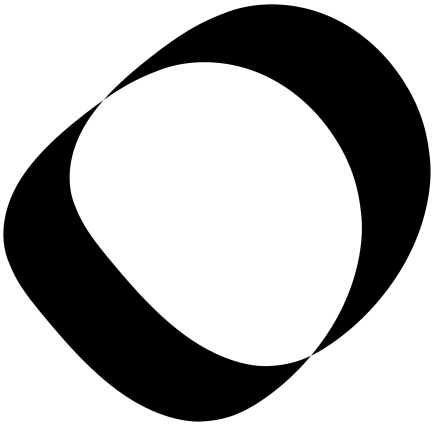
Obymare

Whitepaper



By Big BLYMP

Contents

I. Introduction

Obymare is a decentralized, non-custodial lending protocol built on the Cardano blockchain. It allows users to lock ADA as collateral to mint OBYUSD, a stablecoin pegged to the US dollar. Key features include:

- **Collateralized Borrowing**: Users can mint OBYUSD by locking ADA as collateral, providing access to liquidity without selling their assets.
- **Flexible Debt Management**: Borrowers can adjust their collateral and debt or close positions to reclaim locked ADA.
- **Liquidation Incentives**: Undercollateralized positions can be liquidated, rewarding liquidators with a portion of the collateral.

The protocol utilizes OBY, its native utility token, to cover fees and facilitate system operations. OBY is distributed by the admin and required to interact with the dApp.

This document provides:

- A clear explanation of the protocol's functionality for both users and administrators.
- Technical specifications, including validators, constants, tokens, datatypes, and transaction processes.
- Commentary on risks, design choices, and additional terms.

For a non-technical overview, please refer to the litepaper.

For users

From a user's perspective, the protocol offers several actions to manage collateral and debt effectively:

- **Opening a Position**:

  Users can lock ADA as collateral at a designated script address and mint OBYUSD (a stablecoin pegged to the US dollar). To ensure system security, the collateral-to-debt ratio must meet the Minimum Collateral Ratio (MCR), a predefined safety threshold. A fee in the protocol's utility token, OBY, is also required.

  When a position is opened, two tokens are minted to govern its ownership:

  - The **validity token**, held in the position's UTXO (unspent transaction output) at the script address.
  - The **auth token**, held by the user to control the position.

  These tokens are paired and referred to as "twins."

- **Managing a Position**:

  The position owner has full control and can:

  - **Adjust the Position**:
    - Mint or burn OBYUSD to increase or decrease debt.
    - Deposit or withdraw ADA to modify collateral.
    - All adjustments must adhere to the protocol's rules, ensuring the position remains compliant with MCR.
  - **Close the Position**:

        - Repay all debt to reclaim the locked collateral.
- **Interacting with Other Positions**:

  Users can interact with positions they don't own:

    - **Redeem OBYUSD Against Another Position**:

      By burning OBYUSD, a user reduces the debt of a position and claims a proportional amount of its collateral. This process is similar to paying off someone else's debt in exchange for part of their collateral.
    - **Liquidate Undercollateralized Positions**:

      If a position's collateral falls below the MCR, anyone can liquidate it. This effectively closes the position and releases the collateral without needing the ownership token.
- **Critical Collateral Ratio (CCR)**:

  If a position falls below the CCR (a lower threshold than MCR), it is deemed uneconomical to liquidate. In this case, liquidators are incentivized with compensation provided by the treasury to ensure the system remains secure.

These features empower users to manage liquidity and risk while maintaining the protocol's overall stability.

For Admin

The administrator plays a central role in managing the protocol's infrastructure and ensuring its smooth operation. Key responsibilities include:

- **Minting and Distributing OBY**:

  The admin is responsible for minting the total supply of OBY, the protocol's utility token. OBY is used to pay transaction fees and maintain system operations. The admin has full discretion over its initial distribution, which may include allocating tokens for treasury, partnerships, or user incentives.
- **Maintaining Exchange Rate Data**:

  The protocol relies on (near) real-time exchange rate data for USD, ADA, and OBY to function correctly. This data is provided through oracles (external data providers). Recognizing that oracle structures or reliability may change over time, the protocol includes an upgradeable mechanism to adapt to such changes.
  - **Initialization**:

    Before users interact with the dApp, the admin submits the `erp-init` transaction. This transaction establishes a "pointer," which links to the current valid exchange rate data verifier (a script that users reference to construct their transactions).
  - **Updating Exchange Rate Sources**:

    At any point, the admin can submit an `erp-update` transaction to modify the data source. This ensures that the protocol can

accommodate changes to oracle infrastructure or switch to a more reliable source when needed.

The admin's role is critical to maintaining the protocol's stability, particularly in ensuring that exchange rate data is accurate and reliable for all user interactions.

Requirements

The protocol's design adheres to specific requirements to ensure functionality, flexibility, and security:

- **Maximum Debt Limit**:
  Each position has a cap on the amount of debt it can accrue. If a user needs to exceed this limit, they can open multiple positions to distribute their debt across separate instances.
- **Stake Management for Locked ADA**:
  Users retain control over the staking of their locked collateral. This includes the ability to change the staking credentials of an existing position. This feature is particularly useful when a position's ownership has been transferred.
- **Exchange Rate Source Flexibility**:
  The dApp relies on off-chain data, such as ADA-USD exchange rates, provided by oracles. Since oracles may evolve or change, the protocol includes mechanisms to update or replace exchange rate data sources as needed. This ensures the system remains functional even if the oracle infrastructure changes.

- **Redeeming Across Multiple Positions**:

  The protocol allows users to redeem OBYUSD against multiple positions in a single transaction, providing efficiency and convenience. However, safeguards are in place to prevent spam or abuse of this feature.

These requirements collectively ensure the protocol's scalability, adaptability, and user-centric design.

II. Constants

These constants are referenced at various points in the dApp:

```
MILLION = 10^6
TOTAL_OBY=10^13 ; 10 million million
FEE_RATE = 1%
MINIMAL_COLLATERAL_RATIO = 120%
CRITICAL_COLLATERAL_RATIO = 102%
TREASURY_TOP_UP = 105%
MAX_DEBT_AMOUNT = 5 * 10 ^ 12 ; in terms of microobyusd
MIN_DEBT_AMOUNT = 5 * 10 ^ 8; ie $500
MIN_REDEEM_AMOUNT = 10 ^ 9
MAX_TREASURY_UTXO = 10 ^ 10 ; in terms of microusd
```

III. Tokens

The following describes the token names:

```
oby = "OBY"
validity_token_pref = #"100"
auth_token_pref = #"222" // See cip-68/69
erp_validity_token = validity_token_pref + "erp"
erp_auth_token = auth_token_pref "erp"
obyusd = "OBYUSD"
user_validity_token = validity_token_pref + "obymare" + <tag>
user_auth_token = auth_token_pref + "obymare" + <tag>
```

Each pair of user twin tokens uses an input utxo oref as a seed to create an essentially unique tag `<tag>`.

The function `mk_tag : OutputReference -> Tag` uses `blake2b_256` plutus builtin as a hash function, and truncates the result to a 20-byte (160-bit) digest. It leaves 12 bytes of space in token names for human-friendly labels.

IV. Validators

The protocol's on-chain operations are managed through a series of **validators**—smart contracts that enforce specific rules for transactions. Validators may interact with one another, forming a dependency structure. These dependencies fall into two categories:

- **Hard Dependencies**:
  A validator hash is "hard-coded" into another validator, meaning the dependency is fixed and cannot be altered.
- **Soft Dependencies**:
  A validator refers to another validator dynamically, allowing for modifications or updates over time.

To maintain system integrity, the dependency structure must be **acyclic** (no loops in the chain of dependencies). For example:

- If Validator A has a hard dependency on Validator B, Validator B cannot have a hard dependency on Validator A, either directly or indirectly through other validators.
- However, Validator B can have a soft dependency on Validator A, allowing for flexibility without creating cyclic dependencies.

This structure ensures that validators operate in a logical, hierarchical manner, preventing infinite loops and facilitating future updates or changes to the protocol.

```
flowchart LR
    om["
    OM : Oby Mint \n
    Mint
    "]

    er["
    ER : Exchange Rate \n
    Withdraw
    "]

    erp["
    ERP : Exchange Rate Pointer \n
    Spend, Mint
    "]

    m["
    M : Main \n
    Spend, Mint, Withdraw
    "]

    om --> er & m
    er -.-> erp
    erp --> m
```

OM: OBY Mint

OM is a one-shot mint validator producing the total supply of OBY, the dApp utility token. It is executed once prior to the rest of the dApp. It is seeded to ensure a unique validator hash.

Datatypes

See the section on functions for explanations of how this is used.

```
aiken-language
type OmParams = OutputReference

type Om2Red {
  Om2Mint
  Om2Burn
}
```

Constraints

Two arguments - the purpose is mint only.

Om2Mint:

1. `OmParams` is spent
2. Own mint is `[(oby, TOTAL_OBY)]`

Om2Burn:

1. Own mint has only negative quantities

Notes

We retain the ability to burn the token. This is helpful during testing and can be disabled before prod. However, its inclusion has no bearing on the functioning of the dApp.

The role of OM is not integral to the rest of the dApp. It is possible to replace it entirely with, say, an existing token, or a token minted with a native script if that is deemed more desirable.

ER: Exchange Rate

The validator checks the presence of the necessary oracles and verifies that its own redeemer matches the exchange rate attested to by the oracle(s).

Datatypes

Outlined below:

```
aiken-language
type Er2Red {
  u_num : Int,
  u_denom : Int,
  o_num : Int,
  o_denom : Int,
}
```

Constraints

Two arguments - purpose is withdrawal only.

Er2Red:

- Ascertain the true exchange rates from oracles or otherwise.
- The numbers in the redeemer correspond to them

Notes

The dApp requires an up-to-date knowledge of exchange rates of:

- OBY/ADA
- USD/ADA

It is not possible to ascertain the latter solely from within the ledger, and thus, we depend on oracles. This will be an external service on which the dApp depends.

The dApp needs to be able to accommodate the case that an oracle changes its data schema or becomes unreliable. In such a change, the ER script may need to be partially or totally rewritten. It provides a stable interface that the main script expects.

We don't specify params datatypes here, but it will necessarily depend on OBY policy id, and the particularities of the oracles used. For an example oracle see orcfax.

We implement a few different versions.

> **Datum version:** Loosely modelled on orcfax and uses reference inputs to discern correctness. We currently do not have an oracle system on which we can depend so we have implemented our own skeletal form for testing. This is the default implementation, with the label `er`.
>
> The script is parameterized by the oracles' hashes. Reference inputs contain a recognized token and are identified as such.
>
> **Signature version:** This version relies on a trusted signature to have signed the redeemer. If there is a usable Oracle on the mainnet at launch, this is the suggested first implementation, which has the label `er_sig`.

Because the redeemer has been designed with the main script in mind, it does not accommodate the additional data needed to carryout validation. Namely, a time stamp and signature.

To work around this, we supply the additional data via the redeemer of a second withdrawal script. The second script does nothing. `er_sig` checks that:

- The pf redeemer has been signed off together with the time stamp
- The timestamp is within the short tx validity range

Emergency version: Relies on the presence of an auth token, effectively

meaning only admin can spend. The label is `er_em`.

ERP: ER Pointer

This spend/mint validator 'points' at the ER. The pointer is a utxo with a datum recording the current ERP script hash. The utxo contains a validity token twinned with one held by admin and has an inline datum of the hash of the current ER validator.

Datatypes

```
aiken-language
type ErpParams = OutputReference

type ErpDat = Credential

type Erp3Red {
  Erp3Update(Int, Int)
  Erp3Close
}
```

```
type Erp2Red {
  Erp2Init
  Erp2Burn
}
```

Constraints

Two arguments: the purpose is mint.

Erp2Init:

1. `ErpParams` is spent
2. Own mint value is
    a. (erp-auth, 1)
    b. (erp-validity, 1)
3. 0th output is continuing output:
    a. Payment address is own
    b. Value is ada and erp-validity
    c. Datum is `ErpDat`

Erp2Burn:

1. Own mint has only negative quantities

Three arguments.

Erp3Update(`auth_idx`, `cont_idx`):

1. Output `auth_idx` is auth output _ie_ contains erp-auth
2. Output `cont_idx` is continuing output
    a. Payment address is own

b. Value is ada and erp-validity
c. Datum is `ErpDat`

Erp3Close:

1. Own mint value has length 2 (See notes on why this is sufficient)

Notes

We include the abilities necessary for decommissioning the pointer. This is helpful during testing and can be disabled before production if desired.

However, its inclusion does not increase potential vulnerabilities - a malicious admin could set the credential to a spurious address or to one that is dishonest about the exchange rates.

A note on the `Erp3Close` constraint. We care only that:

1. Tx authorized by admin
2. Validity token is burnt

The logic is as follows:

1. Own mint value is of length 2
2. Own script executed with purpose mint. There are two possible redeemers:
3. Either **Erp2Init**. But this must be executed precisely once and is prior to the valid state utxo, so this is impossible.
4. Or **Erp2Burn**. This can only burn tokens. It can only burn tokens minted, of which there are at most two. Thus it must be both.

M: Main

This _main_ validator is responsible for position lifecycles and the treasury. It provides the address for locked collateral and treasury $OBY, as well as the minting policy for OBYUSD, and validity/auth twins.

The main validator is multipurpose, which is executed with the purposes of mint, spend, and withdraw. Mint and spend logic is completely deferred to the withdraw purpose. A note on why this design pattern was adopted is found below(#monolith-to-composite).

A position is a utxo at the Main script address containing a validity token and having an inline datum `Position(debt)`, which tracks the amount of OBYUSD minted by the position. The collateral of a position is the ADA in the value.

The treasury is the set of utxos at the Main script address with datum `Treasury`. There are no other types of valid datums for the main script address.

A user is the owner of a position if they control the twinned auth token. With it, they may adjust and close their position. If a position becomes unhealthy wrt MCR, anyone may liquidate it. If a position becomes very unhealthy, it may be liquidated with compensation from the treasury.

Datatypes

```
aiken-language
 type MParams {
   erp_hash : ByteArray,
```

```
  oby_hash: ByteArray,
}

type M3Red = Data
type Tag = ByteArray

type M2Red {
  Open(OpenParams)
  Adjust(AdjustParams)
  Close(Tag)
  Redeem(Tag)
  Lqd(Tag)
  Lwc(LwcParams)
  Tidy
  Burn
}

type OpenParams {
  seed : OutputReference,
  cont_idx : Int,
  treas_idx : Int,
}

type AdjustParams {
  tag : ByteArray,
  auth_idx: Int,
  cont_idx : Int,
  treas_idx : Int, // Used only when needed
}

type LwcParams = {
  tag : ByteArray,
  treas_idx : Int, // Used only when needed
}

type Debt = Int

type MDat {
  Position(Debt),
  Treasury
}
```

We break the naming convention, justified since this is most commonly used and referenced.

Constraints

Two arguments: the purpose is mint.

1. Withdrawals include own credentials

Two arguments: the purpose is to withdraw.

We factor out the frequently used logic of getting exchange rates:

Get exchange rate:

1. Find ERP ref input via validity token.
    a. read datum for `cred`
2. In redeemers, get value `Er2Red` from key `WithdrawFrom(cred)`

Open:

1. Redeemer is `Open({seed, cont_idx, treas_idx})`
2. `seed` is spent
3. `tag = mk_tag(seed)`
4. No own inputs
5. Output at index `cont_idx`
    a. Own payment address
    b. Value is `cont_col` lovelace and validity token with tag `tag`
    c. Datum is `Position(cont_debt)`. `cont_debt >= MIN_DEBT_AMOUNT`
    `cont_debt <= MAX_DEBT_AMOUNT`
6. Output at index `treas_idx`
    a. Own payment address
    b. Value is ADA + `oby_fee` microoby
    c. Datum is `Treasury`
7. Get exchange rates
8. `(cont_col, cont_debt)` respects MCR
9. `oby_fee` is sufficient fee
10. Own mint value is
    a. (auth token `tag`, 1)

    b. (validity token `tag`, 1)
    c. (obyusd, `cont_debt`)

Adjust:

1. Redeemer is `Adjust({tag, auth_idx, cont_idx, treas_idx})`
2. Own inputs is `[own_input]`
3. Extract `(own_col, own_debt)` from `own_input`
4. `auth_idx` output is auth output
    a. Value has auth token with tag `tag`
5. `cont_idx` output is continuing output
    a. Own payment address
    b. Value is `cont_col` lovelace and validity token with tag `tag`
    c. Datum is `cont_debt` `cont_debt >= MIN_DEBT_AMOUNT` `cont_debt <= MAX_DEBT_AMOUNT`
6. Calculate `del_col = cont_col - own_col` and `del_debt = cont_debt - own_debt`
7. If either `del_col != 0` or `del_debt > 0`, then
    a. Get exchange rates
    b. If either `del_col < 0` or `del_debt > 0`, then `(cont_col, cont_debt)` respects MCR
    c. If `del_debt > 0` then `treas_idx` output is treasury output
        i. Own payment address
        ii. Value is ADA + `oby_fee` microoby. `oby_fee` is sufficient
        iii. Datum is `Treasury`
8. Own mint value is
    a. If `del_debt != 0` then `[(obyusd, del_debt)]`
    b. Else is `[]`

Notes

- We do not check the input's validity token. Instead, we depend on the global constraint that the validity token must always exist at a main payment address. To exist in the output, the validity token must be

either in the input or minted. The latter is impossible here since the mint is checked.

Thus the input must contain the validity token with tag `tag`.

- It is possible to perform an adjust that changes neither the collateral nor debt. This may be used to change the stake credentials of a position.

Close:

1. Redeemer is `Close(tag)`
2. Own inputs is `[own_input]`
3. Extract `own_debt` from `own_input`
4. Own mint value is
    a. (auth token `tag`, -1)
    b. (validity token `tag`, -1)
    c. (obyusd, `-own_debt`)

Notes

- Similarly to the above, the `own_input` must contain the relevant validity token as it is burnt.

Redeem:

Redeeming against multiple positions simultaneously is a must-have feature. To accommodate this, the implementation will be more apparent. Each position redeemed against has a corresponding output. The order of

input positions is the order of their continuing outputs. At most one continuing position can have non-zero debt.

1. Redeemer is `Redeem(only_tag)`
2. Calculate input condensate. For each own inputs
    a. Input is a position, and extract `tag`, `col`, and `debt`
3. Get exchange rates
4. The first set of outputs are the continuing position in the same order as the inputs. Thus we fold (or `map2`) over both:
    a. Address is as input
    b. Value is `cont_col` lovelace and validity token with tag `tag`
    c. Datum is `Position(cont_debt)`
    d. Calculate `delta_col = cont_col - col` and `delta_debt = cont_debt - debt`
    e. If `tag != only_tag`, then `cont_debt == 0` else (`delta_debt < 0` and `cont_debt >= 0`)
    f. Check `delta_col` is expected given exchange rate
    g. Calculate fee tier and fee
    h. Aggregate fee and debt as `total_fee` and `total_debt`
5. The next output is treasury output
    a. Address is own
    b. Value is ada + oby fee matching `total_fee`
    c. Datum is `Treasury`
6. `- total_debt >= MIN_REDEEM_AMOUNT`
7. Own mint value i

    a. (obyusd, `- total_debt`)

Note: this is not the most efficient nor the most flexible design - but a halfway house between the two.

Lqd:

1. Redeemer is `Lqd(tag)`
2. Own inputs is `[own_input]`
3. Extract `(own_col, own_debt)` from `own_input`
4. Get exchange rates
5. `(own_col, own_debt)` is below MCR threshold
6. Own mint value is
    a. (validity token `tag`, -1)
    b. (obyusd, `-own_debt`)

Lwc:

1. Redeemer is `Lwc({tag, treas_idx})`
2. Own inputs is `own_inputs`
3. Only one position input, `own_position`, the rest are treasury inputs, `treasury_inputs`.
4. Extract `(own_col, own_debt)` from `own_position`
5. Extract (`total_comp`, `min_comp`), the total and min microoby respectively from `treasury_inputs`
6. Get exchange rates
7. `(own_col, own_debt)` is below CCR threshold.
8. `comp` is calculated by:
    a. If `treas_idx >= 0`, then `treas_idx` output is treasury output

        i. Own payment address
        ii. Value is ada and `over_comp` microoby
        iii. Datum is Treasury `comp = total_comp - over_comp`
    b. Else `comp = total_comp`
9. All treasury inputs are necessary: `total_comp - min_comp < comp`.
10. Verify `comp` is legitimate
11. Own mint value is
    a. (obyusd, `-own_debt`)
    b. (validity token `tag`, -1)

Tidy:

1. Redeemer is `Tidy`
2. `n_inputs` of own inputs:
    a. Value totals `input_ada` of lovelace and `input_oby` of microoby
    b. Datum is always `Treasury`
3. First `n_outputs` are treasury outputs:
    a. Address is (unstaked) own address
    b. Value is `output_ada` of lovelace and `output_oby` of microoby
    c. Datum is `Treasury`
4. Number of own outputs no greater than own inputs ie `n_inputs >= n_outputs`
5. Output ADA at least as great as input ie `input_ada <= output_ada * n_outputs`

6. Output microoby is (almost) as great as input. (Grace allows for rounding down)
7. Get exchange rates
8. Calculate `max_treasury_oby` from the dollar limit `MAX_TREASURY_UTXO`
9. `max_output <= max_treasury_oby`
10. Own mint value is []
11. Reference inputs includes a position below CCR

Burn:

1. Redeemer is `Burn`
2. Own inputs are []
3. Own mint quantities are < 0

Three arguments:

1. Withdraws includes own credential

Notes

Note that the withdrawal amount may be 0 or otherwise, provided it agrees with the current ledger state.

A user declares their intent in the withdraw redeemer. Each intent places a set of constraints that the tx must satisfy. Only one intent is permitted per tx. The withdrawal script is split into cases along the lines of intents, with some shared logic.

As the withdraw validator is responsible for minting and spending, it must always loop over all the inputs and outputs and check its own mint value.

Edit: earlier iterations of the design were more ambitious and flexible. In the interests of time, this has now been simplified.

V. Functions

Here, we provide a description of some auxiliary functions that the validator and transaction sections brush over.

The main validator needs to calculate:

- The appropriate OBY fee for collateral
- Whether the collateral-to-debt ratio is below MCR or CCR
- In the latter case, how much compensation is the liquidator entitled to
- OBY fee on redemption

First, we describe the formulae without reference to code. Suppose `u` is a rational number representing usd->ada conversion. That is, `u lovelace = 1 microdollar`, equivalently `u ada = 1 dollar`. Let a `(C,D)`-position be a position with `C` collateral ada (as lovelace) and `D` debt. We say that the _collateral to debt ratio_ of a `(C,D)`-position is `C/uD`. Thus, for example, a `(C,D)`-position is above MCR if and only if `C/uD >= 1.2`. More generally, we say a `(C,D)`-position _satisfies an `q`-CR_ if `C/uD >= q`.

Suppose `o` is a rational number representing OBY to ADA conversion. That is, `o lovelace = 1 microoby`, equivalently `o ada = 1 oby`. Then opening a `(C,D)`-position or adjusting an existing position with an increase of in collateral of `C` requires a fee of `(1/100) C/o` OBY to be paid to the treasury.

The task of describing the constraints in code is made mildly more laborious in the absence of rationals. However, it is well beyond our needs to introduce full support for rationals. Let us represent the rational `a` as a 2-tuple, `a = (a_num, a_denom)`, and assume `a_denom > 0`. Thus, for example, if `x` amount of ADA is equivalent to `y` amount of USD then `u_denom * x == u_num * y`.

A `(col,debt)`-position satisfies the `(q_num,q_denom)`-CR if:

```
sample
q_denom * u_denom * col - q_num * u_num * debt >= 0
```

Thus MCR is the `(120, 100)`-CR case, and CCR is the `(103, 100)`-CR case.

A new position with debt `debt`, or an adjustment with an increase of debt `debt` requires an amount of `fee` in OBY to be paid to the treasury.

```
sample
fr_denom * o_num * u_denom * fee - fr_num * o_denom * u_num *
debt >= 0
```

The coefficients for the fee ratio `fr = (1,100)`, representing 1%. In the case of lwc, suppose the compensation in OBY is `compensation`. The compensation cannot exceed `TREASURY_TOP_UP`. Thus:

```
sample
t_denom * u_denom * ( o_num * compensation + o_denom * col )
<= t_num * o_denom * u_num * debt
```

where `t = (105, 100)`.

Separately, we must calculate `max_treasury_utxo` in microoby. The constant `MAX_TREASURY_UTXO` is set in microusd. Thus:

```
sample
max_treasury_utxo_microoby = (MAX_TREASURY_UTXO * u_num *
o_denom) / (u_denom * o_num)
```

In the case of a redemption, the fee rate depends on the CR. The fee rate is 1% for CR `<= 150%` and rises linearly to a maximum of 3% fee at CR `>=1000%`. We wish to compute this to 2dp (as a %) resolution.

The line `y = mx + c` passing through the points `(150, 100)`, `(1000, 300)` has the coefficients `(m, c) = (4/17, 1100/17)`. This translates to the following:

```
f_num_lin = ( 400 * u_denom * col + 1100 * u_num * debt ) /
(17 * u_num * debt )
f_num = [math.min(300, math.max(f_num_lin, 100)), 100]
```

We aggregate `fee_usd_100 = delta_debt * f_num`, and finally convert to microoby (and flip sign).

```
fee_oby = - ( fee_usd_100 * u_num * o_denom ) / ( u_denom *
o_num * 100 )
```

The user can redeem `- (delta_debt * u)` from the collateral. Thus, the condition to check is:

```
delta_col * u_denom >= delta_debt * u_num
```

VI. Transactions

A transaction consists of inputs and outputs of utxos. Each utxo has a:

- Unique reference,
- An address it belongs to
- A value of ADA + native assets (possibly no native assets)
- Maybe a datum or datum hash. (Note that a utxo at a script address must have a datum or datum hash to be spendable.)

Addresses correspond to either agents (e.g., normal addresses), or script addresses. All transactions must have at least one input that belongs to a payment address _ie_ an agent's address. When there are no constraints as to who the payment address belongs to, then we may write _Any_ as the address owner.

Addresses, unless otherwise stated, refer to only the payment credential while the staking credential is unrestricted. The diagrams are to be indicative and informative, rather than provide an exhaustive description.

In the following diagrams, boxes denote utxos. Owners of an address are stated first, then any notable assets or ADA (that is, ignoring min ada or ada value doesn't change or change is negligible), and then datums where relevant. All datums are assumed inline unless stated otherwise. A download icon indicates withdrawals.

`tx.mint_oby`

Admin mints OBY. This tx is _seeded_ to ensure that the policy id is unique.

```
mermaid
flowchart LR
    i_admin["
    fa:fa-address-book Admin\n
    oref: seed
    "]

    s_om["
    fa:fa-circle OM\n
    params: seed
    "]

    m["tx"]

    o_admin["
    fa:fa-address-book Admin\n
    fa:fa-circle (oby, TOTAL_OBY) \n
    "]

    i_admin --> m --> o_admin
    s_om -.->|Om2Init| m
```

`tx.erp_init`

Admin initializes the pointer, creating validity/auth twins. This tx is _seeded_ to ensure that the policy IDs and script address cannot be duplicated.

```
mermaid
flowchart LR
    i_admin["
    fa:fa-address-book Admin\n
    oref: seed
    "]

    s_erp["
    fa:fa-circle ERP\n
    params: seed
```

```
    "]

    m["tx"]

    o_erp["
    fa:fa-address-book ERP\n
    fa:fa-circle erp-validity \n
    fa:fa-database ErpDat
    "]

    o_admin["
    fa:fa-address-book Admin\n
    fa:fa-circle erp-auth \n
    "]

    i_admin--> m --> o_erp & o_admin
    s_erp -.->|Erp2Init| m
```

`tx.erp_update`

Admin updates the pointer datum.

```
mermaid
flowchart LR
    i_erp["
    fa:fa-address-book ERP\n
    fa:fa-circle erp-validity \n
    fa:fa-database ErpDat
    "]

    i_admin["
    fa:fa-address-book Admin\n
    fa:fa-circle erp-auth \n
    "]

    m["tx"]

    o_erp["
    fa:fa-address-book ERP\n
    fa:fa-circle erp-validity\n
    fa:fa-database ErpDat
    "]

    o_admin["
    fa:fa-address-book Admin\n
    fa:fa-circle erp-auth\n
```

```
    "]

    i_erp -->|Erp3Update| m
    i_admin --> m --> o_erp & o_admin
```

`tx.erp_burn`

Admin closes the pointer datum and burns auth/validity tokens.

```
mermaid
flowchart LR
    i_erp["
    fa:fa-address-book ERP\n
    fa:fa-circle erp-validity \n
    fa:fa-database ErpDat
    "]

    s_erp["
    fa:fa-circle ERP\n
    "]

    i_admin["
    fa:fa-address-book Admin\n
    fa:fa-circle erp-auth \n
    "]

    m["tx"]

    i_admin --> m
    i_erp -->|Erp3Close| m
    s_erp -->|Erp2Burn| m
```

`tx.open`

A user opens a position.

```
mermaid
flowchart LR
    i_ada["
    fa:fa-address-book User\n
    fa:fa-circle ada, oby\n
    "]
```

```
    r_erp["
    fa:fa-address-book ERP \n
    fa:fa-circle erp-validity \n
    fa:fa-database Cred(ER)
    "]

    w_m["
    fa:fa-download M \n
    "]

    m_m["
    fa:fa-circle M \n
    "]

    w_er["
    fa:fa-download ER \n
    "]

    r_ora_usd["
    fa:fa-address-book UsdOra \n
    fa:fa-circle oracle-nft \n
    fa:fa-database Oracle-usd
    "]

    r_ora_oby["
    fa:fa-address-book ObyOra \n
    fa:fa-circle oracle-nft \n
    fa:fa-database Oracle-oby
    "]

    m["tx"]

    o_position["
    fa:fa-address-book M\n
    fa:fa-circle ada, user-validity-token \n
    fa:fa-database Position
    "]

    o_treasury["
    fa:fa-address-book M\n
    fa:fa-circle oby \n
    fa:fa-database Treasury
    "]

    o_obyusd["
    fa:fa-address-book User \n
    fa:fa-circle obyusd, user-auth-token\n
    "]
```

```
    i_ada --> m --> o_position & o_treasury & o_obyusd
    w_m -.->|Open| m
    m_m -.->|Data| m
    w_er -.->|Er2Red| m
    r_erp & r_ora_usd & r_ora_oby -.-o m
```

`tx.adjust`

A user adjusts their position. There are different conditions that must be satisfied depending on whether the collateral and/or debt are increased, decreased or unchanged.

```
mermaid
flowchart LR
    i_position["
    fa:fa-address-book M\n
    fa:fa-circle ada, user-validity-token \n
    fa:fa-database Position(debt)
    "]

    i_ada["
    fa:fa-address-book User\n
    fa:fa-circle ada, user-auth-token, oby\n
    "]

    w_m["
    fa:fa-download M \n
    "]

    m_m["
    fa:fa-circle M \n
    "]

    w_er["
    fa:fa-download ER \n
    "]

    r_erp["
    fa:fa-address-book ERP \n
    fa:fa-circle erp-validity \n
    fa:fa-database Cred(ER)
```

```
    "]

    r_ora_usd["
    fa:fa-address-book UsdOra\n
    fa:fa-circle oracle-nft \n
    fa:fa-database Oracle?
    "]

    r_ora_oby["
    fa:fa-address-book ObyOra\n
    fa:fa-circle oracle-nft \n
    fa:fa-database Oracle?
    "]

    m["tx"]

    o_position["
    fa:fa-address-book M\n
    fa:fa-circle ada, validity-token \n
    fa:fa-database Position
    "]

    o_treasury["
    fa:fa-address-book M\n
    fa:fa-circle oby \n
    fa:fa-database Treasury
    "]

    o_obyusd["
    fa:fa-address-book User \n
    fa:fa-circle auth-token, obyusd \n
    "]

    i_position -->|Data| m
    w_m -.->|Open| m
    m_m -.->|Data| m
    w_er -.->|Er2Red| m
    i_ada --> m --> o_position & o_treasury & o_obyusd
    r_erp & r_ora_usd & r_ora_oby .-> m
```

`tx.close`

```
mermaid
flowchart LR
    i_position["
    fa:fa-address-book M\n
```

```
    fa:fa-circle ada, validity-token \n
    fa:fa-database Position
    "]

    w_m["
    fa:fa-download M \n
    "]

    m_m["
    fa:fa-circle M \n
    "]

    i_auth["
    fa:fa-address-book User\n
    fa:fa-circle auth-token, obyusd\n
    "]

    m["tx"]

    o_owner["
    fa:fa-address-book User \n
    fa:fa-circle ada \n
    "]

    i_position -->|Data| m
    w_m -.->|Close| m
    m_m -.->|Data| m
    i_auth --> m --> o_owner
```

`tx.redeem`

A user redeems with other users' positions. In this example, the user redeems using two positions, clearing entirely the debt of the first.

```
mermaid
flowchart LR
    i_user["
    fa:fa-address-book User\n
    fa:fa-circle obyusd \n
    "]

    i_position_1["
    fa:fa-address-book M\n
    fa:fa-circle ada, v1 \n
    fa:fa-database Position(d1)
```

```
    "]

    i_position_2["
    fa:fa-address-book M\n
    fa:fa-circle ada, v2 \n
    fa:fa-database Position(d2)
    "]

    w_m["
    fa:fa-download M \n
    "]

    m_m["
    fa:fa-circle M \n
    "]

    w_er["
    fa:fa-download ER \n
    "]

    r_erp["
    fa:fa-address-book ERP \n
    fa:fa-circle erp-validity \n
    fa:fa-database Cred(ER)
    "]

    r_ora_usd["
    fa:fa-address-book UsdOra \n
    fa:fa-circle oracle-nft \n
    fa:fa-database Oracle?
    "]

    r_ora_oby["
    fa:fa-address-book ObyOra \n
    fa:fa-circle oracle-nft \n
    fa:fa-database Oracle-oby
    "]

    m["tx"]

    o_position_1["
    fa:fa-address-book M\n
    fa:fa-circle ada, v1 \n
    fa:fa-database Position(0)
    "]

    o_position_2["
    fa:fa-address-book M\n
```

```
    fa:fa-circle ada, v2 \n
    fa:fa-database Position(d2')
    "]

    o_treasury["
    fa:fa-address-book Main\n
    fa:fa-circle oby \n
    fa:fa-database Treasury
    "]

    o_user["
    fa:fa-address-book User\n
    fa:fa-circle ada \n
    "]

    i_user --> m --> o_user
    i_position_1 & i_position_2 -->|Data| m --> o_position_1
& o_position_2 & o_treasury
    w_m -.->|Redeem| m
    m_m -.->|Data| m
    w_er -.->|Er2Red| m
    r_erp & r_ora_usd & r_ora_oby .-o m
```

`tx.lqd`

A user liquidates another user's position.

```
mermaid
flowchart LR
    i_position["
    fa:fa-address-book M\n
    fa:fa-circle ada, user-validity-token \n
    fa:fa-database Position(debt)
    "]

    w_m["
    fa:fa-download M \n
    "]

    m_m["
    fa:fa-circle M \n
    "]

    w_er["
    fa:fa-download ER \n
    "]
```

```
    r_erp["
    fa:fa-address-book ERP \n
    fa:fa-circle erp-validity \n
    fa:fa-database Cred(ER)
    "]

    r_ora_usd["
    fa:fa-address-book UsdOra \n
    fa:fa-circle oracle-nft \n
    fa:fa-database Oracle?
    "]

    r_ora_oby["
    fa:fa-address-book ObyOra \n
    fa:fa-circle oracle-nft \n
    fa:fa-database Oracle-oby
    "]


    m["tx"]

    o_user["
    fa:fa-address-book User\n
    fa:fa-circle ada \n
    "]

    i_position -->|Data| m --> o_user
    w_m -.->|Lqd| m
    m_m -.->|Data| m
    w_er -.->|Er2Red| m
    r_erp & r_ora_usd & r_ora_oby .-o m
```

Note: Main does not need the Oby ER, however, it is simpler to always include it.

`tx.lwc`

A user liquidates another user's position with treasury assistance.

```
mermaid
flowchart LR
    i_position["
    fa:fa-address-book M\n
    fa:fa-circle ada, user-validity-token \n
    fa:fa-database Position(debt)
    "]

    i_treasury_0["
    fa:fa-address-book M\n
    fa:fa-circle oby \n
    fa:fa-database Treasury
    "]

    i_treasury_1["
    fa:fa-address-book M\n
    fa:fa-circle oby \n
    fa:fa-database Treasury
    "]

    w_m["
    fa:fa-download M \n
    "]

    m_m["
    fa:fa-circle M \n
    "]

    w_er["
    fa:fa-download ER \n
    "]

    r_erp["
    fa:fa-address-book ERP \n
    fa:fa-circle erp-validity \n
    fa:fa-database Cred(ER)
    "]

    r_ora_usd["
    fa:fa-address-book UsdOra \n
    fa:fa-circle oracle-nft \n
    fa:fa-database Oracle?
    "]
```

```
    r_ora_oby["
    fa:fa-address-book ObyOra \n
    fa:fa-circle oracle-nft \n
    fa:fa-database Oracle-oby
    "]

    m["tx"]

    o_treasury["
    fa:fa-address-book Main\n
    fa:fa-circle oby \n
    fa:fa-database Treasury
    "]

    o_user["
    fa:fa-address-book User\n
    fa:fa-circle ada \n
    "]

    i_position & i_treasury_0 & i_treasury_1 -->|Data| m -->
o_treasury & o_user
    w_m -.->|Lwc| m
    m_m -.->|Data| m
    w_er -.->|Er2Red| m
    r_erp & r_ora_usd & r_ora_oby .-o m
```

`tx.tidy`

A user small treasury utxos into a single one. This is useful when Lwc is called on very large positions. There is a small ADA reward for performing the tx, provided there are enough inputs (from surplus min ADA).

```
mermaid
flowchart LR
    i_treasury_0["
    fa:fa-address-book M\n
    fa:fa-circle oby \n
    fa:fa-database Treasury
    "]

    i_treasury_1["
    fa:fa-address-book M\n
    fa:fa-circle oby \n
    fa:fa-database Treasury
    "]

    i_treasury_n["
    fa:fa-address-book M\n
    fa:fa-circle oby \n
    fa:fa-database Treasury
    "]

    w_m["
    fa:fa-download M \n
    "]

    m["tx"]

    o_treasury["
    fa:fa-address-book Main\n
    fa:fa-circle oby \n
    fa:fa-database Treasury
    "]

    o_user["
    fa:fa-address-book User\n
    fa:fa-circle ada \n
    "]

    i_treasury_0 & i_treasury_1 & i_treasury_n -->|Data| m
--> o_treasury & o_user
```

```
    w_m -.->|Tidy| m
```

`tx.burn_auth`

This final tx is to accommodate burn auth tokens for liquidated positions.

```
mermaid
flowchart LR
    i_user["
    fa:fa-address-book User\n
    fa:fa-circle ada, user-auth-token \n
    "]

    w_m["
    fa:fa-download M \n
    "]

    m_m["
    fa:fa-circle M \n
    "]

    m["tx"]

    o_user["
    fa:fa-address-book User\n
    fa:fa-circle ada \n
    "]

    i_user -->|Data| m --> o_user
    w_m -.->|Burn| m
    m_m -.->|Data| m
```

VII. Notes

Risks and Weaknesses

The protocol includes certain risks and trade-offs, particularly in its reliance on oracles (external data providers for exchange rates).

- **Oracle Vulnerabilities**:
  Oracles, by their nature, are a potential point of compromise. They operate off-chain and are subject to issues such as downtime, data inaccuracies, or manipulation.
- **Mitigation via Updatable Scripts**:
  To address these risks, the protocol uses a separate, updatable script to manage oracle integration. This approach mitigates the risk of relying on a single, static oracle by enabling updates or replacements when an oracle becomes unreliable or defunct.
- **Trade-off**:
  While this design resolves some vulnerabilities, it introduces another: the potential for a malicious admin to replace the oracle with one that provides fraudulent data.

On balance, this updatable mechanism is considered necessary to ensure the protocol's long-term functionality and adaptability. However, it highlights the importance of robust governance and oversight in the administration of the protocol.

VIII. Appendix

Monolith to Composite

The initial design of the dApp relied on a monolithic spend-mint multipurpose validator. This single validator handled multiple transaction types, which led to challenges, especially when dealing with multiple script UTXOs (unspent transaction outputs) in a single transaction. One such challenge was the risk of **double satisfaction attacks** (where a single input satisfies conditions multiple times, potentially causing inconsistencies).

To address this, several solutions were considered:

1. **Single Script UTXO Constraint**:

   The simplest solution was to enforce a "one script UTXO per transaction" rule wherever possible. For most transaction types—such as adjust, close, or liquidate—this approach was acceptable from a user experience (UX) standpoint.

   - **Trade-off**:

     This constraint prevents certain operations, such as simultaneously adjusting multiple positions in a single transaction. However, this use case was deemed too niche to justify the added complexity of alternative solutions.

2. **Leader-Follower UTXO Model**:

   Another option was to designate one UTXO as the **leader** to manage transaction logic, while others acted as **followers**.

- **Implementation**:
  - A leader UTXO would handle the main logic.
  - Follower UTXOs would reference the leader and execute subordinate logic.
- **Challenges**:

  Each follower must identify its leader, and the leader must reference all followers. This method introduces significant redundancy, as similar computations are performed across multiple script executions.

3. **Withdraw Mechanism**:

   A third approach involved using the **withdraw script purpose** (a type of smart contract execution mode). Unlike the spend purpose, which is executed for every script UTXO involved, the withdraw purpose runs only once per transaction validation.

   - **Advantages**:

     This approach reduces redundant computations and mitigates the risk of double satisfaction attacks.

     It also simplifies complex transactions, such as **liquidations with compensation** (LWC, involving treasury UTXOs) or standard liquidations.
   - **Further Benefits**:

     Liquidations with compensation are conceptually similar to standard liquidations, differing mainly in their inclusion of treasury logic. Likewise, a liquidation resembles a position close (performed below MCR and without an auth token), and a

close resembles an adjustment where both collateral and debt are reduced to zero.

- **Outcome**:
  These similarities made it logical to shift all spend logic to the withdraw mechanism, consolidating the system's operations.

To Multi-Intents and Back Again

The withdraw mechanism also introduced the possibility of **multi-intent transactions**, where users could perform multiple actions within a single transaction. For example:

- Opening multiple positions simultaneously.
- Adjusting one position while liquidating another.
- **Feasibility**:
  While technically achievable, this design significantly increased script complexity. The additional layers of logic made the system harder to comprehend, validate, and maintain.
- **Decision**:
  Although multi-intent transactions offered enhanced flexibility, their complexity outweighed the benefits. The design reverted to simpler, single-purpose transactions to prioritize clarity and security.

In summary, the protocol evolved from a monolithic design to a composite, withdraw-based system. This approach balanced functionality, security, and maintainability while minimizing unnecessary complexity. However, the exploration of multi-intent capabilities highlighted the trade-offs inherent in designing for both flexibility and simplicity.

Glossary

- **Auth Token**:

  A token minted during position creation, held by the user, which grants ownership and control over the position.
- **CCR (Critical Collateral Ratio)**:

  A lower threshold than MCR. When a position's collateral-to-debt ratio falls below CCR, it becomes uneconomical to liquidate, requiring compensation from the treasury to incentivize liquidators.
- **Collateral**:

  ADA locked in a script address to secure a position and enable minting of OBYUSD.
- **LWC (Liquidation with Compensation)**:

  A liquidation scenario where positions below CCR are closed with additional compensation provided to liquidators from the treasury.
- **MCR (Minimum Collateral Ratio)**:

  The minimum ratio of collateral (ADA) to debt (OBYUSD) required to maintain a healthy position. Falling below this ratio makes the position eligible for liquidation.

- **OBY**:

  The protocol's native utility token, used for paying transaction fees and supporting system operations.
- **OBYUSD**:

  A stablecoin pegged to the US dollar, minted by locking ADA as collateral within the protocol.
- **Oracle**:

  An external service that provides off-chain data (such as ADA-USD exchange rates) to the protocol. Oracles are critical for accurate calculations within the dApp.
- **Redeemer**:

  Input data provided to a validator script during transaction execution. Redeemers specify user intent and provide information necessary for the script to validate the transaction. For more info see here.
- **Script Address**:

  A blockchain address controlled by a validator script, rather than an individual or entity. Funds sent to this address are governed by the script's rules.
- **UTXO (Unspent Transaction Output)**:

  A data structure that represents unspent funds on the blockchain. UTXOs are consumed as inputs and generated as outputs in transactions.

- **Validity Token**:

  A token minted during position creation, stored with the collateral in the position's UTXO, which acts as a counterpart to the auth token.
- **Validator**:

  A Plutus script that defines the conditions required to spend funds locked at its address or to mint/burn assets tied to a specific PolicyID. Validators enforce the rules of the protocol on-chain. For more info see here.
- **Withdraw Purpose**:

  A transaction validation mode in Plutus scripts that executes once per transaction, simplifying complex logic and reducing redundant computations.

Documentation Issues

- Mermaidjs bug: direction of subgraphs untameable