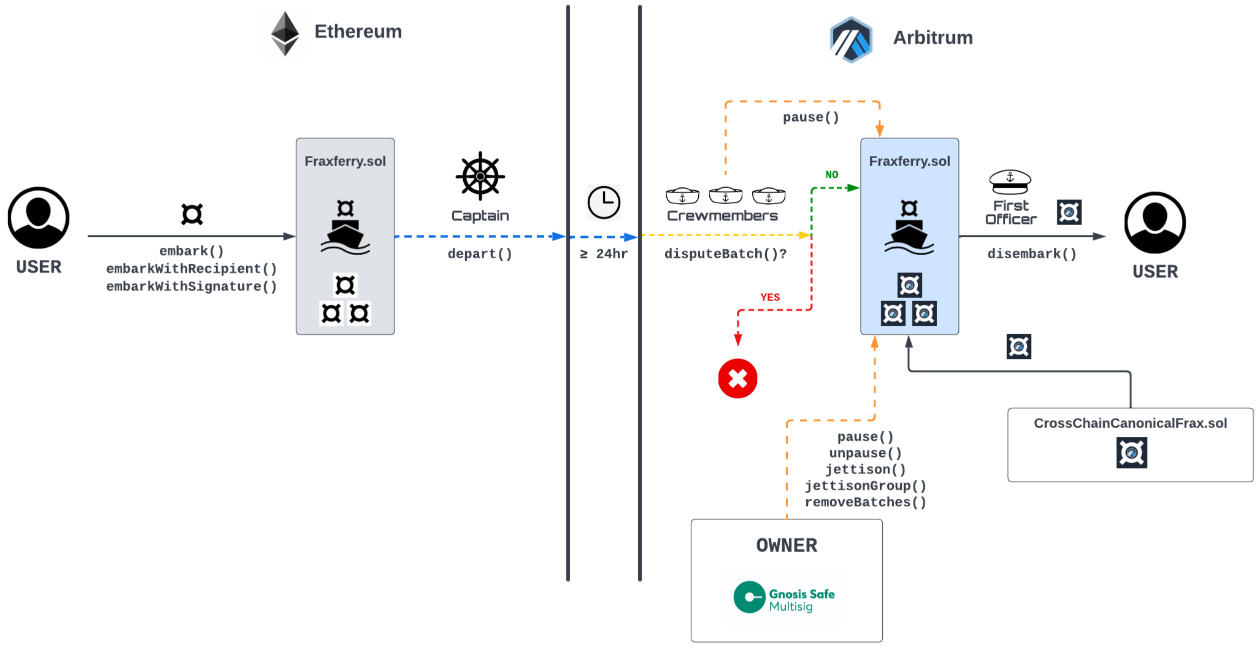
Benefits

- Risk is capped by token amounts in bridge contracts. No risk of infinite mints.
- Slower transactions give more time for bad batches to be caught and stopped, assuming they are not cancelled automatically by bots.
- Crewmembers can pause contracts so any issues can be investigated.

Risks

- Captain is tricked into proposing a batch with a false hash AND all crewmembers bots are offline/censured/compromised and no one disputes the proposal.
- Reorgs on the source chain. Avoided, by only returning the transactions on the source chain that are at least one hour old.
- Rollbacks of optimistic rollups. Avoided by running a node.
- Operators do not have enough time to pause the chain after a fake proposal. Avoided by requiring a minimal amount of time between sending the proposal and executing it.
- Centralization

Details



Overview

Process

1. User sends tokens to the contract. This transaction is stored in the contract. `embark()`, `embarkWithRecipient()`, or `embarkWithSignature()`.
2. Captain queries the source chain for transactions to ship.
3. Captain sends batch (start, end, hash) to start the trip. `depart()`
4. Wait at least 24 hrs.
5. Crewmembers check the batch and can dispute it if it is invalid. `disputeBatch()` or `do nothing`.
6. Non disputed batches can be executed by the first officer by providing the transactions as calldata. User receives their tokens on the other chain. `disembark()`
7. Hash of the transactions must be equal to the hash in the batch.
8. In case there was a fraudulent transaction (a hacker for example), the owner can cancel a single transaction, such that it will not be executed. `jettison()`, `jettisonGroup()`, `removeBatches()`.
9. The owner can manually manage the tokens in the contract and must make sure it has enough funds.

Frax Base Pool (FRAXBP)

Overview

A FRAX based stableswap pool to pair liquidity with to receive additional FXS incentives

Summary

A slower, simpler, but more secure method of bridging tokens.

Motivation

- Too many bridge hacks from bugs, team rugs, anon devs, etc.
- Risks of infinite mints.
- Some chains have slow bridges (Arbitrum, Optimism, etc).

Benefits

- Risk is capped by token amounts in bridge contracts. No risk of infinite mints.
- Slower transactions give more time for bad batches to be caught and stopped, assuming they are not cancelled automatically by bots.
- Crewmembers can pause contracts so any issues can be investigated.

Risks

- Captain is tricked into proposing a batch with a false hash AND all crewmembers bots are offline/censured/compromised and no one disputes the proposal.
- Reorgs on the source chain. Avoided, by only returning the transactions on the source chain that are at least one hour old.
- Rollbacks of optimistic rollups. Avoided by running a node.
- Operators do not have enough time to pause the chain after a fake proposal. Avoided by requiring a minimal amount of time between sending the proposal and executing it.
- Centralization

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.

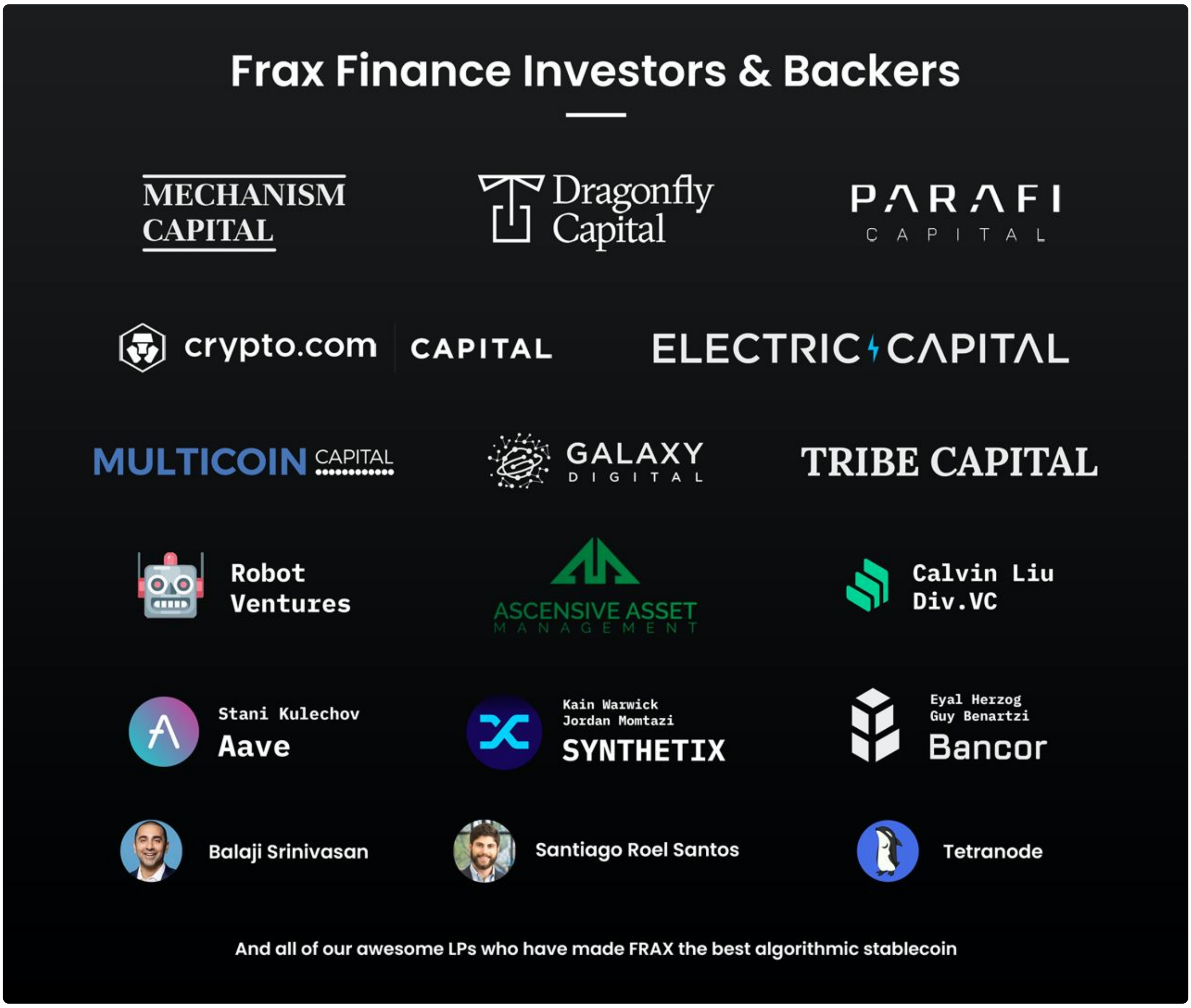
12% – Accredited Private Investors – 2% unlocked at launch, 5% vested over the first 6 months, 5% vested over 1 year with a 6 month cliff
The first round in Frax was done in August of 2020 with a small allocation that was sold out in under 2 hours. This allocation will have a small amount of their tokens,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



Frax Finance Investors & Backers

MECHANISM CAPITAL

Dragonfly Capital

PARAFI CAPITAL

crypto.com CAPITAL

ELECTRIC CAPITAL

MULTICOIN CAPITAL

GALAXY DIGITAL

TRIBE CAPITAL

Robot Ventures

ASCENSIVE ASSET MANAGEMENT

Calvin Liu Div.VC

Stani Kulechov Aave

Kain Warwick Jordan Momtazi SYNTHETIX

Eyal Herzog Guy Benartzi Bancor

Balaji Srinivasan

Santiago Roel Santos

Tetranode

And all of our awesome LPs who have made FRAX the best algorithmic stablecoin

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…

Saddle D4 (alUSD/FEI/FRAX/LUSD) staking

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

Fraxswap / FPI

Fraxswap

Guide : How to use AMM/TWAMM on FraxSwap.
Medium

FPI

Guide : How to Mint/Redeem FPI Stablecoin on FraxSwap.
Medium

Matic / Polygon

Avalanche

How to get Avalanche (AVAX)

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

BSC

Fantom

Harmony

Moonriver

Smart Contracts

All Contracts

1) https://github.com/FraxFinance/frax-solidity/blob/master/src/types/constants.ts
2) Search for "CONTRACT_ADDRESSES"

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
Evmos	0xE03494D0033687543a80c9B1ca7D6237F2EA8BD8
Fantom	0xdc301622e621166BD8E82f2cA0A26c13Ad0BE355
Harmony	0xFa7191D292d5633f702B0bd7E3E3BcCC0e633200
Moonbeam	0x322E86852e492a7Ee17f28a78c663da38FB33bfb
Moonriver	0x1A93B23281CC1CDE4C4741353F3064709A16197d
Optimism	0x2E3D870790dC77A83DD1d18184Acc7439A53f475
Polygon	0x45c32fA6DF82ead1e2EF74d17b76547EDdFaFF89
Solana	FR87nWEUxVgerFGhZM8Y4AggKGLnaXswr1Pd8wZ4k cp

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()` .

```
uint256 public global_collateral_ratio
```

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

```
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.

```
uint256 public minting_fee
```

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

```
address public DEFAULT_ADMIN_ADDRESS
```

Set in the constructor. Used in `AccessControl` .

```
bytes32 public constant COLLATERAL_RATIO_PAUSER
```

A constant used in the pausing of the collateral ratio.

```
bool public collateral_ratio_paused
```

Whether or not the collateral ratio is paused.

View Functions

oracle_price

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

Get the FRAX or FXS price, in USD.

frax_price

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

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

fxs_price

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

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

frax_info

```
frax_info() public view returns (uint256, uint256, uint256, uint256, uint256, uint256, uint25
```

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

```
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

```
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

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

Public implementation of internal _mint().

pool_burn_from

```
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	0x9d2F299715D94d8A7E6F5eaa8E654E8c74a 8A7
Aurora	0xBb8831701E68B99616bF940b7DafBeb4CDb e0b
Avalanche	0x214DB107654fF987AD859F34125307783fC8 387
Boba	0xae8871A949F255B12704A98c00C2293354a3 013
BSC	0xe48A3d7d0Bc88d552f730B62c006bC925ead eE
Ethereum	0x3432B6A60D23Ca0dFCa7761B7ab56459D9 64D0
Evmos	0xd8176865DD0D672c6Ab4A427572f80A72b4 A9C
Fantom	0x7d016eec9c25232b01F23EF992D98ca97fc2A 5a
Harmony	0x0767D8E1b05eFA8d6A301a65b324B6b66A1 C14c
Moonbeam	0x2CC0A9D8047A5011dEfe85328a6f26968C8a A1C
Moonriver	0x6f1D1Ee50846Fcbc3de91723E61cb68CFa6D E98
Optimism	0x67CCEA5bb16181E7b4109c9c2143c24a1c22 5Be
Polygon	0x1a3acf6D19267E2d3e7f898f42803e90C9219 2
Solana	6LX8BhMQ4Sy2otmAWj7Y5sKd9YTVVUgfMsBz 6B9W7ct

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

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

Change the address of the price oracle.

setFRAXAddress

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

Set the address of the FRAX contract.

setFXSMinDAO

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

Set minimum FXS required to join DAO groups.

mint

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

Mint new FXS tokens.

pool_mint

```
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

```
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

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

Transfer FXS tokens.

transferFrom

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

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

Public Functions

getCurrentVotes

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

Gets the current votes balance for `account`.

getPriorVotes

```
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

```
_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

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

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.

```
bytes32 private constant BUYBACK_PAUSER
```

`AccessControl` role for the buyback pauser.

```
bool mintPaused = false
```

Whether or not minting is paused.

```
bool redeemPaused = false
```

Whether or not redeem is paused.

```
bool buyBackPaused = false
```

Whether or not buyback is paused.

View Functions

unclaimedFXS

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

Return the total amount of unclaimed FXS.

unclaimedCollateral

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

Return the total amount of unclaimed collateral.

collatDollarBalance

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

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

availableExcessCollatDV

```
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

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

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

Public Functions

mint1t1FRAX

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

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

mintFractionalFRAX

```
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

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

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

redeem1t1FRAX

```
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

```
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

```
toggleRedeeming() external onlyRedeemPauser
```

Toggle the ability to redeem.

toggleBuyBack

```
toggleBuyBack() external onlyBuyBackPauser
```

Toggle the ability to buyback.

setPoolCeiling

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

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

setOracle

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

Set the `oracle_address`.

setCollateralAdd

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

Set the `collateral_address`.

addOwner

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

Add an address to the array of owners.

removeOwner

```
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

```
FRAXStablecoin private FRAX
```

Instance of the FRAX contract.

```
ERC20 public rewardsToken
```

Instance for the reward token.

```
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

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

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

stakingDecimals

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

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

rewardsFor

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

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

lastTimeRewardApplicable

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

Used internally to keep track of `rewardPerTokenStored` .

rewardPerToken

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

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

earned

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

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

getRewardForDuration

```
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.se
```

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(add
```

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 onlyOwner
```

Set the maximum multiplier for locked stakes.

setLockedStakeTimeForMaxMultiplier

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

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. This bounty is a "no questions asked" policy for disclosures and/or immediate return of funds after any incident.

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.

This bounty applies to all smart contracts deployed by the Frax Deployer addresses including Fraxswap AMM, Fraxlend, and frxETH.

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

Fraxswap

Smart contract addresses for Fraxswap V1 & V2

Fraxswap V2

V2 includes a new feature, allowing for different LP Fees per pool

Ethereum

Fraxswap Factory: `0x43eC799eAdd63848443E2347C49f5f52e8Fe0F6f`

Fraxswap Router: `0xC14d550632db8592D1243Edc8B95b0Ad06703867`

Fraxswap Multihop Router: `0x25e9acA5951262241290841b6f863d59D37DC4f0`

Arbitrum

Fraxswap Factory: `0x8374A74A728f06bEa6B7259C68AA7BBB732bfeaD`

Fraxswap Router: `0xCAAaB0A72f781B92bA63Af27477aA46aB8F653E7`

Avalanche

Fraxswap Factory: `0xf77ca9B635898980fb219b4F4605C50e4ba58afF`

Fraxswap Router: `0x5977b16AA9aBC4D1281058C73B789C65Bf9ab3d3`

Binance Smart Chain

Fraxswap Factory: `0xf89e6CA06121B6d4370f4B196Ae458e8b969A011`

Fraxswap Router: `0x67F755137E0AE2a2aa0323c047715Bf6523116E5`

DogeChain

Fraxswap Factory: `0x67b7DA7c0564c6aC080f0A6D9fB4675e52E6bF1d`

Fraxswap Router: `0x0f6A5c5F341791e897eB1FB8fE8B4e30EC4F9bDf`

Note it is currently labeled as Fraxswap V1 on Dogechain Explorer

Fantom

Fraxswap Factory: `0xDc745E09fC459aDC295E2e7baACe881354dB7F64`

Fraxswap Router: `0x7D21C651Dd333306B35F2FeAC2a19FA1e1241545`

Harmony

Fraxswap Factory:

Fraxswap Router:

Moonbeam

Fraxswap Factory: `0x51f9DBEd76f5Dcf209817f641b549aa82F35D23F`

Fraxswap Router: `0xd95fe880d7717f7f20981FE6e41A2315f3EFeAb5`

MoonRiver

Fraxswap Factory: `0x7FB05Ca29DAc7F5690E9b5AE0aF0415D579D7CD3`

Fraxswap Router: `0xD8FC27ec222E8d5172CD63aC453C6Dfb7467a3C7`

Optimism

Fraxswap Factory: `0x67a1412d2D6CbF211bb71F8e851b4393b491B10f`

Fraxswap Router: `0xB9A55F455e46e8D717eEA5E47D2c449416A0437F`

Polygon

Fraxswap Factory: `0x54F454D747e037Da288dB568D4121117EAb34e79`

Fraxswap Router: `0xE52D0337904D4D0519EF7487e707268E1DB6495F`

Fraxswap V1

DEPRECATED

Ethereum

Fraxswap Factory: `0xB076b06F669e682609fb4a8C6646D2619717Be4b`

Fraxswap Router: `0x1C6cA5DEe97C8C368Ca559892CCce2454c8C35C7`

Arbitrum

Fraxswap Factory: `0x5Ca135cB8527d76e932f34B5145575F9d8cbE08E`

Fraxswap Router: `0xc2544A32872A91F4A553b404C6950e89De901fdb`

Avalanche

Fraxswap Factory: `0x5Ca135cB8527d76e932f34B5145575F9d8cbE08E`

Fraxswap Router: `0xc2544A32872A91F4A553b404C6950e89De901fdb`

Binance Smart Chain

Fraxswap Factory: `0xa007a9716dba05289df85A90d0Fd9D39BEE808dE`

Fraxswap Router: `0x0AE84c1A6E142Ed90f8A35a7E7B216CB25469E37`

Fantom

Fraxswap Factory: `0xF55C563148cA0c0F1626834ec1B8651844D76792`

Fraxswap Router: `0xa007a9716dba05289df85A90d0Fd9D39BEE808dE`

Harmony

Fraxswap Factory: `0x5Ca135cB8527d76e932f34B5145575F9d8cbE08E`

Fraxswap Router: `0xc2544A32872A91F4A553b404C6950e89De901fdb`

Moonbeam

Fraxswap Factory: `0x5Ca135cB8527d76e932f34B5145575F9d8cbE08E`

Fraxswap Router: `0xc2544A32872A91F4A553b404C6950e89De901fdb`

MoonRiver

Fraxswap Factory: `0x5Ca135cB8527d76e932f34B5145575F9d8cbE08E`

Fraxswap Router: `0xc2544A32872A91F4A553b404C6950e89De901fdb`

Optimism

Fraxswap Factory: `0xBe90FD3CDdaf0D3B8576ffb5E51aDbfD304d0437`

Fraxswap Router: `0xffE66A866B249f5d7C97b4a4c84742A393bC9354`

Polygon

Fraxswap Factory: `0xc2544A32872A91F4A553b404C6950e89De901fdb`

Fraxswap Router: `0x9bc2152fD37b196C0Ff3C16f5533767c9A983971`

API

Combined Data

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

Staking / APRs

https://api.frax.finance/pools

Other

Audits

2020

Nov 2020 - Certik

2021

June 2021 - Trail of Bits
Dec 2021 - Trail of Bits

2022

April 2022 - Shipyard [Ongoing]
May 2022 - Trail of Bits
September 2022 - Code4rena

Media Kit / Logos

FRAX

FRAX_Logos.zip 3MB
Binary

FXS

FXS_Logos.zip 1MB
Binary

FPI

FPI_Logos.zip 94KB
Binary

FPIS

FPIS_Logos.zip 72KB
Binary