Dexterity Whitepaper

Jarry Xiao[1], Daniel Gunsberg[2], Robert Levy[3], Michael Setrin[4], Gregory Snow[5]

Abstract

Dexterity is a collection of smart contracts on the Solana blockchain. It provides a framework for trading any instrument with a defined payoff function, including but not limited to options, futures, perpetual swaps, and bonds. Dexterity leverages the on-chain order book pioneered by Project Serum to manage trades and portfolio balances. The framework has a modularized design that creates a hard separation between the mechanical components of trading market infrastructure (e.g. the order book data structure, position tracking, funding distribution) and the application-specific logic that relates to the instruments and exchange operators (e.g. settlement algorithms, margin calculation, liquidation thresholds, fee model). We believe the flexibility this modular design creates will allow for a wide spectrum of on-chain market types to be created and supported by the protocol, making Dexterity a new fundamental building block for decentralized trading applications.

[1]Engineer at Solana Labs: jarry@solana.com
[2]Co-founder at Hxro Foundation: dan.gunsberg@hxro.io
[3]Co-founder at Hxro Foundation: rob.levy@hxro.io
[4]Quantitative Researcher at Jump Crypto: msetrin@jumptrading.com
[5]Principal Engineer at Chicago Trading Company: greg.snow@chicagotrading.com

1 Introduction

1.1 Why Derivatives

Derivatives trading volume is primarily driven by 2 factors:

1. Derivatives allow traders to hedge various risks associated to their existing positions.
2. Derivatives allow for leveraged risk exposure with varying payoff functions (standardized and exotic) and tenors thus allowing traders to both speculate and hedge in a capital efficient manner. This provides incentives for both speculators and market makers to participate.

There have been many past attempts to build on-chain derivatives — on Solana and on other chains — but the team believes that the approach taken for the Dexterity protocol is the most generalized and scalable.

1.2 Limitations of Serum

In 2020, Project Serum made a major advancement in on-chain trading by demonstrating the Solana blockchain's ability to support a functional on-chain central limit order book (CLOB). While early versions of the Serum CLOB supported spot markets, the protocol design made it difficult to support derivatives trading due to the following:

1. Serum couples the order book, matching engine, and asset custody components into a single contract. Forcing derivatives contracts to take the form of fungible SPL tokens creates an unintuitive developer interface (and likely a clunky user experience).
2. Serum requires every order (not just trades) to be 100% collateralized. This has major implications for the capital requirements of both market makers and speculators. Liquidity providers on Serum are required to lock up their funds in order to make markets for tokens. This capital can no longer be used for other investments and thus poses a heavy opportunity cost.
3. Derivatives are created or destroyed when trades occur because they are artificial contracts tied to some underlying asset. This naturally fits well with a callback paradigm when processing trade events. However, the initial Serum implementation did not support the ability to inject custom logic for processing trades.

These early design elements led to the creation of a new version of Serum known as Asset Agnostic Order Book.

The Asset Agnostic Order Book (AAOB) is a program that extracts the data structure for Serum's order book and matching engine into a separate smart contract.

The main benefits of this separation are:

1. The AAOB no longer requires composing programs to represent assets with SPL Tokens.
2. Custom processing of the event queue is performed by an upstream program. This allows custom callback logic for processing trades.

The innovations made in AAOB unlocked the eventual development of Dexterity.

1.3 Dexterity Overview

Dexterity is a collection of smart contracts that allows for the creation and exchange of any instrument with a defined payoff function. This encapsulates all traditional derivative contracts such as futures, options and perps, but also includes fixed income contracts and binary options (prediction markets). It additionally interfaces with both a customizable fee model and a customizable risk engine that handles cross-product margining and liquidation. There is also built-in support for combo (a.k.a. synthetic) products, enabling users to trade contracts with arbitrary leg ratios (e.g. futures spreads). As a result, the protocol opens up the door for liquid, decentralized, and capital-efficient trading.

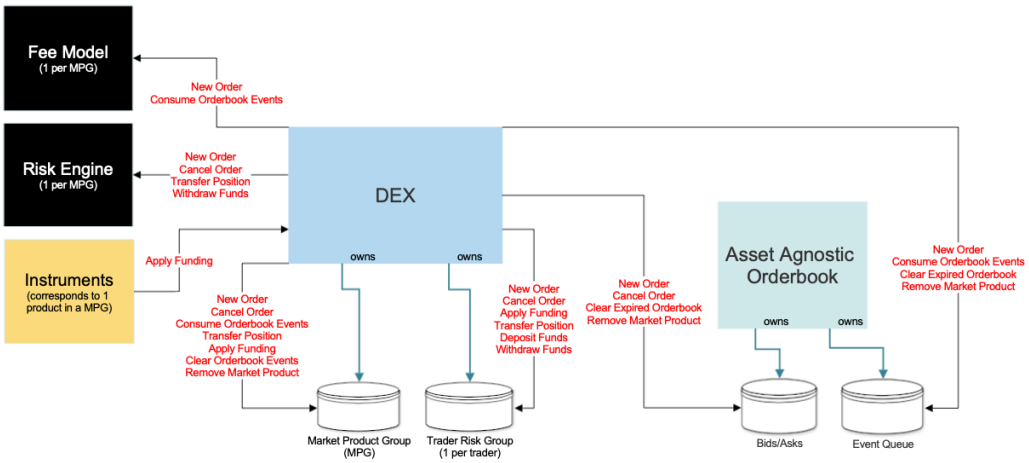

Figure 1: High-level diagram of Dexterity protocol

The Dexterity Protocol relies on 5 major components:

1. **Dex**: The Dex contract handles all of the accounting of orders, trades, deposits, withdrawals, and funding that occur on the exchange. All of the product metadata is stored in one `MarketProductGroup` account. Each trader has an associate `TraderRiskGroup` account that tracks the state of that trader's orders, positions, and deposits in a given `MarketProductGroup`.

2. **AAOB**: The Dex contract interacts with AAOB to keep track of the order book (`bids` and `asks` accounts) as well as trades (`event_queue` account).

3. **Instruments**: An instrument can be any contract that implements an instruction that makes a CPI to the `ApplyProductFunding` instruction in the Dex contract. The team built a reference **Instruments** contract that implements the payoff functions of perpetual and expiring options (a future is just a call option with a strike price of 0). It also implements the payoff function for a simple fixed income contract. For derivative settlement, data is pulled from both Pyth and the Dex order books to determine the appropriate contract payout. An instruction is exposed that performs a cross program invocation (CPI) to the Dex contract to record the funding payment for the given instrument. The `MarketProductGroup` that stores this instrument will be updated

accordingly, and any `TraderRiskGroup` with an active position in the corresponding instrument will have unsettled funds that must be applied before any future position changes.

4. **Fee Model**: The Dex contract provides a specific interface for computing trading fees. A fee program that implements this interface is required for setting up a `MarketProductGroup`. Whenever a trade occurs, the Dex contract will make a CPI to the Fee Model (explicitly configured on the corresponding `MarketProductGroup`), which then extracts a taker and maker fee for each `TraderRiskGroup`, and that fee or rebate is applied to the transaction.

5. **Risk Engine**: The Dex contract also provides a specific interface for risk calculations. Whenever an action is taken that could potentially increase the risk of a trader's position, a CPI is made from the Dex to the Risk Engine (explicitly configured on the corresponding `MarketProductGroup`) to determine whether that action is acceptable. If the action brings a trader below the required account health, the CPI will fail and roll back the entire transaction.

A key design decision was made to separate the mechanical accounting operations in the Dex and AAOB contracts from the business logic in the Instruments, Risk Engine, and Fee Model contracts. A common analogy in DeFi compares the individual components of composable smart contracts to Lego blocks. The goal with this modular design is to allow customizable Legos to be used for any of the business logic components, while keeping the accounting logic fixed. The Dex does not know or care about fee structure, risk calculation, or contract settlement. It receives output data from "API calls" and performs basic numerical operations on those outputs.

2 Design Considerations

2.1 Solana Constraints

Constraints in Solana application development fall into 5 main categories:

1. **On-chain computation**: Given that the Solana blockchain aims to target a 400ms block time, the runtime places a hard cap on the number of operations that can be performed in a single transaction. BPF instructions are assigned a specific number of "compute units" in the Solana runtime, and any transaction that exceeds a certain limit will fail. Currently this cap is set at 200000 units. This limit is expected to increase in future network upgrades.

2. **Rent cost**: Solana transaction fees are cheap, but allocating account data can be expensive. It is important to be cognizant of the cost imposed on a user to allocate on-chain state.

3. **Transaction size**: The size of a Solana transaction is limited to approximately 1.2 KB. The primary consequence of this is that it limits the number of 32 byte public keys (i.e. accounts) with which a transaction can interact. The maximum transaction size will roughly double in future network upgrades.

4. **Runtime memory limits**: In the runtime, Solana programs limit the size of the stack to exactly 4KB. This requires the developer to be diligent about not allocating too many large variables on the stack. Additionally, new accounts that are allocated through CPIs are limited to approximately 10KB in size, which prevents the use of Program Derived Addresses (PDAs) for large state accounts.

5. **BPF instruction set**: Solana programs are compiled to eBPF, which has a limited instruction set (primarily with respect to floating point operations). These limitations need to be considered when thinking about what types of numeric algorithms are feasible.

Protocol and application design considerations need to manage the trade-offs resulting from every decision. In the design of Dexterity, all of these constraints were evaluated when building out account storage, instruction interfaces, and core algorithms.

2.2 Anchor

The Dexterity protocol uses the Anchor framework. The biggest benefit of using Anchor is the auto-generation of an IDL file that is used for building out client code that interacts with the on-chain protocol. The other main benefits of Anchor are:

1. The reduction of boilerplate code required for account parsing and validation
2. Autogenerated Rust interfaces for interacting with program instructions
3. On-chain program verification

2.3 Object Representation

Early on, a decision was made to store all of the exchange products into one large account as opposed to many small accounts. There were a few main considerations contributing to this decision:

- The Solana transaction size makes it advantageous to minimize the number of public keys used in a transaction.
- Many operations (like potential risk engine implementations) require data from multiple products. By putting every product into a `MarketProductGroup` and putting all of a trader's positions into a `TraderRiskGroup`, these calculations could look through all products and positions with minimal processing from input data.
- Solana clients need to make network requests to pull data from the blockchain. By keeping fewer large accounts, the client can sustainably make fewer RPC requests and avoid getting rate limited by the provider.

The only major issue of the large account approach is that the protocol can no longer deterministically set the account addresses by storing the data in PDAs. The large Dex accounts easily exceed the 10KB limit, and as a result must be allocated to traditional keypairs. The primary downside is that those public keys cannot be derived from the client, but the simplicity of the protocol interface easily outweighs this minor inconvenience.

2.4 Protocol Limitations

Currently, all of the accounts owned by the Dex are allocated to be a fixed size. Because of this, there are limitations on the number of supported outrights, combos, positions, and orders for the `MarketProductGroup` and `TraderRiskGroup` objects.

```
pub const MAX_OUTRIGHTS: usize = 128;
pub const MAX_COMBOS: usize = 128;
pub const MAX_TRADER_POSITIONS: usize = 16;
pub const MAX_OPEN_ORDERS_PER_POSITION: u64 = 256;
pub const MAX_OPEN_ORDERS: usize = 1024;
```

It is possible to make some of these constraints more flexible in the future, but there is currently no workaround. The main restrictions here are rent cost for the user and compute budget for handling complex operations like cross margining. A user with 16 (the current maximum) active positions will require more compute for determining account health and risk limits than a user with a single active position. Another nice aspect of having statically sized objects is that it greatly improves the capability of client code to query and search for accounts. Solana is a heavily write-optimized blockchain, but this can affect read-efficiency. Therefore, making an active effort to design for efficient reads can have positive downstream effects on user experience.

2.5 Position Tracking

The most important role of the Dex contract is keeping track of the positions of individual traders. The primary challenge is the asymmetry between aggressive and passive fills on an order book. Each passive fill event (i.e. market maker fill) contains enough information to atomically update the positions and balances for both buyer and seller. However, trades are always initiated by the aggressor. As a result, there can be instances of aggressive trades that remove a large number of passive quotes. From a user experience perspective, it is beneficial to know that

1. The trade transaction was successful and the aggressive order was filled.
2. The fill size of the trade as well as the corresponding change in the account balance match the expected behavior from the client.

The faster the trader can get a response for their market orders, the less ambiguity there is between reality and expectation for the end user.

This is possible to guarantee for the taker, but not for the maker(s) due to Solana's transaction size limitation. Because it is clunky (and often impossible) to include the accounts of every maker who was filled on a large trade transaction, this necessitates the existence of a queue to asynchronously process trades after the order book state has been modified. This implementation detail has existed since the first version of Serum and is still present in the AAOB. However, the AAOB does not perform any accounting on the trades after they are processed and instead expects the calling program to record state changes.

To deal with the taker/maker asymmetry, the `TraderRiskGroup` keeps track of taker fills that have yet to be processed from the event queue in a variable named `pending_position`. When the fill is eventually processed, the `position` field of both maker and taker are

updated and the taker's `pending_position` is modified to reflect this change. Importantly, only `position` is taken into account when considering funding.

All deposits and withdrawals to the exchange will transact a specific SPL Token that is specified on `MarketProductGroup` initialization. In practice, it will likely be a stablecoin. Cash positions from deposits and trades are stored in a variable on the `TraderRiskGroup` called `cash_balance`. This variable (and the corresponding `pending_cash_balance` variable from taker trades) are also updated on trade fill events.

2.6 Order Tracking

Because accounts on Solana cannot be reallocated as of version 1.8 (expected to be a feature in future network updates), storing dynamic objects will always have limitations. The AAOB contract represents orders with a `u128` identifier. This identifier encodes the price of the order (most significant 64 bits) as well as the sequence number for the event (least significant 64 bits). It is then used as the key in a critbit tree (trie) representing orders. Cleverly, the contract will invert the bits of the sequence number when creating a new order for a bid, to preserve the desired sorting order. This is how the order book supports the price-time FIFO matching algorithm.

Each `TraderRiskGroup` needs to track existing orders for each of its positions. The Dex protocol represents all of the orders in a custom data structure: a collection of disjoint linked lists that are all stored in a fixed size buffer. The benefits of this structure are:

- It keeps the orders for all products tightly packed.
- A trader can have a higher order limit for an individual product as opposed to a similar data structure like the following:

  ```
  struct OpenOrders {
       orders: [u128; MAX_OPEN_ORDERS_PER_POSITION * MAX_TRADER_POSITIONS],
  }
  ```

  This schema might be intuitive at first glance, but it restricts a trader with few active positions from potentially creating more orders for a specific product. It's a small nuance, but the custom data structure provides this extra flexibility.

2.7 Funding

The funding mechanisms for derivative contracts generate inherent value. The market price of a derivative contract is safeguarded by arbitrage. Because of funding, poorly priced derivatives should in theory be brought back in line when knowledgeable traders exploit discrepancies between the contract price and funding payout to generate a risk-free profit. In the case of a decentralized derivative contract, blockchain-native risks like security exploits, erroneous contract code, and network instability might increase some of the uncertainty around funding. However, given the proper accounting, audits, and dependency management, the same arbitrage arguments should hold for both decentralized and centralized trading.

Every outright in the `MarketProductGroup` keeps track of the cumulative funding per unit (`funding`). Likewise, every position on a `TraderRiskGroup` keeps track of `last_funding`

which refers to the cumulative funding snapshot of the product at the time of the previous funding settlement.

The settlement algorithm works as follows:

1. When an instrument is ready to fund, the Instrument contract that stores the instrument metadata computes the funding amount per contract unit (`funding_per_unit`) based on an oracle price (or any arbitrary source).

2. The Instrument contract then makes a CPI to the Dex program (`ApplyProductFunding`) to update the `funding` variable:

   ```
   product.funding += funding_per_unit;
   ```

3. For a trader's `position` in an outright, the funding will be credited or debited from that trader's `cash_balance`

4. The funding is applied with the following algorithm:

   ```
   let amount = (product.funding - trader.last_funding) * trader.position;
   trader.cash_balance += amount
   trader.last_funding = product.funding
   ```

Note that the above algorithm only works if funding is always applied prior to any new position changes.

2.8 Combo Trading

Support for combo trading (and spread trading in particular) has the potential to boost overall market liquidity. There are a few reasons for this:

- For cross margining purposes, entering a spread position (e.g. long BTC short ETH) is less risky than entering each leg of the spread individually if the corresponding assets are positively correlated. Entering spread trades allows traders to put on positions with larger notional size for less required capital.

- Traders who need to exit positions (i.e. flatten their portfolio) might opt to use combo markets as opposed to outright markets. Because some combo positions incur less risk than the individual legs, the average bid-ask spread on the combo market will usually be tighter than one or both of the corresponding outright markets. By transacting with the combo order book, liquidity takers can avoid paying a large amount of slippage compared to entering an equivalent position through the outright order books.

Given the many advantages of combo trading, the accounting mechanism of trading combos was designed into the Dex contract.

Combos each have their own order books and metadata in the `MarketProductGroup`, but they do not have associated positions in the `TraderRiskGroup` object. Instead, when a combo trades, the individual legs of that combo are updated in the `TraderRiskGroup` accounts of both the maker and taker according to the combo ratios.

2.9 Risk

The Dex contract treats risk as a black box. This puts the onus of on-chain risk management on an external protocol, allowing the Dex to operate entirely on well-defined inputs. The custom Risk Engine component can choose to support leverage, but it will need to manage the individual margin requirements of each `TraderRiskGroup`. The interaction between the Dex and Risk Engine occurs through a series of CPIs. Any operation that changes a trader's risk profile will trigger a check to the exchange-configured Risk Engine. If the CPI returns an error or indicates that the proposed action brings the `TraderRiskGroup` into an unhealthy state, the full transaction will be blocked.

Every risk implementation will need to be catered towards the supported instruments of a given `MarketProductGroup`. Different instruments will have different (and oftentimes non-linear) risk profiles that will require custom logic for accurately determining margin requirements. Exchange operators that wish to support exotic instruments will likely need to build out robust risk engines to complement these products. Ultimately, the risk interface provides the opportunity for new teams to continually innovate on top of existing base layers of Dexterity to provide more Lego blocks for the Solana DeFi ecosystem.

2.10 Liquidation

When the Risk Engine classifies a `TraderRiskGroup` as below the liquidation threshold, the Dex will allow users to call an instruction called `TransferFullPosition`. Invoking this instruction will initiate the auto-deleveraging procedure. Any trader in a `MarketProductGroup` can choose to take on the portfolio of another trader who is below the liquidation threshold. The instruction first checks that the at-risk trader (`liquidatee`) is below the required threshold. If this check succeeds, the `liquidator` accumulates the positions of the `liquidatee`

```
for (product, value) in liquidatee.positions.iter_mut() {
    liquidator.update_position(product, value)?;
    *value = 0;
}
```

The Risk Engine will also return a liquidation price for the `liquidatee` portfolio. This price should be discounted from the fair value of the portfolio to punish individuals who take on too much risk.

```
liquidatee.cash_balance = liquidation_context.liquidation_price;
liquidator.cash_balance -= liquidation_context.liquidation_price;
```

In the case that the portfolio value is negative, funds will be pulled out of an insurance fund to cover the deficit. If the insurance fund is drained (or does not exist), the deficit will be socialized across all exchange participants. The exact distribution of this social loss is handled by the Risk Engine. Future versions of the protocol will additionally support partial liquidations.

3 Technical Challenges

3.1 Decimal Math

Precision is incredibly important when dealing with accounting. eBPF's limited support of floating point math necessitates the use of integers for numerical operations. This can be bypassed by using fixed point math, but fixed point calculations can lead to unintuitive and potentially inaccurate results as there will always be rounding error when working with non-integer base 10 operations. A future protocol update might opt to build out a similar interface for fixed point math if such a feature will yield significant improvements in performance.

Most programs represent decimals by using a large integer (generally a `u64` or `i64`) to represent the mantissa (m) and a small unsigned integer (usually `u8`) to represent the number of decimal points to shift the mantissa (x). A number can be stored as (m, x) where the value of the number is $m * 10^{-x}$.

When trades occur between different products, the protocol will encounter many arithmetic operations with fractional inputs. Oftentimes these inputs will also have different exponent values. To gracefully manage what essentially results in a large dimensional analysis problem, the team implemented a custom data structure that handles the computation of decimal arithmetic. This interface abstracts away much of the complex overflow checking and rounding logic that is performed when adding, subtracting, multiplying or dividing.

3.2 Negative Prices and Price Transformation

In the world of derivatives trading, prices are not always positive. Futures spreads are the canonical example. Suppose there was a futures spread contract where purchasing a single unit would correspond to gaining a long position in the future that expires in 1 month and gaining a short position in the future that expires in 3 months. If the futures curve is in contango (i.e. the 3 month future trades at a higher market price than the 1 month future), the price of this futures spread will be negative. This is because "buying" the futures spreads yields a net cash credit to trader. This cash credit is offset by the negative value of the position.

This is not always the case, but because combos can be defined with arbitrary ratios, it is important to be able to support negative trade prices to allow for truly adaptable markets. A futures spread that changes from being in backwardation to being in contango necessitates an order book supporting both positive and negative prices.

The technical challenge is that the AAOB only supports positive prices. These prices are interpreted as fixed point numbers with 32 integer and 32 fraction bits (`fp32`), but represented in the runtime as `u64`s. Additionally, the order book data structure is lexicographically sorted by the 64-bit string that represents the price. It so happens that `u64`s preserve both numeric and lexicographical ordering in the 8-byte representation.

However, the same property does not hold for `i64`s. Because `i64` represents negative numbers using Two's Complement, all negative numbers are lexicographically larger than all positive numbers. Therefore, a solution to this problem would represent negative numbers in a such a way that neither property (lexicographical or numeric ordering) is violated. The solution is to translate what each `u64` value maps to based off a predetermined offset (custom per product):

```
-OFFSET maps to 0x0000000000000000
```

`u64::max_value() - OFFSET maps to 0xFFFFFFFFFFFFFFFF`

An unrelated but desirable property is to represent prices on the order book as integer tick sizes as opposed to direct prices. This conveniently side steps the fixed point math done by the AAOB (the 32 fractional bits of integers represented as `fp32` are all 0), but the trade-off is a loss in precision in the maximum number of supported price ticks. The team could not think of a case where 4,294,967,296 unique prices would be insufficient for a given product or combo, so this was an acceptable design trade-off.

Adjusting for tick size only requires a division, so the new map looks like the following:

```
-OFFSET / tick_size maps to 0x00000000
(u32::max_value() - OFFSET) / tick_size maps to 0xFFFFFFF
```

Note that the precision of the range decreased as a result of constraining prices to integer tick sizes.

When sending an order at price `P` from the Dex to AAOB, the following linear transformation is applied:

```
((P + OFFSET) / tick_size) << 32
```

The final 32 bit shift casts the unsigned 32 bit adjusted tick size to `fp32` (stored in a `u64`).

To retrieve data from the order book, the protocol just applies the inverse of the transformation that was performed to process the data. However, there is nuance related to how trades should be handled.

Let o denote the offset applied to the limit price. When processing trades from the order book, the matched cash quantity (amount of cash exchanged) is computed as $\sum_j (price_j + o) * size_j$ where j is the index of the each filled order.

The desired target is $\sum_j price_j * size_j$ so the bias term of $o * \sum_j size_j$ must be removed from the original expression. The AAOB returns the variables `total_base_qty_posted` and `total_base_qty` to represent the quantity of contract that was traded. It is known that the matched base quantity is equivalent to `total_base_qty - total_base_qty_posted`, but this is also equivalent to $\sum_j size_j$. We can now compute and remove the bias term and then convert all of the AAOB prices (in adjusted tick space) back into real prices by reversing the original linear transformation.

Disclaimer

This paper is for general information purposes only. It does not constitute investment advice or a recommendation or solicitation to buy or sell any investment and should not be used in the evaluation of the merits of making any investment decision. It should not be relied upon for accounting, legal or tax advice or investment recommendations. This paper reflects current opinions of the authors and is not made on behalf of Solana Labs, Hxro Foundation, Jump Crypto, Chicago Trading Company, or their affiliates and does not necessarily reflect the opinions of Solana Labs, Hxro Foundation, Jump Crypto, Chicago Trading Company, or their affiliates or individuals associated with them. The opinions reflected herein are subject to change without being updated.