Glittr Bitcoin Metaprotocol

Periwinkle Doerfler, Ph.D., CTO of Glittr, Inc.

July 20, 2024

This is V0.1 of the whitepaper for the Glittr Metaprotocol and explanations of the DeFi Enablement it provides. As Glittr approaches a full-scale, main-net launch the remaining sections will be documented here and all code will be made publicly available. Subsequent updates to the protocol, as facilitated and voted upon by the community, will also be reflected in future versions of this document which should be seen as living. This document focuses on accessibility and readability, and thus reserves technical details such as byte encoding and decoding for developer docs.

V0.1 provides an overview of the fundamentals of the protocol and a comprehensive explanation of Pool Equity Glyphs and how they can be used to create self-enforcing, permissionless AMM pools.

Abstract

Glittr is a new Bitcoin metaprotocol for DeFi Enablement. Glittr provides the building blocks for the vast majority of DeFi applications that people wish to build: AMMs, lending, staking, swapping, and so on. The protocol has decentralized governance, is flexible and adaptable, and we anticipate onboarding new use cases and integrations as the community develops.

> We believe in *transparency, decentralization, and trust minimization.* We want you to trust that you don't have to trust us.

Glittr is a UTXO-based, Bitcoin final protocol that takes inspiration from the Runes Protocol[2]. Glittr transactions live in OP_RETURN and are mined before they are validated. Glittr provides significant novel DeFi functionality by way of configurable validation arguments directly on Glyphs (tokens). Glyphs can be etched using a variety of templates whose configurable arguments are designed to facilitate different DeFi use cases. Glyphs of various templates can be combined to create complex DeFi applications.

1 Introduction

1.1 The Challenge of DeFi on Bitcoin

The comprehensive DeFi ecosystem that exists on Ethereum does not have a corollary on Bitcoin. There are many reasons for this, including throughput, but the biggest contributor is that the two blockchains (and their respective protocols) simply have significantly different designs.

Where the Ethereum protocol could be seen as a collection of bank accounts, Bitcoin should be seen as a series of IOUs (read: UTXOs). These singular, persistent entities inherent in Ethereum make smart contracts as we know them possible. Bitcoin, by comparison, operates using a chain-of-custody model. While a user may choose to re-use a persistent public key[1], UTXOs must still be fully spent, OP codes are limited in complexity, and transactions cannot be triggered from within the chain itself as you would see with an EVM-based smart contract.

Smart contracts are freestanding pieces of code that can be audited and interacted with directly. The code itself is verifiable and the contract - as an entity - is persistent and immutable. EVM functions like AWS in some senses, allowing arbitrarily expensive code to be processed as long as one pays the compute costs (gas). While validating transactions is a necessary step in Bitcoin mining, validating Bitcoin transactions is very cheap ($O(1)$). The computational expense of Bitcoin is derived

[1]Though this is generally considered bad form, and undermines Bitcoin's fundamental privacy and anonymity properties

from the mining itself, whereas Ethereum's proof-of-stake model allows significant computational resources to process smart contracts in a distributed fashion. With Bitcoin you pay for block space, not computation.

"Smart Contract on Bitcoin" solutions which seek to replicate the functionalities and structure of smart contracts on EVM are in fundamental tension with the structure of Bitcoin. The introduction of a VM that can process arbitrary smart contracts necessarily removes validation and enforcement of such contracts from the Bitcoin blockchain. Because the computation cannot be deferred to the validation consensus of the mining network these solutions instead rely on centralized, off-chain computation with the results of the contract then written to a Bitcoin transaction.

The problem with computation leaving the chain - even if it is multi-party or provable, as some solutions propose - is that trustless computation does not imply trustless transactions. [2] For a contract to be enforced, the enforcing party must have access to the underlying assets. On Ethereum this is acceptable because the contract *itself* is the enforcer, but with Bitcoin there will always be a third party (i.e. a company) that facilitates the bridge to the VM. At the very least a VM-based solution introduces an oracle problem. For more complex contracts not achievable through DLCs, the third party may become custodial.

1.2 Glittr Protocol

Instead of attempting to create "Smart Contracts on Bitcoin", Glittr attempts to reframe - do we need *Smart Contracts*^{TM} or do we simply need a system that *enables DeFi*? Glittr enables DeFi by creating a structure that works within the UTXO-based framework of Bitcoin, allowing all validation and enforcement to stay directly on Bitcoin Layer 1. Glittr relies on decentralization and consensus as its enforcement mechanism as opposed to provable or self-executing code. After all, no amount of cryptography can save you from a 51% attack.

Glittr-based tokens - Glyphs - are etched using a series of templates with unique, configurable arguments with each template facilitating a specific use case. A given DeFi application will need to etch one or more Glyphs to facilitate its desired functionality. While Glyphs are fungible tokens, their primary purpose is not speculation but utility. The arguments in the etching of a Glyph are *validation instructions* for the subset of the Bitcoin nodes that recognize the metaprotocol. Glittr transactions are mined before they are validated, meaning that everything is always Bitcoin final. Glittr is a reactive system; users issue transactions that are consistent with the constraints of a given Glyph, and validators simply confirm that the transactions are, in fact, consistent. When this is the case, these transactions become state.[3]

Additional functionality can be added to the protocol in the future through updates to, and the adoption of, templates.

2 Glyphs

Glyphs are etched using one of the recognized templates, which provide specific validation instructions for transactions. A Glyph issued without a template has all the same properties and behaviors as a Rune. Glittr will launch initially with three templates: Pool Equity Glyphs (representing fractional equity in a liquidity pool), Wrapped Runes, and Collateralized Glyphs (anything minted against collateral, e.g. a stablecoin).

2.1 Why templates?

Glittr relies on templates for Glyphs to maintain the financial viability of decentralized *post-mining* validation, consensus, and therefore enforcement. Templates are designed such that validating any transaction for any Glyph can be done in constant time.

Allowing for arbitrary programmability - e.g. Turing Completeness - creates the need to either incentivize validators proportionately to their expended compute (read: pay gas) or to remove computation from the decentralized parts of the protocol, e.g. to an off-chain VM, as discussed in Section

[2]If the purpose of a smart contract execution is simply to cause computation to occur and the resulting proof to be documented forever, some VM-based solutions which are entirely robust. This presents exciting possibilities in the field of verifiable MPC, for example.

[3]This is similar to the post-mine validation we see with Runes.

1.1. While Turing Completeness is often touted as a desirable standard, it is very rarely necessary. The vast majority of DeFi functionality - lending, staking, swapping, AMMs, etc. - can be built within $O(1)$-validatable UTXOs using Glittr. [4]

Functionality that cannot be achieved with the existing templates can be added in the future through additional templates so long as the validator network agrees that they are adequately efficient to incorporate.

3 Glittr Transactions

Glittr Protocol transactions are stored in Bitcoin transaction outputs through the use of OP_RETURN.

There are three major categories of Glittr Transaction:

- Etching: the creation of a new Glyph, specifying the template and validation instructions
- Simple Transfers: transactions that only move a Glyph from one UTXO to another
- Validated Transactions: these include minting, burning, and other transactions that are specific to a template, e.g. swaps for AMM pools

The first argument in a Glittr Transaction is TxType, which has five potential values: ETCH, MINT, BURN, TRANSFER, and SPECIAL. Etching, minting, and burning are discussed below. Simple transfers operate identically to analogous Runestones[5], and are signed. SPECIAL is a flag for a template-specific transaction type, e.g. a Swap for a Pool Equity Glyph. A further argument - SpecialTx - specifies the exact type of transaction.

3.1 Glyph Etching

The major functionality of a Glyph is defined in the initial etching. If an etching specifies a ticker that ticker must be unused. Etching transactions must specify a template and provide [valid] values for all non-optional arguments.[6] Etchings have additional validation criteria which vary by template.

3.2 Validated Transactions

Applications on Glittr are self-enforcing as a result of validated transactions. After etching, subsequent transactions must be compliant with the rules defined in the Glyph's etching. These transactions generally include a proof that a required condition has been met and a statement of the expected outcome. The original etcher of the Glyph does not need to sign for these transactions and does not have the power to further regulate them. [7]

Transactions which pass validation become state, and those which do not validate correctly are considered cenotaphs. The consequences of invalid transactions differ between templates. Some templates have a strict burn, as with Runes, where others simply disregard the transaction. Some templates allow the etcher to specify the consequences for an invalid transaction. The number and types of validated transactions vary by template, but include minting, burning, and special transaction types.

3.2.1 Minting and Burning

Once a Glyph is etched it can be minted dynamically pursuant to its rules. Most templates do not have supply caps or time constraints for minting, nor do they require the signature of the original etching party (or any other centralized authority) for creation of additional supply.

For example, a user can mint a Glittr stablecoin so long as they provide proof that they have appropriately collateralized relevant assets (pursuant to the rules laid out in the etching of the Glyph). As opposed to an EVM protocol like MakerDAO[3] - which can be trusted to accurately mint Dai

[4] A notable exception would be any 'contracts' that require a random number generator. Glittr transactions are deterministic.

[5] This is the name given to the content of a Runes transaction

[6] Glyphs can be etched without a template, in which case they default back to the rules of Rune etchings.

[7] It is possible to create templates which allow for centralized control to be created and enforced, but the initial ones do not.

(and sign for the transfer) when collateralization conditions are met because it is an auditable piece of self-executing code - a Glittr-based stablecoin can be trusted simply because there is no centralized party to trust. The mint itself is unsigned, with the user instead 'manifesting' their new stablecoin bag into existence and the validators simply determining whether to greenlight the transaction.

Burn transactions function similarly, with the user signing for the burn of Glyphs and stating the intended result - e.g. the return of collateral. So long as they have the right keys for the UTXOs, the burn is pursuant to the Glyph's etching, and the requested return amount is correct, the burn transaction will then serve as a signable UTXO for the returned collateral. There is no need for a further signed transaction to return those Glyphs from a vault or liquidity pool.

3.2.2 General Validation Principles

Validation instructions vary by transaction type and template, but all Glittr transactions require the following to be validated:

- Reference to initial Glyph etching
- Reference to a previous transaction indicating current state of the Glyph
- Demonstration of compliance with Glyph rules, e.g. collateral deposited
- Indication of expected outcome, e.g. amount of Glyph to be minted
- User signature (but not app signature)

4 Pool Equity Glyphs

Pool Equity Glyphs [PEGs] represent a fractional share of the assets in a liquidity pool, such as an AMM pool. These pools operate similarly to those overseen by Uniswap[1] using a Constant Product Model[4].

> ***Disclaimer:*** *While the actual implementation of PEGs will use a constant product model to disincentivize over-large swaps dramatically swaying pool ratios the numbers provided in these simplistic examples do not. These are for illustrative purposes - to make the math easy to understand they assume that transactions use the pool ratio state from before the transaction.*

PEGs have five transaction types:

- Etching
- Providing Liquidity (minting)
- Retrieving Liquidity (burning)
- Swaps [8]
- Retrieving Yield [9]

4.1 Etching

The etching of a PEG functionally *creates* an AMM pool. The pool exists on a single public key specified in the initial etching. The etching transaction is signed by the pool key.

Thereafter, pools are custodial but permissionless; the holder of the private key for the pool does not need to sign for any transactions after etching, and is not capable of moving the pool's assets through use of that key. [10]

[8] These transactions do not actually move any amount of the PEG, but the transactions are still attributed to its etching

[9] A PEG may or may not have a unique Tx type for this, dependent on its fee and yield structure.

[10] We suggest that developers create a new public key which will hold *only* this asset pair and serve no other purpose.

Glittr Transaction Example 1: PEG Etching

```
TxType: Etch,
Template: PEG,
PEGTicker: [optional],
Asset1: [optional] A•GLYPH,
Asset2: [optional] ANOTHER•GLYPH,
Asset1EtchRef: Block:Tx,
Asset2EtchRef: Block:Tx,
PoolPubKey: 0x.... ,
Asset1UTXOList: {Block:Tx, Block:Tx},
Asset1Volume: 100,
Asset2UTXOList: {Block:Tx, Block:Tx},
Asset2Volume: 100,
InitialRatio: 1,
InitialPEG: 1000,
IPEGPubKey: [optional],
LPEvenRatio: True,
LiveTime: [optional] BlockHeight,
FeeAssetType: SWAP_IN,
FeePCT: 0.03,
YieldType: PEG
```

FeeAssetType	Direction	Fee Asset	Input	Output
SWAP_IN	1 ->2	Glyph 1	103 $G1	100 $G2
	2 ->1	Glyph 2	103 $G2	100 $G1
SWAP_OUT	1 ->2	Glyph 2	100 $G1	97 $G2
	2 ->1	Glyph 1	100 $G2	97 $G1
ASSET1	1 ->2	Glyph 1	103 $G1	100 $G2
	2 ->1	Glyph 1	100 $G2	97 $G1
ASSET2	1 ->2	Glyph 2	100 $G1	97 $G2
	2 ->1	Glyph 2	103 $G2	100 $G1

Table 1: Fee Structures

The etching may optionally specify a future block height at which the pool will become live for swaps, allowing a time period for LPs to contribute at a fixed ratio. If not, the pool will be live immediately. LP transactions are still possible after the pool is live.

The etching of a PEG specifies the following properties of the pool:

- The pair of assets the pool will contain
- The initial ratio of those assets (i.e. market rate)
- The public key of the pool
- The fee structure for swaps
- The yield structure for LPs

For a PEG etching transaction to be valid a few conditions must be met: 1) all non-optional arguments are provided and valid (e.g. UTXO pointers are correct), 2) the Tx is signed by the pool, 3) the initial ratio accurately reflects the ratio of assets held by the pool, 4) all optional arguments are valid if they are provided (e.g. LiveTime block height can't be in the past).

Example 1 shows a PEG etching transaction. The arguments are explained in detail below.

4.2 Etching Arguments

The first two arguments tell the indexer what code to use to validate the transaction. Because PEGs are a utility token, a ticker is optional and likely unnecessary.

4.2.1 Asset Pair

For the self-enforcing and permissionless aspects of PEGs to work correctly, all assets contained in a Glittr-backed pool must be Glyphs. This said, templates will be available for the creation of Glyphs which represent other Bitcoin-native assets, including Bitcoin itself. These templates will allow application developers to create wraps in whatever way they feel best serves their use case and customers, including immediate support for several industry-standard wrapping mechanisms as well as a proprietary wrapping structure invented by Glittr. [11] Glittr will launch with support for Wrapped Runes, with support for other token standards like Ordinals and BRC-20 to follow.

The next arguments specify what the assets in the pool will be. As not all Glyphs need tickers, specifying the tickers for the assets is optional. The indexer uses arguments **Asset1** and **Asset2** to reference the etching of the Glyph assets. For the purposes of the pool, these Glyphs are treated simply as fungible tokens even if they are templated Glyphs with additional validation instructions. (i.e. a Swap transaction is a transaction on the PEG, with the transactions for the underlying Glyph assets being simple transfers)

4.2.2 Initial Pool State

The next set of arguments specify the state of the pool at its inception. Before etching a PEG, the developer should consolidate UTXOs for the asset pair onto a single address, which will become the pool's address as specified by **PoolPubKey**. [12]

Asset1UTXOList and **Asset2UTXOList** then list the UTXOs that provide the initial assets held by the pool. **Asset1Volume** and **Asset2Volume** provide the initial volumes and **InitialRatio** provides the initial market rate between the two assets, expressed as a ratio of Asset1:Asset2.

InitialRatio must equate to the actual ratio of the two initial volumes, and the stated volumes must equal the sum of the listed UTXOs. The etching then specifies the **InitialPEG** volume, i.e. how many units of the PEG to generate for the initial liquidity. This number is somewhat arbitrary, as the supply of PEG will scale proportionately with the liquidity in the pool and PEGs are subdividable. Optionally, the etching may specify an address to which the initial PEG should be allocated. If no key is specified the PEG is owned by the pool directly. [13]

4.2.3 Providing Liquidity

The next set of arguments specify conditions for providing liquidity. **LPEvenRatio** is a Boolean argument indicating whether liquidity providers must always deposit both pool assets in equal amounts (where 'equal' is determined by the current trading ratio of the pool). [14] **LiveTime** is an optional argument specifying a future block at which the pool will become active for swaps. If LiveTime is not specified, the pool is active immediately. The benefit of specifying a future LiveTime is that it allows LPs to contribute at a preset initial market rate, avoiding potential losses due to volatility at the opening of a pool. It is strongly recommended to set LPEvenRatio to True when setting a LiveTime.

4.2.4 Fees and Yield

The remaining arguments specify fee structure for swaps and yield structure for LPs. **FeeAssetType** has five possible values: SWAP_IN, SWAP_OUT, ASSET1, ASSET2, and OTHER.

[11] Details on the Wrapped Runes template, and the available implementations for Wrapped Bitcoin, will follow in subsequent updates to this whitepaper.

[12] A pool cannot be created empty, though the starting balances can be arbitrarily low. This is necessary to avoid a divide-by-zero error in generating proportional PEGs.

[13] Because pools are permissionless, it is recommended that any PEGs which a developer intends to keep as a reserve are held by a separate key as retrieving liquidity does require a signature. Developers may also choose to treat initial liquidity as buffer, and leave the PEG in the pool with no intention of retrieval.

[14] Setting LPEvenRatio to true is likely to ensure a more stable pool, where trading price can only be influenced by asymmetric swaps. Alternately, setting it to false may allow for a pool to reach equilibrium with other market forces more quickly without relying on swap arbitrage as a forcing function.

SWAP_IN and **SWAP_OUT** specify that fees are to be paid in the incoming or outgoing swap asset, respectively. **ASSET1** and **ASSET2** indicate that fees are to be paid in a specific one of the pool's paired assets, regardless of the directionality of the swap. [15]

When FeeAssetType is any of SWAP_IN, SWAP_OUT, ASSET1, or ASSET2 the following argument is **FeePCT**, indicating the percentage of the transaction that needs to be paid as a fee. In Example 1 using SWAP_IN, FeePCT = 0.03, and a ratio of 1, this would mean that a user should deposit 103 A•GLYPH to receive 100 ANOTHER•GLYPH in return. All four possibilities are explained in Table 1.

The final argument in this section is YieldType, this specifies how LPs are to be paid their yield. When using SWAP_IN, SWAP_OUT, ASSET1, or ASSET2 the recommended solution is **YieldType = PEG** This simply means that LPs receive their yields based on the appreciation of the value of their PEGs. When fees are paid in one of the two pool assets the total liquidity in the pool changes without the volume of the PEG changing, thus each unit of the PEG corresponds to a greater amount of liquidity. LPs can retrieve their yields by burning PEG and retrieving liquidity. When YieldType is set to PEG, no further arguments are needed.

4.2.5 FeeAssetType = OTHER

Assigning FeeAssetType to **OTHER** allows a developer to specify a third Glyph, likely their own native token, in which the fees for a swap should be paid. Using OTHER has clear financial benefits for developers but does create complexities with fee structures, yield payments, and the potential need for an oracle to determine the value of the fee asset relative to the trading assets. [16]

When a third Glyph is used for fees it is necessary to pay LP yield in that asset as the PEG is not inherently appreciating. Additional configurations are needed to outline how those yields are distributed and how often. These will be presented in V0.2 of this paper.

4.3 Providing Liquidity (Minting PEG)

LP transactions are PEG minting transactions. In a single transaction the user transfers Glyphs to the pool's public key - these are signed 'simple transfers' - and generates a proportionate amount of the pool's PEG. The mint is unsigned; so long as the liquidity is provided and PEG minted in adherence to its etching, the transaction is validated and indexed and becomes state.

Example 2 shows a PEG minting Rosetta, and the arguments are explained below.

4.3.1 Functional Arguments

The first several arguments assist the indexer in correctly validating the transaction. Although **PoolPubKey** is specified in the initial etching it is restated in the minting Rosetta as a safety precaution, because these transactions will sign a transfer of Glyphs to that address. If the key specified in the transaction does not match the key in the etching, the transaction is invalid.

4.3.2 Pool Ratio

The next set of arguments provide the current state of the pool including the total volume of both assets and the ratio of the two assets. These are stated explicitly (as opposed to inferred) to minimize computation and avoid the risk of recursion. **PoolStateRef** is a reference to the last transaction which impacted the state of the pool. The ratio argument is optional as it can be inferred from the stated volumes. The PoolRatio argument is optional as it can be inferred from the asset volumes, but is helpful in visualizing this example. [17]

[15]Using ASSET1 and ASSET2 will inherently cause the market rate to bias towards the fee baring asset, but may be desirable for a pool that anticipates asymmetric swap demand. For example, a pairing of anything with wrapped Bitcoin may expect more swaps *into Bitcoin*, and thus mandate that Bitcoin is also the fee asset to help maintain the pool's balance.

[16]The need for an oracle can be mitigated by referencing another Glittr PEG's state, assuming the developer holds pools that trade each of the two assets against their native Glyph, and that those pools use one of the other four FeeAssetType arguments.

[17]Pool ratio is not validated against the asset volumes because rounding discrepancies between users/applications and canonical Glittr indexing code may lead to unnecessary cenotaphs.

Glittr Transaction Example 2: PEG Minting (LP transaction)

```
TxType: Mint,
Template: PEG,
PEGTicker: [optional],
PEGEtchRef: Block:Tx,
PoolPubKey: 0x...,
Asset1Vol: 300,
Asset2Vol: 500,
PoolRatio: [optional] 0.6,
PoolStateRef: Block:Tx,
Asset1UTXOs: {Block:Tx, Block:Tx, ...},
Asset1InAmt: 30,
Asset2UTXOs: {Block:Tx, Block:Tx, ...},
Asset2InAmt: 50,
ReturnAmt1: 4.6,
ReturnAmt2: 0,
ReturnAddress: [optional] 0x...,
PEGTotalVol: 5000,
PEGVolRef: Block:Tx,
PEGMint: 500,
MintPubKey: [optional] 0x...
```

4.3.3 Liquidity Provision

For the sake of ease, we will assume that the LP has consolidated UTXOs holding the respective assets to be provided to the pool. [18] **Asset1UTXOs** and **Asset2UTXOs** provide lists of UTXOs containing each asset.

The sum of the Glyphs in the UTXOs must be *at least* equal the the respective arguments **Asset1InAmt** and **Asset2InAmt**, but do not need to equal this figure exactly. Glyphs are consumed from the referenced UTXOs greedily (in the order they are listed) until the InAmt is reached, and the rest are returned to **ReturnAddress**. If ReturnAddress is not specified, the public key initiating the mint is used. When Glyphs are returned in this manner the *mint Rosetta itself becomes the spendable UTXO for that change*, thus, it is necessary for the user to also specify their expected return amounts in **ReturnAmt1** and **ReturnAmt2** to avoid future recursion. If the sum of InAmt and ReturnAmt for a given asset do not equal the sum of the provided UTXOs, the transaction is invalid.

If the PEG was etched with LPEvenRatio = True, then the ratio of Asset1InAmt to Asset2InAmt must be equal to the pool ratio. If LPEvenRatio = False, the user is free to deposit the two Glyphs in whatever quantity they desire.

4.3.4 PEG Mint

The remaining arguments specify the amount of PEG to be minted and where it should go. **PEGTotalVol** specifies the current total volume of PEG the pool has. **PEGVolRef** points to the last transaction that impacted the balance of PEG, which may or may not be distinct from the transaction specified in PoolStateRef. This is because a Swap transaction will impact the balance of assets in a pool, and therefore the trading ratio, but will not impact the supply of PEG. **PEGMint** specifies the amount of PEG the user expects to receive. The ratio of PEGMint to PEGTotalVol must be equal to the ratio of the deposited assets to the total liquidity in the pool, or the transaction is invalid.

In Example 2, the user deposits 30 Asset1 and 50 Asset2 to a pool with 300 and 500 of these assets, respectively. The user is depositing an amount equivalent to 10% of the existing liquidity, and thus should mint PEG equivalent to 10% of the existing supply. In this case, they mint 500 PEG where the latent supply is 5000.

[18] It is of course possible to perform this consolidation in the same transaction, but requires additional logic to add multiple signatures.

4.4 Retrieving Liquidity (Burning PEG)

Retrieving liquidity is a PEG burning transaction. In a single transaction the user retrieves Glyphs from the pool's public key and burns PEG. The burn is signed by the user, and the return of Glyphs from the pool is unsigned. So long as the requested return is consistent with the state of the pool and the PEG etching, the transaction is validated and the burn and return become state. See Example 3, and the explanations below for the arguments and their validation conditions.

4.4.1 Functional and Pool State Args

These are functionally the same as in minting transactions: the first several arguments exist to facilitate the indexer in validating the transaction correctly. The next group of arguments provide the current state of the pool. The total volume for each asset is provided along with a reference to the UTXO that last impacted these numbers (i.e. any pool transaction). The total volume of PEG is provided with a reference to the last transaction which impacted it, namely the last mint or burn.

4.4.2 Burn Arguments

The user then specifies **BurnAmt**, and provides **PEGUTXOs** accounting for at least that much PEG. These UTXOs may be from PEG mints or transfer transactions. (While the primary function of a PEG is not to serve as a fungible token, they are fungible and can be traded and transferred.) If the BurnAmt does not equate to the sum of the PEGUTXOs, the user must specify **PEGReturn** - the amount of PEG that should be returned and not burned. If the sum of BurnAmt and PEGReturn does not equal the sum of PEG accounted for in PEGUTXOs, the transaction is invalid. PEG will be burned greedily from UTXOs in the order listed until BurnAmt is reached. If there is PEGReturn, the user can specify a **PEGReturnAddress** for custody of the remaining PEG. If none is specified, the key that initiated the burn transaction is used.

4.4.3 Return Arguments

Finally, the user specifies the amount of each asset they expect to receive back in **Asset1Return** and **Asset2Return**. The user can optionally specify a **ReturnAddress** for these assets to be delivered to, else the key that initiated the burn transaction is used. Even if a pool has LPEvenRatio = False, liquidity is always returned proportionately to the current pool ratio. If the stated return values are not consistent with the ratio of assets in the pool, the transaction is invalid. If the stated return values are not proportionate to the PEG being burned, the transaction is invalid.

In Example 3, the user is burning 50 PEG out of a total volume of 5000 (inclusive of their as-yet unburned PEG), or 1% of the total supply. The pool contains 300 Asset 1 and 500 Asset 2, so the user is entitled to receive 1% of each of those supplies - 3 and 5 Glyphs, respectively.

4.5 Swaps

Swap transactions are a unique functionality of the PEG Template. Swaps do not actually transact PEG, but they are still attributed to the PEG's etching to maintain consensus about the state of the underlying pool, its asset volumes, and market price.

A swap transaction transfers ownership of one Glyph to the pool and retrieves another. The transfer of assets to the pool is signed, and the transfer of assets out of the pool is not. So long as the swap is consistent with the market ratio and fees are paid appropriately, the transaction is indexed and validated and becomes state.

Example 4 shows what a swap might look like, and the arguments are explained in detail below.

4.5.1 Functional and Pool State Args

The first several arguments facilitate indexing and validation. **TxType** has a flag SPECIAL, which indicates that the transaction is specific to the template. The indexer will then process the **Template** argument, and then **SpecialTx** which specifies the type of special transaction.

The next group of arguments provide the volume of assets and the pool (and therefore implied market ratio), and a reference to the previous state.

Glittr Transaction Example 3: PEG Burning (Liquidity Retrieval)

```
TxType: Burn,
Template: PEG,
PEGEtchRef: Block:Tx,
PoolPubKey: 0x...,
Asset1Vol: 300,
Asset2Vol: 500,
PoolStateRef: Block:Tx,
PEGTotalVol: 5000,
PEGVolRef: Block:Tx,
BurnAmt: 50,
PEGUTXOs: {Block:Tx, Block:Tx,...},
PEGReturn: [optional] 0,
PEGReturnAddress: [optional] 0x...,
Asset1Return: 3,
Asset2Return: 5,
ReturnAddress: [optional] 0x...
```

4.5.2 Input Arguments

The user then specifies **InAsset** and **InVol**. InAsset has only two possible values: Asset1 and Asset2, where these are the assets defined in the PEG etching. **InUTXOs** provide a list of transactions the user will sign over to the pool to provide InAsset. The sum of InAsset in InUTXOs must be at least InVol, but does not need to equal InVol. If they are not equal, the user must specify the expected remainder in **ReturnVol**, and may optionally specify an address in **ReturnAddress**. If no address is specified, the public key that initiated the swap is used. If the sum of InVol and ReturnVol is not equal to the sum of InAsset in InUTXOs, the transaction in invalid.

4.5.3 Fee Arguments

FeeAsset and **FeeAmt** explicitly outline the fees being paid as part of the swap. For a PEG etched using FeeAssetType of SWAP_IN, SWAP_OUT, ASSET1, or ASSET2, the only possible values for FeeAsset are Asset1 and Asset2. The fee asset specified in the swap transaction must be consistent with the directionality of the transaction and the fee structure specified in FeeAssetType in the etching. If the wrong fee asset is used, the transaction is invalid. FeeAmt is expressed in units of FeeAsset.

In Example 4 the user is paying a fee of 1.5 Asset2, which would be consistent with a SWAP_OUT fee structure at a fee rate of 0.03 (3%). [19] The FeeAmt (in units) must be equal to the FeePct as specified in the etching. In this example, 30 Asset1 would correlate to 50 Asset2 based on the ratio in the pool. Since we are using SWAP_OUT at 3%, that is 1.5 Asset2. If FeeAmt is not equal to the correct percentage, the transaction is invalid.

4.5.4 Output Arguments

Finally, the user specifies the amount of Glyphs they expect to receive from the swap in **OutVol**. The user does not need to specify which type of Glyph they expect to receive, as a pool contains only two assets and the 'Out Asset' is inherently whatever InAsset isn't. The user may optionally specify a new address **OutAddress** for the Glyphs to be directed to. If no OutAddress is specified, the key that initiated the swap is used.

In cases where FeeAsset == InAsset, OutVol must equal $InVol/(1 + FeePct) * Ratio$

In cases where FeeAsset == OutAsset, Outvol must equal $(InVol * Ratio) * (1 - FeePct)$

If the value of OutVol in incorrect, the transaction is invalid. [20]

[19] It would also be consistent with an ASSET2 fee structure, but we are assuming that this PEG has SWAP_OUT

[20] Note that 'ratio' as described above will, in practice, refer the post-trade ratio defined by the Constant Product Model, not the pre-trade ratio used in these examples.

Glittr Transaction Example 4: PEG Swap

```
TxType: Special,
Template: PEG,
SpecialTx: Swap,
PEGEtchRef: Block:Tx,
PoolPubKey: 0x...,
Asset1Vol: 3000,
Asset2Vol: 5000,
PoolStateRef: Block:Tx,
InAsset: Asset1,
InVol: 30,
InUTXOs: {Block:Tx, Block:Tx, ...},
ReturnVol: [optional] 0.5,
ReturnAddress: [optional] 0x...,
FeeAsset: Asset2,
FeeAmt: 1.5,
OutVol: 48.5,
OutAddress: [optional] 0x...
```

5 The Role of the Application

Glittr provides the building blocks for DeFi applications on Bitcoin, but the applications themselves are still important. Glittr makes it *possible* to build truly decentralized, trust-minimized DeFi applications on Bitcoin, but the fact that an app is built on Glittr does not *guarantee* these properties.

Developers will need to ensure that their Web2 implementations are secure, and that any additional Web3 components they introduce are robust. Some applications may still wish to leverage DLCs, multisigs, and other Bitcoin primitives in conjunction with Glyphs.

5.1 AMM Pools

While the PEG template provides for a fully functioning AMM without any Web2 UI, in reality we expect that most Glittr-backed AMMs will be administered through an application.

Glittr transactions are mined on Bitcoin directly, and the block time is slow. It is possible to execute multiple Glittr transactions within the same block by referencing the previous transaction with its hash as opposed to Block:Tx reference. [21] For transfers of Glyphs - and indeed most use cases for most templates - this is likely to work fine by itself, as one would need to know of the incoming UTXO to then sign it.

The PEG template and AMM use case present a slightly more complicated situation, because there is a need to reference a previous transaction to which the user is not a party - the last transaction to the pool. While it is possible for users to interact with these pools by issuing transactions directly to the peer-to-peer network, relying on doing so would likely lead to a large number of invalid transactions. As such, we expect that in practice the adminstrators of such pools will provide an interface and API that generates and formats transactions for its users, maintaining an intra-block state, and ensuring that transactions are ordered correctly. Figure 1 provides a visual representation of what this user data flow might look like. This would not in any way change the custody or permission structure of the PEG.

[21]This is similar to how Runes transactions within the same block are referenced.

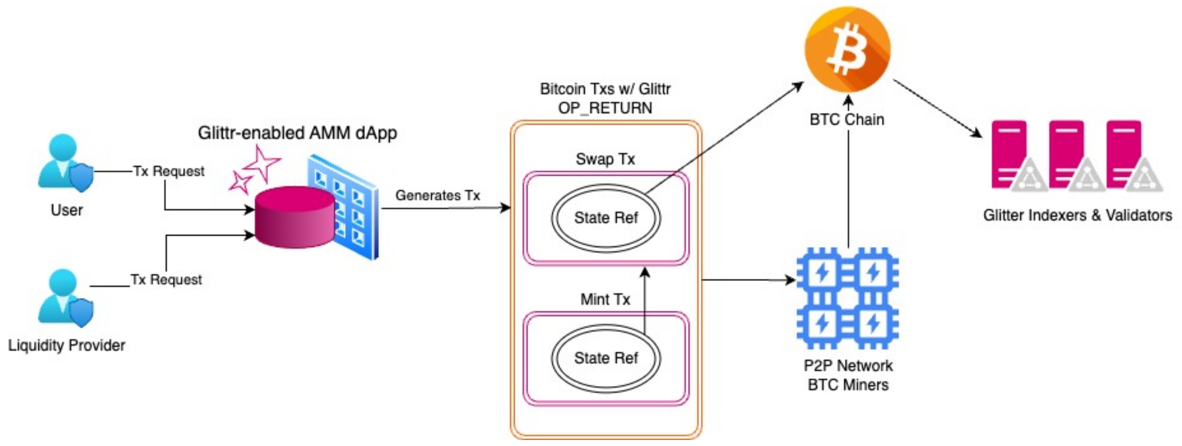

Figure 1: Glittr-enabled AMM dApp Data Flow

6 Future Work

Glittr's governance structure allows for additional templates to be added, existing templates to be altered, integration with other systems, and more. While Glittr as a company intends to release additional features in the short term, we also hope that the community will propose, build, and approve new features as well.

> ***Disclaimer:*** *Unless explicitly stated, the contents of this section are not promises that the Glittr team will develop features. This should be seen as discussion of possibilities. As the Glittr ecosystem evolves, the Glittr team is committed to offering hands-on developer support to onboard new use cases. If you are a developer interested in building on Glittr, please do not hesitate to reach out to us.*

6.1 Template updates

6.1.1 PEG Template

- The PEG template could be updated to allow for different market-making algorithms to be specified, as opposed to requiring all PEGs to utilize a Constant Product Model.

6.2 New Templates

6.3 Integrations

References

[1] Adams, H. Uniswap Whitepaper. Tech. rep., Uniswap, 2020.

[2] Rodarmor, C. Runes Protocol [Whitepaper]. Tech. rep., Ordinals Foundation, 2024.

[3] Team. The Maker Protocol: MakerDAO's Multi-Collateral Dai (MCD) System. Tech. rep., Maker Foundation, 2014.

[4] Zhang, Y., Chen, X., and Park, D. Formal Specification of Constant Product Market Maker Model and Implementation. Tech. rep., Runtime Verification, Inc., 2018.