# Melon Protocol Specification

Melonport AG
team@melonport.com

***Abstract*—The abstract goes here.**

## I. Introduction

Fund administration consists of four parts. Fund Custodian, Fund Accountant, KYC/AML and Risk Management. In the following paper we will describe how fund administration can be implemented using smart contracts.

## II. Background

### A. Custodian

### B. Decentralized Execution

*1) Assets:* An example for such computer code is known as the ERC20 standard. Essentially a small (¡100 lines of code) piece of software implementing a bitcoin-like cryptocurrency.

*2) Exchanges:* Another example are exchanges. Given above concept one now can implement computer code one how exchange of above assets can be facilitated in a decentralised way.

*3) Investment Funds:* Using the concept that smart contracts can be custodian of assets. Once a smart contract holds assets there needs the be custom functions built into the smart-contract in order to spend those assets again. Lack of such function means that assets are forever lost.

we can now build smart contracts that act as fully functional investment funds.

To manage their holdings these investment funds use decentralised exchanges to buy and sell assets.

## III. Investment Funds Design

### A. Governance

The main functionality of the Governance contract is to add new protocol versions such as this Version contract and to shut down existing versions once they become obsolete.

Adding a *new protocol version* is done by anyone proposing a version to be added and is executed once authority consensus has been established.

Shutting down an existing protocol version is done by anyone proposing a version to be shut down and is executed once authority consensus has been established.

Shutting down a version disables the ability to setup new funds using this version and enables anyone to shut down any existing funds of this version.

### B. Investment Fund

Melon fund as a smart contract that acts as the custodian and accountant. Broadly speaking a Melon fund should be capable of accepting/executing subscription and redemption requests, make orders on an exchange, take orders on an exchange and receiving management and performance rewards.

A Melon fund is created by triggering a Solidity function in the version contract.

### C. Modules

Functionality outside the core Fund logic is provided by *Melon modules*. These modules interact with the Fund in a standardized way, depending on their type. This enables modules to be interchangeable within their class, such that two modules of the same class may fill the same *role*, albeit with differing logic, parameters or other characteristics.

Modules, operating outside the core of the protocol, may be built by third parties without the permission of Melon protocol maintainers. *Module builders* are incentivised to create and maintain modules through an inflation mechanism built into the protocol. They are rewarded with Melon tokens based on some usage characteristics for their module.

Melon has six different module classes:

- Exchange Adapters
- Rewards
- Participation
- Risk Management
- Asset registrar
- Data feed

These classes can be categorized into three subsets:

- Libraries
- Boolean functions
- Infrastructure

## IV. Specifications

### A. Asset Registrar

Asset registrar module is intended for registering assets and information associated with them on-chain. Subsequent mod-

ules check if an asset is registered in the Asset registration before performing different operations. Registration can be done via the register function that takes the following parameters:

| Name | Data Type | Description |
|---|---|---|
| ofAsset | address | A human readable name of the fund |
| name | string | Human-readable name of the Asset as in ERC223 token standard |
| symbol | string | Human-readable symbol of the Asset as in ERC223 token standard |
| decimal | uint | Number of decimals for precision |
| url | string | for extended information of the assetn |
| ipfsHash | bytes32 | Same as url but for ipfs |
| chainId | bytes32 | Chain where the asset resides |
| breakIn | address | Address of break in contract on destination chain |
| breakOut | address | Address of break out contract on this chain |

Since storage on-chain is expensive, a url and / or an ipfsHash of the document containing asset information is stored. An ipfsHash is although preferred to ensure data integrity.

### B. Data Feed

Data feeds are instances of smart contracts that route external data which include asset prices into smart contracts. These inputs could be staked and validated on-chain in order to prevent incorrect / manipulative inputs.

Data feed uses update to periodically update prices of all the assets in one pass. Historic updates are also stored for reference purposes.

*getReferencePrice* can be used to query the price of any given asset in reference to another asset. As of now, It is necessary that either of the assets needs to be the reference asset ($QUOTEASSET$) which is set during contract creation. Following the analogy of currency pair, it takes the addresses of the base and quote assets as the input parameters. The quote asset is used as a reference to return the relative value of the base asset.

If the quote asset is just the reference asset, the latest updated price of the base asset from the data history is returned. If not, *getInvertedPrice* is called which returns the price based on the formula of (*getDecimals(baseAsset)* * *getDecimals(quoteAsset))* / *latestPrice(baseAsset)*. It basically returns the price of reference asset relative to the base asset. It also ensures input and quote assets are compatible in terms of precision (decimals).

*getOrderPrice* takes the inputs of address of base asset, sellQuantity and buyQuantity. It returns the price of the order based on these inputs using $buyQuantity * (10 * *uint(getDecimals(ofBase)))/(sellQuantity)$.

**Algorithm 1** Update algorithm

1: **procedure** UPDATE($assets, prices$) ▷ Update asset prices
2: $i \leftarrow nextUpdateId$
3: **for** $j = 0; j < assets.length; j \leftarrow j + 1$ **do**
4: $dataHistory[i][assets[j]] \leftarrow prices[j]$ ▷ To keep track of historial price updates
5: **end for**
6: $nextUpdateId \leftarrow nextUpdateId + 1$
7: **end procedure**

### C. Accounting

### D. Rewards

### E. Participation

Participation module specifies rules for whitelisting investors. It comprises of two boolean functions *isSubscriptionPermitted* and *isRedemptionPermitted* which enforce rules to check if a particular address is allowed to invest and redeem from the fund. The functions *attestForIdentity* and *removeAttestion* both of which take a single parameter of the participant address can be used to add or remove the participant address from the whitelist respectively.

### F. Exchange Adapter

Exchange adapter is an adapter between Melon protocol and decentralized exchanges such as OasisDex, Kyber, Bancor. The exact implementation details are dependant on the decentralized exchange to be compatible with, but there are three primary functions.

*makeOrder* can be used to make an order on the given exchange. It takes the following input parameters:

| Name | Data Type | Description |
|---|---|---|
| onExchange | address | Address of the exchange |
| sellAsset | address | Reference price obtained through DataFeed contract |
| buyAsset | address | Asset (as registered in Asset registrar) to be bought |
| sellQuantity | uint | Quantity of sellAsset to be sold |
| buyQuantity | uint | Quantity of buyAsset to be bought |

*takeOrder* can be used to take a specific order on the given exchange. It takes the following input parameters:

| Name | Data Type | Description |
|---|---|---|
| onExchange | address | Address of the exchange |
| id | uint | Order id |
| quantity | uint | Quantity of order to be executed (For partial taking) |

*cancelOrder* can be used to cancel a specific order. It takes the following input parameters:

| Name | Data Type | Description |
|---|---|---|
| onExchange | address | Address of the exchange |
| id | uint | Order id |

### G. Risk Management

Risk management module primarily comprises of two boolean functions *isMakePermitted* and *isTakePermitted* both of which take input the following parameters:

| Name | Data Type | Description |
|---|---|---|
| orderPrice | uint | Price of Order |
| referencePrice | uint | Reference price obtained through DataFeed contract |
| sellAsset | address | Asset (as registered in Asset registrar) to be sold |
| buyAsset | address | Asset (as registered in Asset registrar) to be bought |
| sellQuantity | uint | Quantity of sellAsset to be sold |
| buyQuantity | uint | Quantity of buyAsset to be bought |

Custom logic can be defined based on the input parameters that enforces the behaviour of those two boolean functions. (Example: return true if $orderPrice >= referencePrice$).

Our simple flavor of risk management module called RMMakeOrders checks whether the orderPrice falls within the maximum allowed deviation relative to the reference price. It evaluates $orderPrice <= referencePrice - (riskLevel * referencePrice)/riskDivisor$ and disallows the make or take order if it is true. riskLevel is the magnitude of deviation allowed from reference price.

The intuition behind it is that the order price should always be higher than or close to the reference price. Order price is directly proportional to the buyQuantity and inversely proportional to the sellQuantity. It is always beneficial to have the buyQuantity as high as possible.

### H. Version

The Version contract maps one-to-one with a protocol version, meaning that each deployment of the protocol contains a single Version contract.

The purpose of the deployed Version is to index Funds in each protocol version, and to allow managers to set up and shut down their Funds. Indeed, Funds can *only* be created through calling *setupFund* on a particular Version contract (V-E). Thus, with each particular Version, the interface and initial template of each new fund is identical. Likewise, Funds can be shut down with the manager calling *shutDownFund* through the Version with which the Fund was deployed.

This method of deploying Funds lends itself well to Governance; old Versions can be deprecated to eliminate bugs, or new Versions introduced to add features. Adding a Version starts with a proposal for a new Version, and is executed when authority consensus is established. This process is performed by the Governance contract, and is described in IV-I. Similarly, a Version contract is deprecated when the Governance contract calls *shutDown* on the Version, again requiring authority consensus. This prevents new Funds from being created by this Version, and allows anyone to shut down Funds created through it.

The parameters required for creating a new Version are as follows:

| Name | Data Type | Description |
|---|---|---|
| versionNumber | string | Unique identifier for this protocol version |
| ofGovernance | address | The Governance contract responsible for this Version |
| ofMelonAsset | address | The Melon token for the current blockchain |

### I. Governance

Governance uses *ds-group* module to require and establish consensus of authorities for the operations of adding and shutting down versions. Governance contract is first created by specifying the following parameters:

| Name | Data Type | Description |
|---|---|---|
| ofAuthorities | address[] | Array of addresses of authorities |
| ofQuorum | uint | Minimum number of signatures from authorities required for proposal to pass by |
| ofWindow | uint | Time limit for proposal validity |

*proposeVersion* can be used by an authority to propose a new version address.

*approveVersion* can be used by an authority other than the proposer to specify their endorsement for the new version proposal. Number of confirmations is incremented.

*triggerVersion* checks if the number of confirmations for that particular version proposal satisfies the quorum condition and adds the proposed version to the version list.

Shutting down a version follows a similar flow with the methods *proposeShutdown*, *approveShutdown* and *triggerShutdown* instead.

## V. Interactions

This section describes how the three main agents of the Melon protocol interact with each other.

The three main agents being: Investor, Manager, Module Operator.

### A. Operator runs Asset registrar

An asset registrar tracks the universe of assets available for management, and information about each asset. The operator is responsible for keeping the registry in check, which may include adding and removing assets.

If an operator desires to register an asset, he must call the *register* function with information about the asset including, but not limited to, its contract address, name, and ticker symbol. An operator may de-register an asset simply by calling *remove* with the asset's contract address.

### B. Operator runs Data feed

A Data feed module requires regular updates from an operator, which act according to the algorithm described in IV-B. The data injected into the feed is used within the fund for accounting purposes, such as calculating the price of shares. The *update* function receives the following inputs:

| Name | Data Type | Description |
|---|---|---|
| ofAssets | address[] | Array of asset addresses |
| newPrices | uint[] | Array of asset prices, in terms of reference asset |

It is the operator's responsibility to call *update* at the interval defined at data feed setup.

### C. Operator runs Participation module

An operator can use *attestForIdentity* and *removeAttestation* to achieve the vanilla functionality of whitelisting and removing from whitelist an investor. They may also define additional functionality, e.g. requiring the subscribing participant to be in a special whitelist status in case the subscription request is for more than 'n' number of shares.

### D. Operator runs Risk management module

An operator can define custom logic for risk management based on order and reference prices. It can also include dynamic data inputs. E.g. Operator can add checks on orderPrice by aggregating the referencePrices from different exchanges. This can be done by inputting price data from different exchanges on-chain periodically preferrably through a custom data feed module.

### E. Manager sets up a new Melon fund

A new Melon fund can be setup by specifying the following parameters via setupFund function of the version contract:

| Name | Data Type | Description |
|---|---|---|
| name | string | A human readable name of the fund |
| referenceAsset | address | Asset against which performance reward is measured against |
| managementRewardRate | uint | Reward rate in referenceAsset per delta improvement |
| performanceRewardRate | uint | Reward rate in referenceAsset per managed seconds |
| participation | address | Participation module |
| riskMgmt | address | Risk management module |
| sphere | address | Sphere module |

### F. Participant invests in a Melon fund

A participant starts to invest in a fund *F* by first creating a subscription request *R*.

Parameters to be specified are:

| Name | Data Type | Description |
|---|---|---|
| giveQuantity | uint | Quantity of Melon tokens to invest |
| shareQuantity | uint | Quantity of fund shares to receive |
| incentiveQuantity | uint | Quantity of Melon tokens to award the entity executing the request |

Properties of *R* are checked against restriction rules specified in the participation module *P* by the boolean function *isSubscriptionPermitted* (e.g checking whether a participant has an attested Uport identity).

*R* is then executed by some entity calling *executeRequest* after certain conditions are satisfied, described immediately below in V-F1.

*1) Delayed requests:* The nature of the blockchain means that data can only be delivered in intervals, the minimum interval being a single blocktime. Because of this, smart contracts are only able to receive and operate on data in intervals as well. Thus, an agent $A$ with access to real time data could interact with a smart contract $C$ between data delivery intervals, and have an information advantage over the contract.

To describe the vulnerability of this model in terms of a Melon fund, let us assume that the share price $S$ is defined as in IV-C, and that $S$ is calculated at regular intervals $I$, such that $I$ is some number of blocktimes.

We use $S_i$ to denote the share price calculated at the last interval $i$.

With no restrictions, $A$ could notice an increasing share price and invest before the next interval, such that $S_i < S_{i+1}$, allowing $A$ to game the system. Note that the same principle applies for selling shares with a decreasing price.

To eliminate this advantage, we have decided to delay investment *execution* time by at minimum one interval.

This way, if $A$ invested between intervals $i$ and $i+1$ it would only have its investment executed at $i+2$.

Here, the most $A$ can know is $S_{i+1}$, which contract $C$ knows as well, since it coincides with data delivery interval $i+1$.

Said differently, by delaying execution of an investment to $i+2$, $A$'s information collapses to that of $C$ at $i+1$.

### G. Participant redeems from a Melon fund

A participant can redeem from a fund by first creating a redemption request.

Parameters to be specified are:

| Name | Data Type | Description |
|---|---|---|
| shareQuantity | uint | Quantity of fund shares to redeem |
| receiveQuantity | uint | Quantity of Melon tokens to receive in return |
| incentiveQuantity | uint | Quantity of Melon tokens to award the entity executing the request |

Properties of *R* are checked against restriction rules specified in the participation module *P* via the boolean function *isRedemptionPermitted*.

*R* is then executed in the way described in V-F, with a delay as described in V-F1.

Redemptions from a Melon fund can be requested as either (i) a redemption for the shares directly in reference asset, or (ii) redeeming a *slice* of the fund's allocation, proportional to the amount of shares being redeemed. The formula for slice allocation is defined in IV-C.

### H. Manager toggles subscription

A manager can turn off and on the ability subscribe to (i.e. invest in) their fund. The function to do so takes no parameters, and is called *toggleSubscription*.

When disabled, subscriptions can no longer be requested by prospective investors.

### I. Manager toggles redemption

Similar to above, a manager may also disable redemption *in reference asset*.

The option to redeem in slices, as described in V-G, is always available. This prevents a large redemption from disturbing the fund's reference asset allocation.

### J. Manager makes an order

Manager can make an order by specifying asset pair, sell and buy quantities as parameters. Asset pair is checked against datafeed module DF through the function *DF.existsData*. Order parameters are then checked against restriction rules specified in the risk management module R via the boolean function *R.isMakePermitted*.

The specified quantity of the asset is given allowance to the selected exchanging via ERC20's approve function.

Order is then placed on the selected exchange through the exchangeAdapter contract E via *E.makeOrder* by specifying the exchange and order parameters as parameters.

The order is filled on the selected exchange (in the future, this could be any compatible decentralized exchange like OasisDex, Kyber, etc.) when the price is met.

### K. Manager takes an order

Manager can take an order by specifying an order id and quantity as parameters. Asset pair is checked against datafeed module DF through the function *DF.existsData*. Order parameters are then checked against restriction rules specified in the risk management module R via the boolean function *R.isTakePermitted*.

The specified quantity of the asset is given allowance to the selected exchanging via ERC20's approve function.

Order id must correspond to a valid, existing order on the selected exchange. Order is then placed on the selected exchange through the exchangeAdapter contract E via E.takeOrder by specifying the exchange and order parameters as parameters.

### L. Manager converts rewards into shares

Manager rewards in the form of ownerless shares of the fund F can be allocated to the manager via *F.convertUnclaimedRewards* function. Ownerless shares refer to the quantity of shares, representing unclaimed rewards by the Manager such as rewards for managing the fund and for performance. First internal stats of F are calculated using *F.performCalculations* function. The quantity of unclaimedRewards is calculated internally using *calcUnclaimedRewards* function.

A share quantity of *unclaimedRewards* * *gav* (from Calculations) is assigned to the manager.

### M. Manager shuts down the fund

A Manager can shut down a fund *F* he owns via *F.shutdown* function.

Investing, redemption (only in reference asset, investors can still redeem in the form of percentage of held assets), managing, making / taking orders, convertUnclaimedRewards are rendered disabled.

## VI. Conclusion

The conclusion goes here.


## Acknowledgment

The authors would like to thank...


## References


[1] H. Kopka and P. W. Daly, *A Guide to LaTeX*, 3rd ed. Harlow, England: Addison-Wesley, 1999.

## APPENDIX A
## TERMINOLOGY

| | |
|---|---|
| Token | By the term token is meant, a digital token of value adhering to the Ethereum token standard [**?**]. |
| Portfolio | A collection of smart-contracts, divided into a core smart-contract(s) and into auxiliary smart-contracts called modules. |
| Portfolio Manager | Portfolios are managed by one person or a group of persons, referred to as a Portfolio Manager. |
| Module | A module is on or a set of smart-contracts which has an auxiliary functionality to the core smart-contract of the portfolio. |
| Time Step | The term time step means, the time interval between blocks on the Ethereum Blockchain. Where time is taken from the block timestamp issued by the miner. For example $(t_i, t_{i+1}]$ is the time between after block with timestamp $t_i$ and the following block with timestamp $t_{i+1}$. |

## APPENDIX B
## PORTFOLIO

Formally we expand the portfolio as follows. Assuming there are $n \in \mathbb{N}$ digital assets available. This constitutes the following vector set $\underline{a}$ of assets:

$$\underline{a} = \begin{pmatrix} a_1 \\ \vdots \\ a_n \end{pmatrix} \tag{1}$$

where $a_k$ is the k-th available asset, for $k \in \mathbb{N}$. By convention the first asset $a_1$ represents Ether.

Then the *portfolio* $\underline{h}_m^{t_i}$, is defined as the vector set of asset holdings of a portfolio $m$ at time $t_i$.

$$\underline{h}_m^{t_i} = \begin{pmatrix} h_{a_1}^{t_i} \\ \vdots \\ h_{a_n}^{t_i} \end{pmatrix} \in \mathbb{R}_{\geq 0}^n \tag{2}$$

where $h_{a_k}^{t_i}$ is the amount, in token units of $a_k$, a portfolio $m$ holds at time $t_i$, for $k \in \mathbb{N}$.

## APPENDIX C
## GROSS ASSET VALUE

Let $\underline{p}^{t_i}$ be the vector set of asset prices, at time $t_i$. Then $\underline{p}^{t_i}$ is:

$$\underline{p}^{t_i} = \begin{pmatrix} p_{a_1}^{t_i} \\ \vdots \\ p_{a_n}^{t_i} \end{pmatrix} \in \mathbb{R}_{\geq 0}^n \tag{3}$$

where $p_{a_k}^{t_i}$ is the price per token unit of asset $a_k$ in Ether, at time $t_i$, for $k \in \mathbb{N}$.

Note, since by convention $a_1$ represents Ether, and the prices are given in Ether the first price $p_{a_1}^{t_i}$ is always equal to one.

The Gross Asset Value or *GAV* $\hat{v}_{h_m}^{t_i}$ in Ether of portfolio $\underline{h}_m^{t_i}$ at time $t_i$ is:

$$\hat{v}_{h_m}^{t_i} = \langle \underline{p}^{t_i}, \underline{h}_m^{t_i} \rangle = \sum_{k=1}^{n} p_{a_k}^{t_i} h_{a_k}^{t_i} \tag{4}$$

with the standard scalar product on $\mathbb{R}^n$. The GAV can be seen as the gross *value* of the portfolio.

## APPENDIX D
## NET ASSET VALUE

The Net Asset Value or *NAV* $v_{h_m}^{t_i}$ in Ether of portfolio $\underline{h}_m^{t_i}$ at time $t_i$ is:

$$v_{h_m}^{t_i} = \hat{v}_{h_m}^{t_i} - \textit{Management Fees}^{t_i} - \textit{Performance Fees}^{t_i} \tag{5}$$

*Management Fees*$^{t_i}$ resp. *Performance Fees*$^{t_i}$ is the management resp. performance fees given to the Portfolio Manager for timestep $t_i$.

## APPENDIX E
## DELTA

To define the *Delta* $\Delta_{(t_i,t_j]}$ of a portfolio $m$, within the time $(t_i, t_j]$, where $t_i < t_j$, we first define the Delta of $\Delta_{(t_i,t_{i+1}]}$, i.e. the Delta of a single time step. Let be:

$$\begin{aligned} t_0 &= \textit{time of contract creation} \\ t_l &= \textit{time of first investment} \\ &= \min_{t_k \in [t_0,t_i]} \{v_{h_m}^{t_k} \neq 0\} \\ I^{t_i} &= \textit{Sum of all investments within } (t_{i-1}, t_i] \\ W^{t_i} &= \textit{Sum of all withdrawals within } (t_{i-1}, t_i] \end{aligned}$$

Where both $I^{t_i}$ and $W^{t_i}$ is a value in Ether. Then the Delta of a single time step is:

$$\Delta_{(t_i,t_{i+1}]} = \frac{v_{h_m}^{t_{i+1}} - I^{t_{i+1}} + W^{t_{i+1}}}{v_{h_m}^{t_i}} \tag{6}$$

By design, the Delta of a portfolio, is independent of funds invested or withdrawn within the time $(t_i, t_{i+1}]$. By factoring together these Deltas of single time steps we get the general definition of the Delta $\Delta_{(t_i,t_j]}$ of a portfolio $m$ as:

$$\Delta_{(t_i,t_j]} = \begin{cases} 1 & \text{if} \quad t_j \leq t_l \\ \prod_{k=l}^{j-1} \Delta_{(t_k,t_{k+1}]} & \text{if} \quad t_i < t_l < t_j \wedge v_{h_m}^{t_k} \neq 0,\ k \in \{l, l+1, \ldots, j-1\} \\ \prod_{k=i}^{j-1} \Delta_{(t_k,t_{k+1}]} & \text{if} \quad t_l \leq t_i \wedge v_{h_m}^{t_k} \neq 0,\ k \in \{i, i+1, \ldots, j-1\} \\ 0 & \text{otherwise} \end{cases} \tag{7}$$

Note, the case where $t_i < t_l < t_j$ and $v_{h_m}^{t_k} = 0$, for a $k \in \{l, l+1, \ldots, j-1\}$ is when the first investment has been made but the funds have been withdrawn completely at some point within time $(t_l, t_{j-1}]$. The GAV in this case can not be calculated by factoring together Deltas of single time steps, as this would mean a division through zero. The Delta in this case is set to 0 for all times, even if the portfolio receives investments in the future. The same is true for the second similar case where $t_l \leq t_i$ and $v_{h_m}^{t_k} = 0$, for a $k \in \{i, i+1, \ldots, j-1\}$.

By induction, we can see that the Delta $\Delta_{(t_i,t_j]}$ remains independent of funds invested or withdrawn within the time $(t_i, t_j]$.

## APPENDIX F
## NET ASSET VALUE PER SHARE

Expressed mathematically the *NAV* per share $p_m^{t_i}$ of portfolio $m$, at time $t_i$ is:

$$p_m^{t_i} = \Delta_{(t_0,t_i]} \tag{8}$$

where $t_0$ is the time of contract creation. The price $p_m^{t_i}$ is denominated in Ether.

## APPENDIX G
## SHARE PRICE

The share price is defined by the Net Asset Value per Share (see appendix F) and is denominated in Ether.