INTRODUCTION

Composable derivatives

What is Elektro?

Elektro is a non-custodial derivatives trading and settlements protocol. At its core, Elektro is a technical framework that will allow users to create fully collateralized, composable derivatives. The Elektro protocol leverages Frequent Batch Auctions (FBAs). In FBAs, time is treated as discrete instead of continuous, and orders are processed in auction batches instead of serially. This eliminates front-running and creates a uniform clearing price. This isn't the cool part.

By utilizing FBAs to clear derivatives, we can **decompose the clearing of financial contracts into packages of 'atomic' instruments**, using **put-call parity.**

Market Disruption

Traditionally/currently, derivative instruments on the same underlying clear in independent markets, usually, not only in digital assets, on different exchanges, with all the concomitant operational, counterparty, funding risk implications. Margin loan and funding markets are also operating independently. Liquidity across these markets is fragmented by definition and gets delivered by market participants that extract the value associated with providing these liquidity pooling services across instruments, venues and market structure arrangements.

Specifically:

1. Traders, arbitrageurs, market makers broadly, enforce the linkages between independent markets

2. Traders of listed derivatives, funding repo, 1 delta products extract revenue associated with spot vs vanilla options delta hedging at inception, lifetime, expiry as well as creating implied 'OTC equivalent' markets on funding rates, collateral posting and collateral swaps

3. Structured products and associated structuring efforts and intermediaries managing internally option path dependence and binary/barrier risk and earning the associated high margins

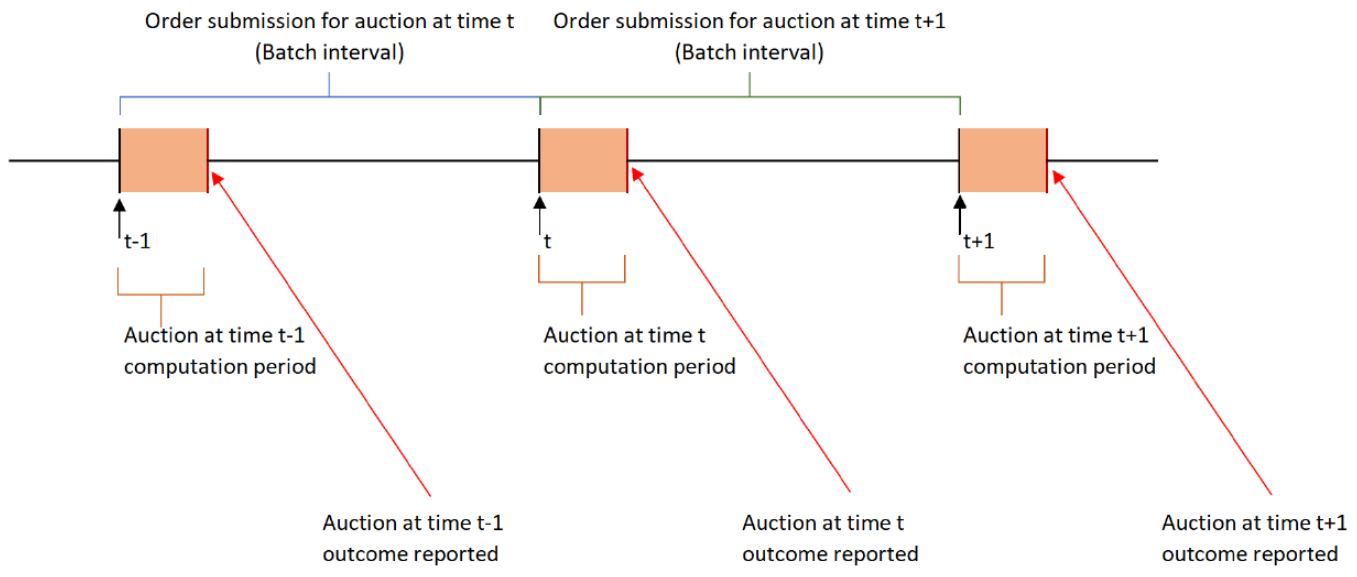
Elektro is disrupting the web of interlocking ad hoc, haphazard, lucrative (but also risky for the undertakers) constraints to trading activity, by embedding the financial logic associated with the activities onto the rules governing matching within the protocol itself.

OVERVIEW

Frequent Batch Auctions (FBA)

The auction process

Elektro uses a Frequent Batch Auction (FBA) mechanism to match orders.
FBAs run across multiple orderbooks with different products, underlyings, maturities; which allows for the seamless handling as well as synchronized execution of multi-leg linked / conditional orders. Such design ensures that ECE features High-Frequency-Trading (HFT) resistance that is thought of as creating a fairer level playing field for all participants.



The FBA process is similar to traditional auction models implemented in many marketplaces. The main difference to traditional auction processes is the duration of each auction cycle. Frequent Batch Auction cycles are very short and are immediately followed by the next FBA cycle.

Order Submission Period

During the Order Submission Period of an FBA, new orders can be submitted into the order book, existing orders can be canceled, however no orders are being matched and no match prices are being generated.

During the Order Submission Period, only limited market data about the state of the order pool is being disseminated including indications for current bid and offer side sizes including volume stemming from conditional orders.

Matching (Computation) Period

At the end of the Order Submission Period, the Matching Period starts. At this point new orders are not accepted into the current auction cycle any more but are instead placed into a temporary queue outside the orderbook until the next Order Submission Period opens. During the Matching Phase, the Elektro Clearing Engine (ECE) will evaluate all live Buy and Sell orders including conditional orders. At the conclusion of the Matching Period the Matching Engine determines a single Match-Price per each contract.

Elektro Clearing Engine (ECE)

The Elektro Clearing Engine is a multi-product cross-orderbook matching engine

ECE has 3 key elements.

1) **Replication** - ECE decomposes financial contracts into packages of 'atomic' instruments using put-call parity relationships. Therefore, orders are matched at the atomic instrument level.

2) **Conditional orders** - 'Synthetic' orders are formed by replication principles while the matching engine algorithm ensures simultaneous consistent execution.

3) **Mixed Integer Programming (MIP) Optimization -** MIP allows searching for clearing prices and associated sets of consistent orders satisfying them that maximize executed volume and satisfy best execution requirements.

The Elektro Clearing Engine (ECE) is composed of discrete-time (periodic), uniform-price (one price per instrument per auction), double (orders submitted by buyers and sellers) auctions that allow for the pooling of orders for different underlying instruments and payoffs and thus the utilization of multiple different sources of liquidity that normally clear in isolation from each other. The ECE, therefore, constitutes the foundation of a marketplace in which growth and liquidity generation are supported by ECE-intrinsic, multiple-sided network effects.

In the standard continuous order book methodology, price discovery for each instrument happens in the context of each order book as a silo, with "cross order book" or "cross instrument" liquidity delivered haphazardly and, on an ad-hoc basis. The ECE is a mechanism for harnessing, in a systematic fashion, all market making tools and making them accessible to every market participant.

Buy and Sell orders, including orders for pre-packaged and structured products and conditional orders are matched based on the principles of maximum matched volume and minimum surplus across all orderbooks covered. If several possible sets of match prices exist that would result in the same amount of total matched volume and the same volume of surplus across all orderbooks, the set of match prices within the group of possible sets that results in the smallest total price deviation to the previous set of reference prices across all orderbooks will be the determined set of match prices.

Once the set of match prices for all contracts is being determined, all orders that are being allocated a fill at the determined prices are filled accordingly and are either removed from the orderbook or their live outstanding size updated.

If there are surpluses of match-able quantity on one side of an orderbook, all orders on that side are partially filled according to their original order size relative to the overall executable quantity. Fully executed orders are removed from the Orderbook at the end of the Matching Phase. Partially filled orders are automatically updated in the Orderbook to appropriately reflect the new and remaining size and state of the order.

1 Overview

The MIP approach simultaneously looks for clearing prices and a set of orders satisfying them, which clear fully against each other (i.e. there are no unmatched tokens), while guaranteeing maximal volume execution, and further best execution constraints. In particular, it does this in the presence of conditional orders and with automatic replication.

1.1 Challenge

Clearing a single book is conceptually straightforward - regular exchanges do it all the time. The difference in the Elektro set-up is the presence of Conditional Orders - orders on more than one token, which execute conditional on the net price of the tokens, and conditional on the presence of other orders to clear against. Suddenly, the clearing of one book depends on another — a conditional order may provide a competitively-priced fill, but only if its other leg is also filled in another book, etc.

Given prices, it is fairly straightforward to determine a set of clearing orders that maximizes volume. Conversely, given a set of clearing orders, it is straightforward to find a price that satisfies them. What is difficult is to do both at the same time.

To overcome this difficulty, Mixed Integer (Linear) Programming is used.

1.2 Mixed Integer Programming (MIP)

MIP is an optimisation technique, solving Linear Programming problems, but with the additional stipulation that some variables must be integers, or even just 0 or 1. Such binary variables can be used to encode logical decisions, such as "if price is greater than 100, this order cannot be filled". This document is a nice introduction to some of the possibilities this opens.

2 Formulating the clearing problem as MIP

At the most general way, the clearing problem can be specified as follows:

$$\text{Maximise} \sum_j x_j \tag{1}$$

$$\text{subject to} \sum_j w_j x_j = 0 \tag{2}$$

$$i\text{th order consistent with prices } p : w_j \cdot p \leq l_j \forall j \text{ or } x_j = 0 \tag{3}$$

$$0 \leq x_j \leq X_j \tag{4}$$

where

1. $x_j \in \mathbb{R}, x \geq 0$ is the amount filled in the ith order
2. $w_j \in \mathbb{R}^n$ is a vector describing which of n possible tokens, and in what sign and proportion, are acquired, when a unit amount of the order is executed. For example, if $w_j = [1, 0.5, 0, -1]$, executing a unit amount of the order will give its owner 1 lot of token 1, 0.5 lot of token 2 and -1 lot of token 4, with nothing in token 3.
3. X_j is the overall size of the ith order, and the maximal executable amount.

Without condition (3), this would be a linear program, however we must also search over prices, and in turn eligibility of the orders for clearing.

Assuming for a moment we can solve this problem via some kind of iterative approach, we will find that for any price vector, there are various orders satisfying (3). An iterative solver would try to maximize the *xj* filled amounts, subject to the clearing condition. But then it would also try to adjust the prices *p*, to come up with a better subset of orders. This sounds computationally infeasible.

Fortunately, this kind of condition is readily expressible in the MIP framework. Consider the following inequality:

$$x - y - Mi \leq 0 \tag{5}$$

$$x, y \in \mathbb{R}, i \in \{0, 1\} \tag{6}$$

where *M* is a large parameter. If *i = 0*, the inequality reduces to *x ≤ y*, but if *i = 1* and *M* is sufficiently large, the inequality is automatically satisfied, and the condition specifies no relationship between *x* and *y*

The problem above can thus be augmented as follows:

$$\text{Maximise} \sum_j x_j \tag{7}$$

$$\text{subject to} \sum_j w_j x_j = 0 \tag{8}$$

$$w_j \cdot p - l_j + A_j i_j \geq 0 \; \forall j \tag{9}$$

$$w_j \cdot p - l_j + B_j(1 - i_j) \leq 0 \; \forall j \tag{10}$$

$$i_j \in 0, 1 \; \forall j \tag{11}$$

$$0 \leq x_j \leq X_j \tag{12}$$

$$x_j \leq i_j X_j \tag{13}$$

Note that all inequalities are linear, so we satisfy the requirements of linearity. Further, conditions (9) and (10) imply that *wj · p ≤ lj ⇐⇒ ij = 1*. Indeed, for carefully chosen values of A and B constraints, if *wj · p ≤ lj* , then (9) implies *ij = 1* (otherwise the inequality cannot be satisfied). Conversely, if *ij = 1*, inequality (10) reduces to *wj · p − lj ≤ 0.* Finally, inequality (13) implies that if *ij = 0*, the associated traded quantity *xj* must also be *0*.

Consequently, the following are true:

1. The optimisation program is a linear, mixed-integer program, and readily solvable with off-the-shelf solvers
2. The optimisation problem also encodes (part of) the Elektro rules for clearing books: choose prices and orders satisfying them, such that the orders clear, and such that the overall crossed volume is maximized
3. Thanks to the formulation, the order selection and price determination happen automatically. The search over prices and order sets occurs inside the MIP solver, in a computationally-efficient way.
4. A regular linear program would not be sufficient here; if the variables *ij* are allowed to take fractional values between *0* and *1*, the logic encoding no longer works.

3 Full problem specification

In this section, I describe in full how the problem is specified in the solver. It is conceptually the same as the example above, but has some further features.

The clearing process consists of solving a sequence of closely related optimisation processes. This is because the clearing process in fact maximizes a number of objectives in sequence: first maximize volume, then minimize surplus subject to maximizing volume, then maximize price surplus subject to keeping the other quantities constant. Finally, select a single price satisfying a final disambiguation criterion. The first three problems are MIP problems, the last one is a Quadratic Program (QP), with possibly simpler formulations being possible.
Overall, the problem is split logically as follows:

1. Order data is pre-processed
2. MIP problem is solved to maximize volume
3. MIP problem is solved to minimize surplus, subject to keeping volume constant
4. MIP problem is solved to maximize price surplus, subject to keeping volume and surplus constant
5. QP is solved to choose final price

3.1 Data pre-processing

The following steps are taken:

1. Orders are grouped by side, tokens, and relative weights of the tokens. The following are examples of orders that would get grouped:

- All buy orders for a single token *X*
- All conditional orders to buy *1 X*, sell *1 Y*
- Conditional order to sell *1 X*, buy *1 Y*, and conditional order to sell *10 X*, buy *10 Y*

If orders are on opposite sides, or acquire different proportions of tokens, they are not grouped. Each group has weights w associated with it, describing what tokens are being traded in the group, normalized so that

$$\sum_i |w_i| = 1.$$

For the above examples, the weights will be *(1), (0.5, −0.5), (−0.5, 0.5)*. I refer to these associations as groups.

2. Within a group, I order and sub-group orders by price, from most aggressive to least aggressive. A sub-group of orders in the same group, and with the same price, is referred to as an *interval*. Each interval has a quantity *q* associated with it, which the maximal amount of tokens that can be filled at this price. The price condition is expressed as

$$w \cdot p \leq l$$

Examples:

- For a buy order for at most 100: $(1) \cdot (p) \leq 100.$
- For a sell order for at least 150: $(-1) \cdot (p) \leq -150.$
- For a conditional order to buy *0.5 X, sell 0.5 Y*, receive *10* premium: $(0.5, -0.5) \cdot (pX, pY) \leq -10$

3. Quantities *A* and *B* are also calculated for each interval, to be defined later.

3.2 MIP problems

The three MIP problems share a common structure, and differ by objective function, and minor internal details.

3.2.1 Common setup

The common part of the setup goes as follows:

1. For *k*th token, introduce a price variable *pk* ∈ *R* with an upper and lower bound, *Lk* and *Uk*

2. For *i*th group, generate a quantity variable *xi* ≥ *0*. This denotes the overall number of real tokens filled in this group.

3. Introduce the clearing constraint. All tokens must clear fully. In other words

$$\sum_i x_i w_i = 0$$

where *wi* are the weights of the *ith* group. This is a vector constraint.

4. For *j*th interval of the *i*th group, introduce a breakpoint variable *bi,j* ∈ *{0, 1}*. This variable is associated with the orders belonging to the interval. If *b = 0*, no orders from this interval are executed.

This gives rise to a number of consistency conditions:

(a) Breakpoint variables must satisfy the inequality:

$$b_{i,1} \geq b_{i,2} \geq b_{i,3} \geq \dots$$

If for some *j, bi,j = 0, bi,j+1 = 1*, that implies orders in group *j* are all unfilled, but some orders in group *j + 1* may be filled, which contradicts the price priority conditions.

(b) If *bi,j = 1*, the price must be consistent with the interval's price inequality. This is expressed with the following inequality:

$$w_{i,j} \cdot p - B_{i,j}(1 - b_{i,j}) \leq l_{i,j}$$

This way, if *b = 1*, this inequality is equivalent to *wi,j · p ≤ li,j* , as defined for the interval. If *b = 0*, *Bi,j* (defined below) is large enough that the inequality *wi,j · p − Bi,j ≤ li,j* is large enough that for any value of *p* within its bounds, the inequality is satisfied — i.e. the constraint becomes trivial.

(c) Similarly, if *bi,j = 0*, we stipulate that the inequality must not be satisfied. This way we achieve that *b = 0* ⇐⇒ interval inequality holds. This is done via:

$$w_{i,j} \cdot p + A_{i,j} b_{i,j} \geq l_{i,j}$$

When *b = 0*, we get the opposite condition to the interval's price inequality. Meanwhile, if *b = 1*, the extra *A* term makes the inequality trivially satisfied.

These inequalities are at the core of the MIP matching engine logic. They create a structure of binary variables associated with intervals. I use them to constrain the filled quantities, calculate surplus and maximize price premium subsequently.

This is done as follows:

(a) Quantity constraining:

$$x_i \leq \sum_j b_{i,j} q_{i,j}$$

with *qi,j* being the interval quantities.

(b) Surplus computation: quantity available to trade is expressed as

$$\sum_{i,j} b_{i,j} q_{i,j}$$

The difference between this and volume is the surplus (In fact, when surplus is being minimized, volume is being held constant, so it suffices to minimize the quantity of orders available to trade).

(c) Price premium is covered in later parts of this document.

5. The above does not specify the objective functions; indeed, these vary depending on the type of problem being solved. When maximizing volume, the above problem is taken, with an objective of maximizing volume =

$$\textstyle\sum_i x_i$$

When surplus is being minimized, the surplus variable is being minimized, with the constraint that the previous value of volume must hold, etc. The common structure of the problem guarantees that any solution produced by the MIP solver satisfies the basic rules of clearing.

It may help to consider some stylised scenarios around point 4 of the above structure.
Maximizing volume When maximizing volume, the solver wishes to make the *xi* variables large, subject to the clearing constraint. Since *xi* P is bound by

$$\textstyle\sum_j q_{i,j} b_{i,j}$$

Increasing values of *xi* force more and more *bi,j* values to set to *1*. This in turn forces the movement of the price variable *p* via the set of inequalities in point 4 above.

Minimizing surplus When volume is fixed, minimizing surplus is equivalent to minimizing

$$\sum_{i,j} q_{i,j} b_{i,j}$$

i.e. setting as many *bi,j* to *0* as possible. This in turn adds increasingly strict inequalities to *p*, confining it to a smaller region of the solution.

3.2.2 Price premium

The final objective is maximizing price premium. For an order with price defined by the inequality

$$w \cdot p \leq l, \sum_k |w_k| = 1$$

Price premium of an order is defined simply as

$$l - w \cdot p$$

It is clearly non-negative for fillable orders, and it is used for ordering fills by overall priority; orders with higher price premium have more aggressive prices and have priority when receiving fills, the same way as a more aggressively priced order has priority in a standard, single-instrument auction.

Priority optimization schemes First, suppose each interval has priority *Pi,j* ; we will use price premium to define it. But how is it even included in the optimisation in the first place?

Suppose you know the maximal volume and minimum surplus. Further, let *yi,j* be the amount filled specific to interval *i, j*. We then have

$$x_i = \sum_j y_{i,j}$$

Further, we can maximize order priority by maximizing:

$$\sum_{i,j} P_{i,j} y_{i,j}$$

subject to maintaining the same volume and surplus. If two intervals, *i, j* and *k, m* are competing for the same fill, the one with the higher priority will get it.

Complications There are two complications: first, we have not defined the quantities *yi,j* in the problem, and second, the priorities *P* in our case depend on the prices, which seemingly invalidates the linear nature of the problem. An expression like *P ·y* expands to *(l−w·p)·y*, leading to a second-order expression.

Vanishing price coefficients Fortunately, the price coefficients cancel out. We have from the clearing condition that

$$\sum_i w_i x_i = \sum_{i,j} w_i y_{i,j} = 0$$

therefore also

$$\begin{aligned}\sum_{i,j}(l_{i,j} - w_i \cdot p) \cdot y_{i,j} &= \sum_{i,j} l_{i,j} y_{i,j} - p \cdot \sum_{i,j} w_i y_{i,j} \\ &= \sum_{i,j} l_{i,j} y_{i,j} - p \cdot 0 \\ &= \sum_{i,j} l_{i,j} y_{i,j}\end{aligned}$$

which is again nicely linear. This means that in fact, price premium can be maximized without knowing the actual prices, it suffices to maximize this expression within the MIP problem framework.

One concern could be whether the final selection of orders is consistent with the prices — could it be that by ignoring the clearing prices in the optimisation, the final selection of orders violates the price conditions. However, by the structure imposed on the problem above, we know that this cannot happen; if an interval does not satisfy the prices, its indicator variable is set to 0 and the quantity for that interval is forced to be 0. Conversely, if a fill for an interval is positive, the indicator variable must be positive too, forcing the price to be consistent with it.

Interval fills The problem specification, for simplicity, did not include the *yi,j* variables, however they are easy to add. We add one for each interval and stipulate that

$$\sum_j y_{i,j} = x_i$$

Further, we need to ensure that fills are accrued in order; an interval can only receive fills if all of its predecessor intervals are fully filled.

This can be established by the following inequalities:

$$\begin{aligned} y_{i,j} &\leq q_{i,j} b_{i,j} \\ y_{i,j} &\geq q_{i,j} b_{i,j+1} \end{aligned}$$

This way, if an interval's *b* variable is *0*, its associated fill is also *0*. If the next interval's *b* variable is *1*, then the current interval must be fully filled. The variable is only free if *bi,j = 1 and bi,j+1 = 0.*

3.2.3 Optimisation objectives

The previous section outlines the structure of the problem, it also needs some optimisation objectives. Three problems are solved in sequence:

Maximize volume Optimisation objective is max

$$\sum_i x_i$$

Minimize surplus Optimisation objective is min

$$\sum_{i,j} b_{i,j} q_{i,j}$$

while keeping volume constant.

Maximize price premium Optimisation objective is max

$$\sum_{i,j} y_{i,j} l_{i,j}$$

3.3 Final price determination

Solving the three rounds of MIP problems yields final fills, and various constraints on prices. It does not necessarily narrow down on a unique price; the simplest example would be a market with two orders:

- Buy *X* for 150
- Sell *X* for 100

In single-instrument auctions, typically the price is chosen to minimize distance from a previous price (plus some fairly arbitrary condition if there is no last price, for example choosing minimal price, or mid-price of possible values). Here, this is more complicated, since there are potentially conditional orders setting multivariate bounds on order prices.

The analogy therefore is to minimize distance between new and previous price over all instruments simultaneously, subject to constraints from the last MIP stage.

3.3.1 Constraints

At this stage, we know which intervals have fills, and which ones do not. We know the orders with fills must satisfy the final price, and the orders without fills should not satisfy the price, to minimize price surplus.
More specifically, if for some *i, j , bi,j = 1* then *wi · p ≤ li,j* and if *bi,j = 0* then in fact *wi · p > li,j* . The *b* variables are now fixed, so there is no need to solve this with a MIP solver.

3.3.2 Objective

The natural distance function would be the *l1* metric:

$$\min \sum_k |p_k^{\text{prev}} - p_k|$$

However, it may not be sufficient to disambiguate prices in this context. It is likely that conditional orders will introduce conditions like *px−pY ≤ C* for some tokens *X, Y* and some constant *C*; then there may be different valid prices *p* with the same objective value; this is demonstrated on figure 1 on page 10.

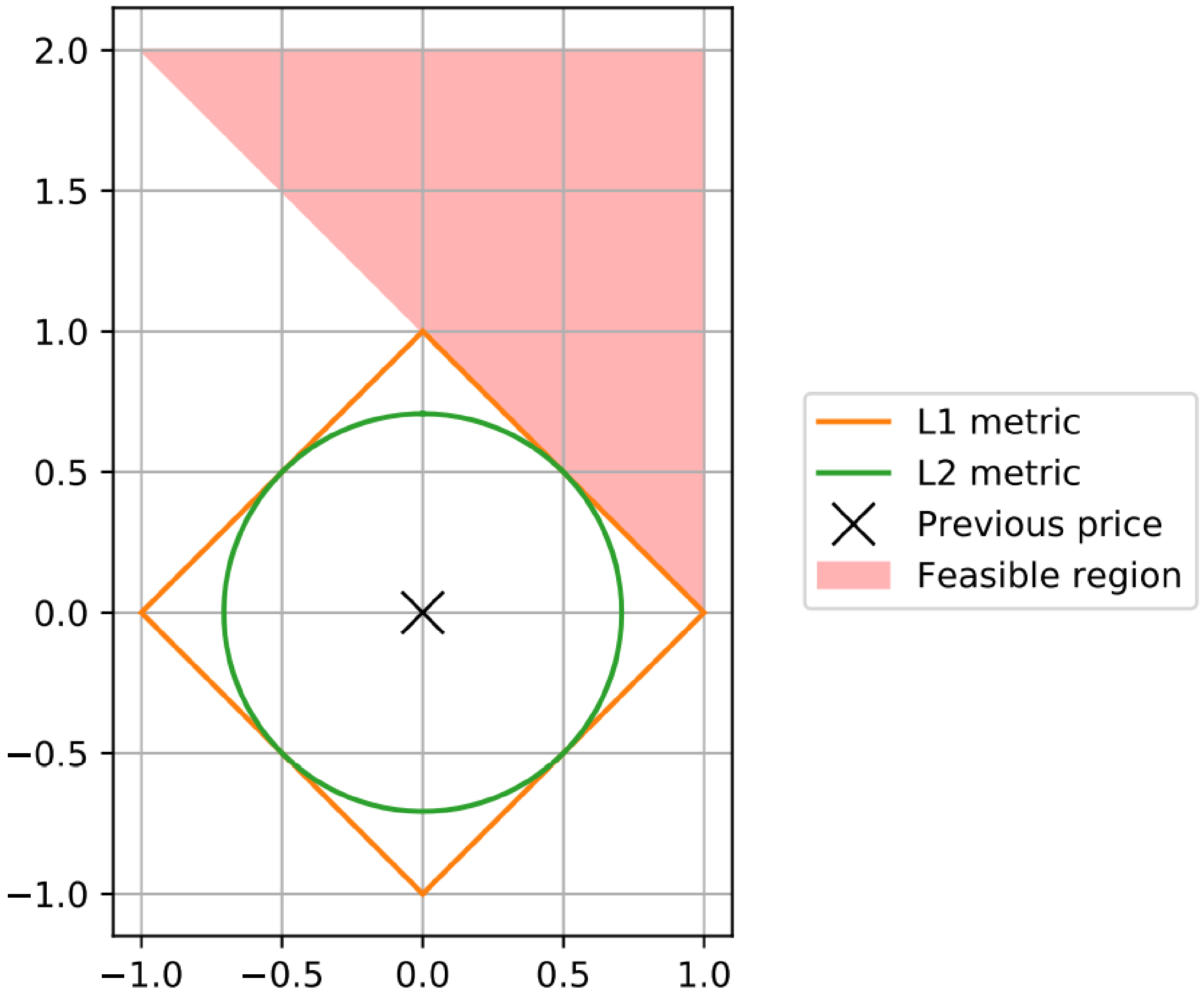


Figure 1: *L1* and *L2* metrics comparison. The square and circle represents points equidistant from previous price according to *L1* and *L2* metrics. The *L1* metric yields many equidistant points in the feasible region, whereas the *L2* metric only returns *1*.
For this reason, I instead propose to use the *l2* metric:

$$\min \sum_k (p_k^{\text{prev}} - p_k)^2$$

While admittedly the *l2* metric does not have an intuitive meaning in this context, its properties should make the final price much less ambiguous.

3.4 Quadratic solver

This problem now needs to be solved using a quadratic solver; one good option is OSQP. For numerical stability, it is good to express *p* as

$$p = p^{\text{prev}} + \delta p$$

then convert the inequalities on *p* into ones on *δp* and minimize *l2(δp).*

4 Replication

A key design idea is that orders in different tokens can be replicated into each other; the classic example is put-call parity matching, whereby being long a call option and short a put option with the same strike is equivalent to being long a forward — so it is possible to match a buyer of a call, seller of a put and buyer of a forward, even though all three want different instruments. The approach we adapt is as follows:

1. Identify a minimum set of atomic instruments that allow all tokens to be represented. For example, calls and forwards are sufficient to represent puts, so puts are not included in this set.
2. Convert all tokens in an order to their atomic representations, and treat them as conditional orders with a net cost/premium condition. For example, an order to buy a put becomes an order to sell a forward and buy a call, with the same overall price.
3. Clear these orders as usual (with a minor modification to volume counting, see below) and allocate fills.
4. Calculate prices of replicated instruments.

Minting of the smart contracts may be involved around that — it needs to deal with e.g. clients who wanted a put option but received a call+forward, but it is beyond the scope of the matching engine. The engine focuses on enforcing the rules that then make it possible to mint the smart contracts. This section focuses on the replication rules.

4.1 Replication mechanics

Any token can either be atomic (non-replicable), meaning it has no other representation, or replicable, meaning it is in fact represented by other tokens inside the MIP matching engine.

Any replicable token defines its immediate replication only. The tokens it is replicated into may themselves require replication. However, since every token is replicated into simpler tokens, iteratively repeating the replication eventually yields an atomic representation.

Some replications require the smart contract to lock in cash in the numeraire asset — for example, a binary put is represented as a constant payoff (cash) less a binary call option with the same strike. For this purpose, we introduce a synthetic **ForwardCash** token. It represents cash, and costs its own face value. There is no need to enter it into the MIP engine: its price is known, and it is supplied naturally when trading contracts. It is however useful for accounting for the cash flows in the smart contracts.

4.2 Unique basis representation

By stipulating that a minimum set of instruments is used, we guarantee that no replications can be missed by the MIP engine (They may still not execute because of the price conditions). Let us assume the chosen instruments form a proper basis of the vector space of possible orders (which is equivalent to saying that each instrument has a unique representation). Suppose one portfolio of tokens

$$\sum_i w_i T_i$$

has the same payoff as another

$$\sum_i v_i T_i$$

where *wi* , *vi* are some weights and *Ti* are atomic tokens. Then we have:

$$\begin{aligned} \sum_i w_i T_i - \sum_i v_i T_i &= 0 \\ \sum_i (w_i - v_i) T_i &= 0 \\ w_i - v_i &= 0 \ \forall i && \text{by the basis property} \end{aligned}$$

so in fact they must have the same representation in weights *w* and *v*.

4.3 Volume counting

One artifact this introduces is that replication changes the number of contracts traded: buyer of a put wants to buy 1 token, but gets replicated into a call and forward (2 tokens). A consequence would be that puts get matched at higher priority than calls, since then *put+put* is 2 fills, whereas *call+call* is 1 fill.

To resolve this, orders' total volume is counted from the original order, not the replicated order. MIP problem is modified to account for the 'declared' volume, and not the actual amount of tokens filled.

Mechanically, this is performed as follows:

1. Each interval counts both the maximal number of real tokens available to fill, as before, but also the total volume count that should be attributed to fully filling the interval. Call this quantity

$$q_{i,j}^{\text{req}}$$

2. Volume is counted as

$$\sum_{i,j} y_{i,j} q_{i,j}^{\text{req}} / q_{i,j}$$

instead of

$$\sum x_i$$

3. Similarly, surplus must now be calculated as

$$\sum_{i,j} b_{i,j} q_{i,j}^{\text{req}}$$

A Computing constants A, B

Constants *A, B* appear in the constraints as follows:

$$w \cdot p + Ab \geq l \tag{14}$$

$$w \cdot p - B(1-b) \leq l \tag{15}$$

for an order (interval) with weights *w*, interval binary variable *b* and limit price *l*. As discussed in section 3.2.1, constants *A, B* are chosen so that if *b = 1*, inequality *14* is trivially satisfied, and if *b = 0*, inequality *15* is trivially satisfied. The inequalities are trivially satisfied if *A* and *B* are large enough that for any valid combination of prices *p*, the inequalities still hold for an appropriate value of *b*, by virtue of the additional (large) term *A, B*, respectively. This way, *b* can be used to knock-in and knock-out inequality terms, as needed.

By rearranging, this yields the following constraints:

$$A \geq l - w \cdot p$$
$$A \geq \sup_p (l - w \cdot p) = l + \sup_p - w \cdot p$$

And

$$B \geq w \cdot p - l$$
$$B \geq \sup_p (w \cdot p - l) = \sup_p w \cdot p - l$$

The suprema are easy to calculate, given *w* is pointwise constrained: for all *k*,

$$L_k \leq p_k \leq U_k$$

Thus for any vector *v* (for example *v = ±w*)

$$\sup_p v \cdot p = \sum_k v_k \times \begin{cases} L_k & \text{if} v_k \leq 0 \\ U_k & \text{otherwise} \end{cases}$$

Applying this formula to *w* and *−w*, and increasing *A, B* by *1* for numerical stability, gives the following equations:

$$A = l + \sup_p - w \cdot p + 1 \tag{16}$$

$$B = \sup_p w \cdot p - l + 1 \tag{17}$$

as expressed in code as well.

Static Replication

What does it all mean?

No-Arbitrage Pricing

No-arbitrage pricing is the concept that a financial instrument should be priced as the sum of components that replicate its economic payoff. Put-call parity is one example. Put differently, two instruments with identical cash flows in all future states of the world must be priced equally today.

Dynamic Replication

The classic approach to hedging a derivative involves maintaining an ever-changing position in the underlying asset, also called dynamic delta hedging. According to Black-Scholes theory, a stock option behaves like a weighted portfolio of risky stock and riskless zero-coupon bonds. The Black-Scholes options formula tells you how to calculate the portfolio weights. They depend on the stock price level, the dividend yield, the stock volatility, the riskless interest rate and the time to expiration.

Put differently, this means you can in principle own a portfolio of stock and riskless bonds, and achieve exactly the same returns as the option. To do so, you must continuously adjust the weights in your portfolio according to the formula as time passes and/or the stock price moves. This portfolio is called the dynamic replicating portfolio. Options traders ordinarily hedge options by shorting the dynamic replicating portfolio against a long position in the option to eliminate all the risk related to stock price movement.

There are difficulties with this hedging method. First, continuous weight adjustment is impossible, and so traders adjust at discrete intervals (e.g. every few minutes/hours). This causes small errors that compound over the life of the option, and result in replication whose accuracy increases with the frequency of hedging. Second, there are transaction costs associated with adjusting the portfolio weights which grow with the frequency of adjustment and can overwhelm the profit margin of the option. Traders have to compromise between accuracy and cost.

Static Replication

Static replication, on the other hand, refers to hedging a position without needing to alter or adjust components with the passage of time. A static hedge may involve setting up a portfolio of simple European options that is guaranteed to match the payout of the instrument to be hedged. By leveraging Frequent Batch Auctions, Elektro introduces a way for derivatives to be replicated statically within the protocol, by decomposing them into atomic instruments. This has huge implications for liquidity.

What does this enable?

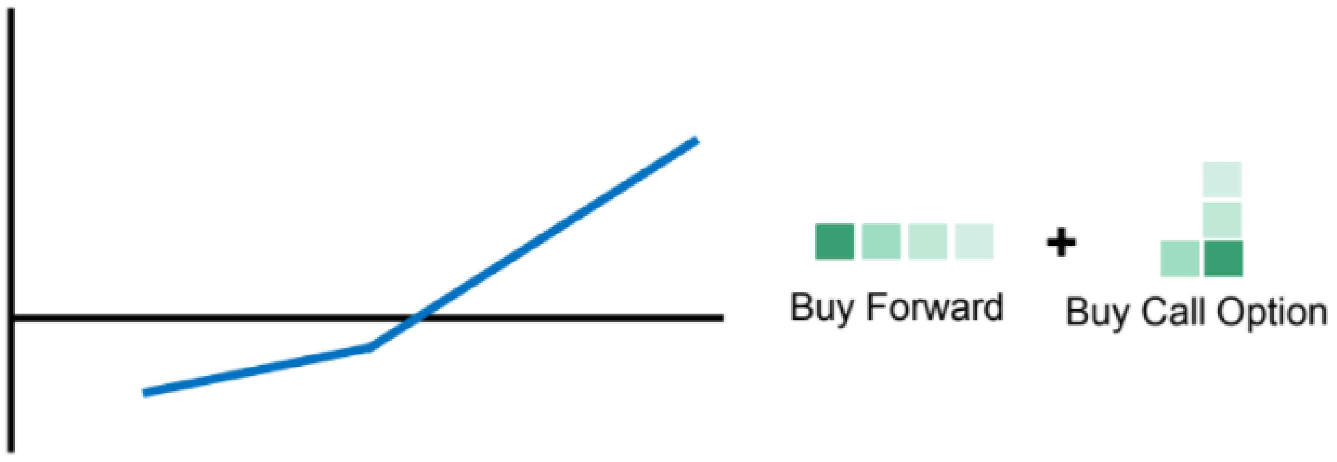
- Elektro allows for internally consistent ‘per auction forward’ to settle ‘within auction’ options and to trade options against delta.
- Ability to trade collateral swaps, spot/forward swaps, forward forward swaps as standard conditional orders.
- Organic ability to accommodate exponential growth inherent in order book combinatorics, by enforcing linear ‘ring wise’ relationships by pair-wise replication at the building block level.
- Embedded put-call parity relationships providing for liquidity bootstrapping by allowing for a variety of potential counterparties for each option trade (in its most simple form, a buyer of a call, is not just matched with a seller of a call, but also can be potentially matched against a seller of a funded put).
- Binary options being among the atomic instruments onto which all payoffs can be decomposed into allows for direct solution to the hedging approximation required for digital risk in markets (all of them) without that standard building block. This, in turn, implies the ability to include all statically replicable structured products as simple order entries into the auctions as they subsequently get perfectly replicated within the protocol into atomic instruments (path dependence as a direction of future research has already been explored with intriguing potential consequences).
- Direct option funding equivalence relationships directly incorporated into matching engine; box spreads, calls + puts vs forwards; variance swaps as multi legged conditional orders, etc
- Variance swaps as multi legged conditional orders
- Market fungible VAR based trigger margin loans that allow for market discovery of risky funding curves; a new huge market that has traditionally been outside the purview of exchanges

Statically replicated structured products

- Discount certificates/reverse convertibles
- Risk/reward ratio certificates
- Variance swaps
- Outperformance certificates
- Bonus certificates
- Twin-win certificates

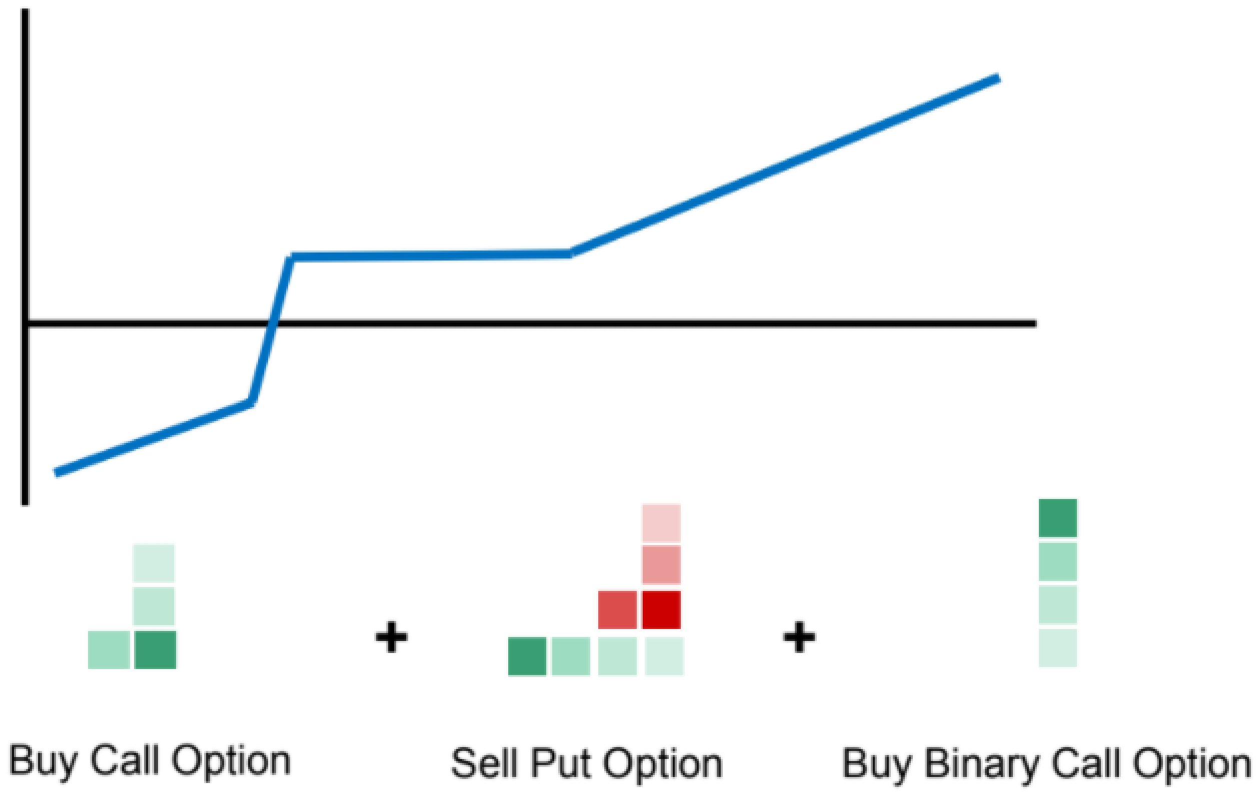
Example 1: Outperformance Certificates

Outperformance certificates enable investors to participate disproportionately in price advances in the underlying coin/pair if it rises higher than a specified threshold value.



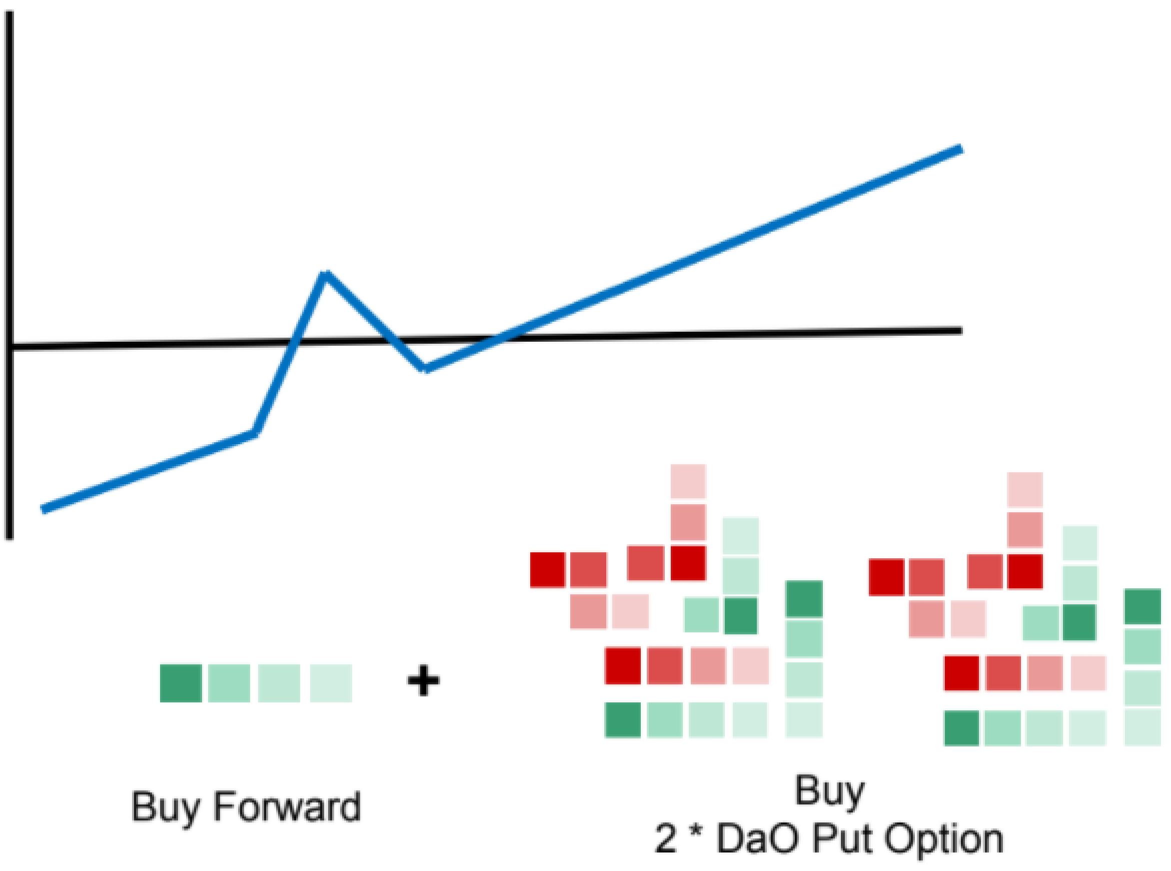
Example 2: Bonus Certificates

Bonus certificates are participation that feature full upside participation and a conditional capital protection as long as the underlying coin/pair doesn't cross a predefined threshold (barrier). Bonus certificates tend to perform well in both sideways and rising markets.



Example 3: Twin-win Certificates

Twin-win certificates generate a profit for the investor not only when the price of the underlying coin/pair goes up, but also if it declines to a certain extent.



Replication Process

How does the replication work?

Tradable products in the system include all derivatives that can be replicated by the Elektro risk sharing building blocks (RSBBs), which are comprised of a ‘ per-auction settled ‘ fully collateralized forward contract ('Elektro Cash'), a 'defined maturity 'fully collateralized futures contract' ('Elektro Forward'), a European call option (and therefore by put/call parity, put options), a binary contract (and therefore all European digital and barrier options), all payoffs that can be statically replicated by the Elektro RSBBs, and an American, VAR based trigger basket option ('Elektro margin loan')

In the replication process, orders are decomposed into orders on atomic instruments.

The designated list of atomic instruments is as follows:

Instrument	**Description**
Spot (C1C2)	Spot trade on underlying pair C1/C2
Forward (C1C2, T)	Forward trade on underlying pair C1/C2 with expiry T
Call (C1C2, K, T)	Call option on underlying pair C1/C2 with strike K and expiry T
BinaryCall (C1C2, K, T)	Binary call option on underlying pair C1/C2 with strike K and expiry T. Pays out if the underlying pair C1/C2 is equal to or greater than K at expiry.

ForwardCash (C1, T)	Pays out 1 unit of currency C1 at expiry T
MarginLoan (C1, R, T)	Margin loan on currency C1, bucket R and maturity T.

The replication tables (below) define the replication rules:

There is one replication defined for each instrument into the immediately simpler instruments. Replication is repeated until all instruments are atomic, giving the final, full replication.

For example:

A Down-and-In Put(K, B) is replicated into Put(B) + (K-B) Binary Put(B). Then, the Put(B) is replicated as Call(B) - Forward, and Binary Put(B) as ForwardCash – BinaryCall(B).

This is the final replication.

Contract	**Replication**
+1 Spot(currencyPair=C1C2)	+1 Spot(currencyPair=C1C2)
+1 Forward(currencyPair=C1C2, expiry)	+1 Forward(currencyPair=C1C2, expiry)
+1 Call(currencyPair=C1C2, strike=K, expiry)	+1 Call(currencyPair=C1C2, strike=K, expiry)
+1 Put(currencyPair=C1C2, strike=K, expiry)	+1 Call(currencyPair=C1C2, strike=K, expiry) -1 Forward(currencyPair=C1C2, expiry)
+1 BinaryCall(currencyPair=C1C2, strike=K, expiry)	+1 BinaryCall(currencyPair=C1C2, strike=K, expiry)
+1 BinaryPut(currencyPair=C1C2, strike=K, expiry)	+1 ForwardCash(currency=C2, expiry) -1 BinaryCall(currencyPair=C1C2, strike=K, expiry)
+1 CallSpread(currencyPair=C1C2, strike1=K1, strike2=K2, expiry)	+1 Call(currencyPair=C1C2, strike=K1, expiry) -1 Call(currencyPair=C1C2, strike=K2, expiry)
+1 PutSpread(currencyPair=C1C2, strike1=K1, strike2=K2, expiry)	+1 Put(currencyPair=C1C2, strike=K1, expiry) -1 Put(currencyPair=C1C2, strike=K2, expiry)
+1 UaI_Call(currencyPair=C1C2, strike=K, barrier=B, expiry)	+1 Call(currencyPair=C1C2, strike=B) +(B-K) BinaryCall(currencyPair=C1C2, strike=B)
+1 UaO_Call(currencyPair=C1C2, strike=K, barrier=B)	+1 CallSpread(currencyPair=C1C2, strike1=K, strike2=B) -(B-K) BinaryCall(currencyPair=C1C2, strike=B)

+1 DaI_Put(currencyPair=C1C2, strike=K, barrier=B)	+1 Put(currencyPair=C1C2, strike=B) +(K-B) BinaryPut(currencyPair=C1C2, strike=B)
+1 DaO_Put(currencyPair=C1C2, strike=K, barrier=B)	+1 PutSpread(currencyPair=C1C2, strike1=K, strike2=B) -(K-B) BinaryPut(currencyPair=C1C2, strike=B)
+1 BinarySpread(currencyPair=C1C2, strike1=K1, strike2=K2)	+1 BinaryCall(currencyPair=C1C2, strike=K1) -1 BinaryCall(currencyPair=C1C2, strike=K2)

Structured Products:

Contract	**Replication**
+1 DiscountCert(currencyPair=C1C2, cap=K, expiry)	-1 Put(currencyPair=C1C2, strike=K, expiry)
+1 ReverseConvertible(currencyPair=C1C2, strike=K, expiry)	+1 Put(currencyPair=C1C2, strike=K, expiry)
+1 BullSpreadCert(currencyPair=C1C2, strike1=K1, strike2=K2, expiry)	+1 CallSpread(currencyPair=C1C2, strike1=K1, strike2=K2, expiry)
+1 BearSpreadCert(currencyPair=C1C2, strike1=K1, strike2=K2, expiry)	+1 PutSpread(currencyPair=C1C2, strike1=K1, strike2=K2, expiry)
+1 OutperformanceCert(currencyPair=C1C2, strike=K, factor=F, expiry)	+F Call(currencyPair=C1C2, strike=K, expiry) 1 Put(currencyPair=C1C2, strike=K, expiry)
+1 CappedOutperformanceCert(currencyPair=C1C2, strike=K1, factor=F, capLevel=K2, expiry)	+F Call(currencyPair=C1C2, strike=K1, expiry) -1 Put(currencyPair=C1C2, strike=K1, expiry) -F Call(currencyPair=C1C2, strike=K2, expiry)
+1 BonusCert(currencyPair=C1C2, bonusLevel=K, barrier=B, expiry)	+1 Spot(currencyPair=C1C2) +1 DaO_Put(currencyPair=C1C2, strike=K, barrier=B)
+1 CappedBonusCert(currencyPair=C1C2, bonusLevel=K1, barrier=B, capLevel=K2, expiry)	+1 Spot(currencyPair=C1C2) +1 DaO_Put(currencyPair=C1C2, strike=K1, barrier=B) -1 Call(currencyPair=C1C2, strike=K2, expiry)
+1 TwinWin(currencyPair=C1C2, strike=K, barrier=B, expiry)	+1 Spot(currencyPair=C1C2) +2 DaO_Put(currencyPair=C1C2, strike=K, barrier=B)

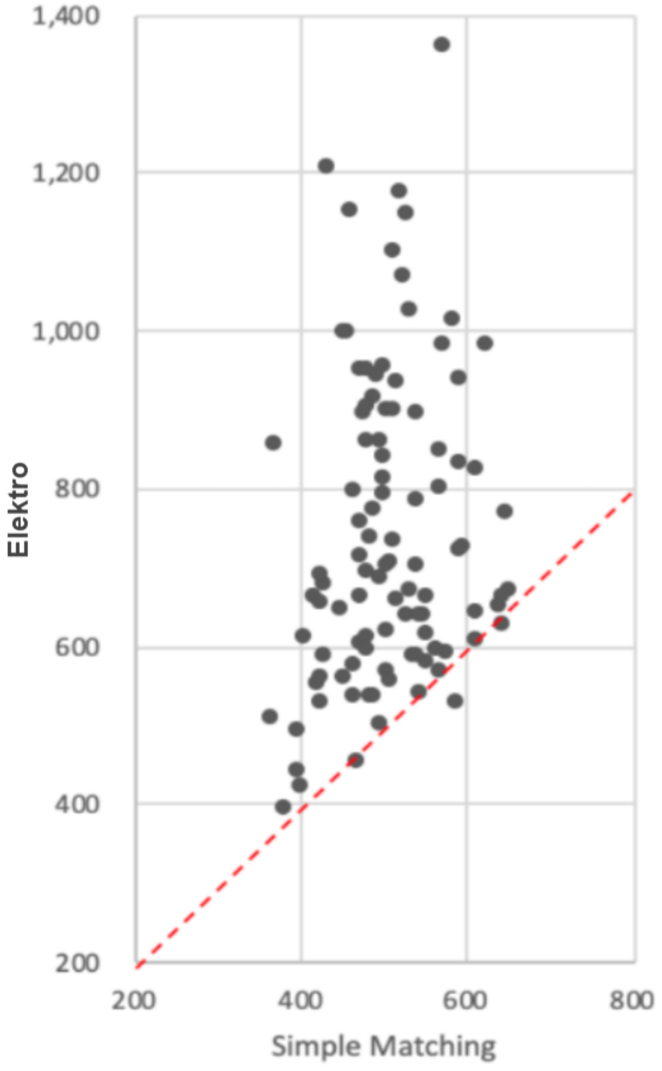
+1 CappedTwinWin(currencyPair=C1C2, strike=K1, barrier=B, capLevel=K2, expiry)	+1 Spot(currencyPair=C1C2) +2 DaO_Put(currencyPair=C1C2, strike=K1, barrier=B) -1 Call(currencyPair=C1C2, strike=K2, expiry)
+1 Airbag(currencyPair=C1C2, strike=K1, airbagLevel=K2,, expiry)	+1 Call(currencyPair=C1C2, strike=K1, expiry) -K1/K2 Put(currencyPair=C1C2, strike=K2, expiry)

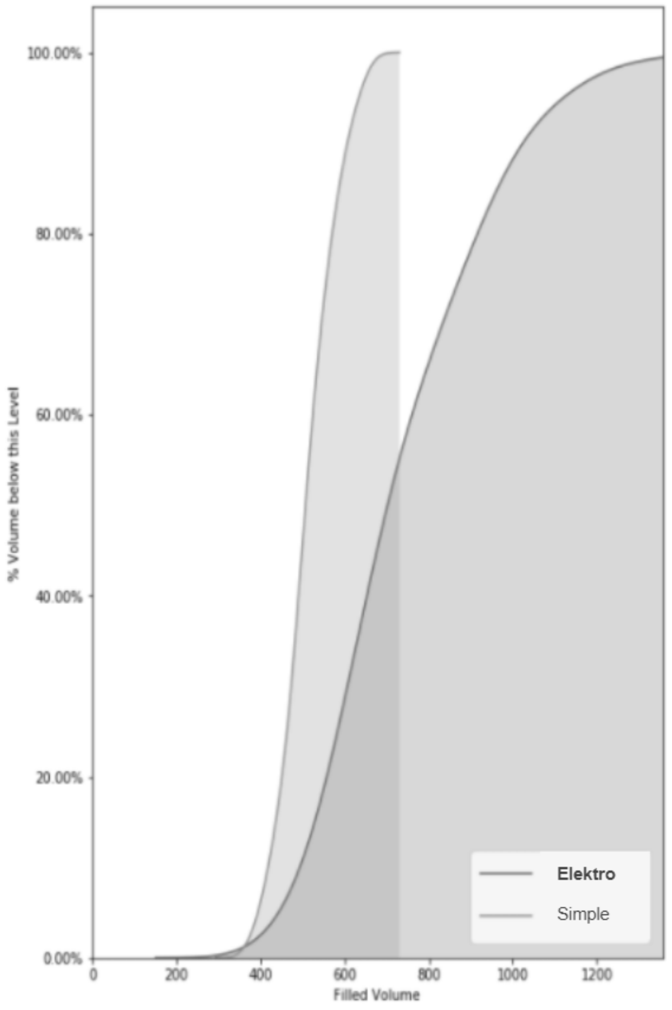
Engine Simulation

We simulated and compared randomly generated order books run through Elektro and a standard auction-based engine.

Test results for Configuration 1: crossed markets in puts, calls and forwards show:

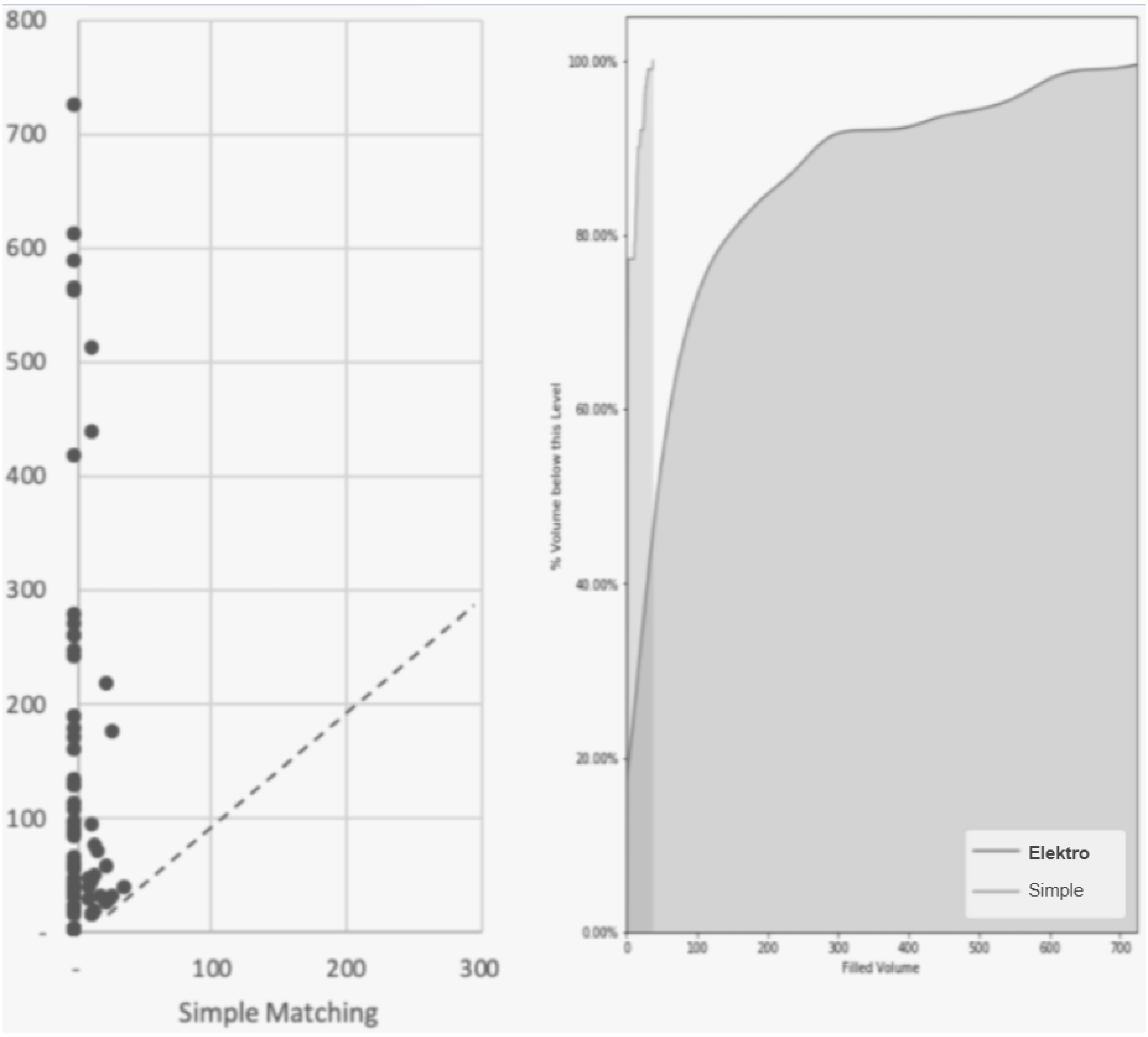
- Elektro ME significantly outperforms a standard auction-based ME
- Increasing the number of contracts increases Elektro's outperformance
- Elektro ME outperformed standard auction-based ME by 44%, under crossed puts calls, and forwards market configuration
- Elektro ME outperformed standard auction-based ME by 52%, under crossed puts calls, and forwards market configuration
- Elektro ME creates significant liquidity that otherwise would not exist under:
    - An uncrossed puts, calls, and forwards market configuration
    - An uncrossed puts, calls, forwards, and binary and barrier options market configuration
    - A standard auction-based ME, by definition, cannot find matches under these circumstances





Configuration 2: Uncrossed markets in puts, calls and forwards

- The outperformance of Elektro ME increases in illiquid markets. In uncrossed markets, simple matching finds few or no matches and the value of pooling liquidity across instruments increases.
- In our simulation, Elektro ME outperformed the simple matching engine by **2,149%.**

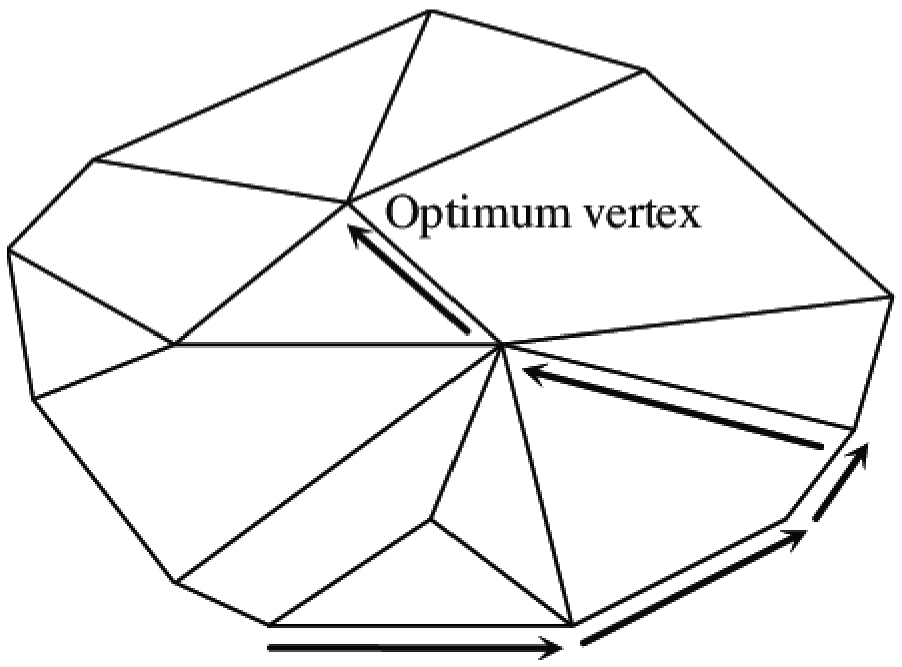


Mixed Integer Linear Programming (MIP) Optimization

Elektro allows for organic continuing inclusion of new products in ways that naturally do NOT cannibalize existing liquidity; to the contrary, more products create synergistically MORE liquidity among existing contracts. The ability to incorporate statically replicable structured products and collateral swap ones open up huge markets that are currently unattainable. This is done via Mixed Integer Linear Programming (MIP) Optimization. MIP optimization allows searching for clearing prices and associated sets of consistent orders satisfying them that maximize executed volume and satisfy best execution requirements.

Maximizing volume is challenging, as it involves optimizing a continuous quantity (volume), subject to discrete constraints. MIP utilizes advanced heuristics to perform an efficient search of the solution space, solving the problem using augmented Linear Programming tools (Simplex algorithm):

- **Branch-and-bound methods enumerate possible values for integer variables and prune away probably un-optimal ones**
- **Cut generators augment the problem with additional constraints, which guide the standard, continuous-valued Linear Program solutions toward integer values**



At current, Elektro is capable of processing auctions of 3,000 orders within 5 seconds, with scope for additional optimization to further improve auction processing time.

Deposit & Withdrawals

FundLock is a smart contract on the Ethereum network that is administered and managed by Elektro.

Participants hold cash assets within the Elektro framework via the FundLock smart contract and use the cash balance to facilitate their trading activity funding the premium and collateral requirements of their orders.

If and when an order gets executed and settled as a contract position, the premium gets settled to and from the participants FundLock accounts and the Collateral related to the new positions gets taken out of the FundLock account into the Elektro Collateral pool. Users may query their FundLock account balances directly on Ethereum.

As soon as the cash deposit has settled on the Ethereum network, the balance in the FundLock address will reflect the deposit and the funds are available to be used to fund new orders on Elektro.

Participants can also always request a withdrawal of their funds from their FundLock account, however, if the withdrawal would reduce the balance below the amount that is currently marked and locked for live open orders, the withdrawal will be delayed until such orders are automatically canceled on the trading Platform freeing up the amount of cash that the participant attempts to withdraw. When a user borrows, funds are moved from FundLock to *MarginLock*. Users may deposit to MarginLock directly on-chain.

A user may withdraw from MarginLock to FundLock up to the Maintenance Margin minimum threshold. A user may not withdraw from MarginLock directly to its wallet, but must instead transfer from MarginLock to FundLock and withdraw from FundLock to its wallet.

TRADING & SETTLEMENT

Collateral Requirements & Management

'Trustless' full collateralization

The Elektro Protocol is based on counterparty-risk free on-chain settlement. Participants must provide full collateral funding for any orders they submit at the time when an order is submitted into an orderbook. Please see the Elektro Contract Specifications for exact details of collateral amounts for the different contract types.

Typically, option buyers have to fully fund the limit price of an order for the potential premium that is to be paid in the case of a matched trade. Option sellers have to provide funds as collateral that match the maximum liability of the option that they are trying to sell.

The currency in which the collateral has to be provided depends on the exact contract for which an order is being submitted. Similar to settled positions after a trade has been matched (please see section On-Chain Settlement for more details), collateral is being held for participants within an escrow like on-chain smart contract called FundLock.

Participants can access their funds and deposit funds into or withdraw funds from the FundLock smart contract at any time. Any deposit or withdrawal is an on-chain transaction.

The orderbook will reject any new order or any updates to existing live orders if the funds required to cover the trade premium or the collateral are not available within the FundLock contract at the time of order submission or order update. If a participant withdraws funds from the FundLock smart contract that render them below the level of required funds for all the outstanding live orders of the participant, the platform will automatically cancel live orders until the available funds match the collateral and premium requirements of the remaining live orders.

When the Elektro Clearing Engine (ECE) executes a trade, Option Contract buyers must pay the premium in the Price Currency; Option Contract sellers receive that premium. Option Contract sellers also have to provide the required Trade Collateral Amount in Collateral Currency for the Option Contract they sold. The Trade Collateral Amount is being locked up in a Smart Contract for the duration of the trade until the traded Contract is exercised (see below).

The calculation of the Trade Collateral Amount follows the below set steps:

Step 1 - Calculation of the Collateral Amount

Collateral	Product Type
Vanilla Call Option	1 Underlying Asset
Vanilla Put Option	Strike
Call Spread Option	Upper Strike minus Lower Strike
Put Spread Option	Upper Strike minus Lower Strike
Binary Call Option	1 Underlying Asset
Binary Put Option	1 Underlying Asset
Up-And-Out Call Option	1 Underlying Asset
Up-And-In Call Option	1 Underlying Asset
Down-And-In Put Option	Strike
Down-And-Out Put Option	Strike
Forward Contract Long	Traded Strike/Fwd Price
Forward Contract Short	1 Underlying Asset

Step 2 – Calculation of the Scaled Collateral Amount The Scaled Collateral Amount is calculated by multiplying the Collateral Amount with the Contract Size and the Multiplier of the Contract.

Step 3 – Calculation of the Trade Collateral Amount The Trade Collateral Amount is calculated by multiplying the Scaled Collateral Amount with the quantity of the trade that was executed for the Option Contract seller

Collateral Optimization
Elektro requires, as described already, orders and trades to be fully collateralized in order to guarantee pay-outs to counterparties. For example, for a user to sell a call option on 1 BTC against USDC, the user needs to post 1 BTC collateral. Collateral Optimization enables the above-mentioned collateral requirements to be determined on a portfolio basis, which results in lower overall collateral requirements because of the presence of offsetting or partially offsetting trades. Under Collateral Optimisation, instead of collateral being determined on a trade-by-trade basis, collateral required is based on the maximum loss in the user's portfolio of Elektro products as a whole.

Portfolio collateral optimization process:

1. When a user places an order, the standard collateral per the Collateral requirements is locked up
2. After execution of the order in step 1), the Elektro Collateral Optimisation engine computes the required collateral for the user's portfolio of Elektro contracts and returns any excess collateral posted to the user

Collateral optimization logic and examples

Assuming BTC/USDC as the underlying (WBTC):

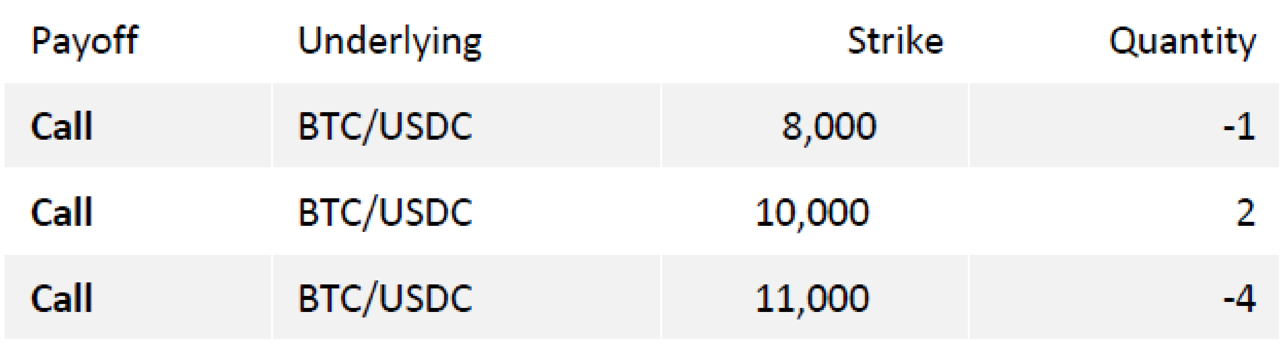
1) Calculate the quantity of BTC needed as collateral when the BTC/USDC price at expiry goes to infinity

2) Calculate USDC collateral required when the BTC/USDC price at expiry: a. Is equal to each level of strikes in the portfolio b. Is equal to 2)(i) +/- a small increment (to check for discontinuities in payoff) c. Zero

3) Compute 1) minus each of the amounts calculated in 2). If any of these amounts is negative, the amount of USDC collateral required is the minimum of the negative amounts.

4) The total collateral required is 1) + 3)

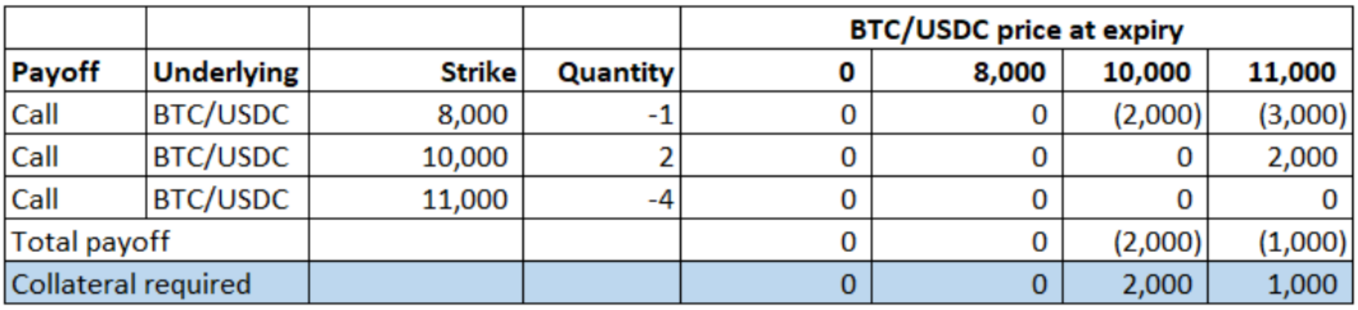
Example 1 Portfolio:



Payoff	Underlying	Strike	Quantity
Call	BTC/USDC	8,000	-1
Call	BTC/USDC	10,000	2
Call	BTC/USDC	11,000	-4

1) If BTC/USDC approaches infinity, the amount of BTC collateral required = 3 BTC

2) USDC collateral at specific BTC/USDC prices:



				BTC/USDC price at expiry			
Payoff	**Underlying**	**Strike**	**Quantity**	**0**	**8,000**	**10,000**	**11,000**
Call	BTC/USDC	8,000	-1	0	0	(2,000)	(3,000)
Call	BTC/USDC	10,000	2	0	0	0	2,000
Call	BTC/USDC	11,000	-4	0	0	0	0
Total payoff				0	0	(2,000)	(1,000)
Collateral required				0	0	2,000	1,000

3) Compute 1) – 2):

	BTC/USDC price at expiry			
	0	**8,000**	**10,000**	**11,000**
1) USDC equiv of 3 BTC	0	24,000	30,000	33,000
2) USDC collateral required	0	0	2,000	1,000
3) Net amount	0	24,000	28,000	32,000

USDC collateral required = 0

4) Total collateral required = 3 BTC

Example 2

Portfolio:

Payoff	Underlying	Strike	Quantity
Put	BTC/USDC	6,000	-1

1) If BTC/USDC approaches infinity, the amount of BTC collateral required = 0 BTC

2) USDC collateral at specific BTC/USDC prices:

				BTC/USDC price at expiry	
Payoff	**Underlying**	**Strike**	**Quantity**	**0**	**6,000**
Put	BTC/USDC	6,000	-1	(6,000)	0
Collateral required				6,000	0

3) Compute 1) – 2):

	BTC/USDC price at expiry	
	-	**6,000**
1) USDC equiv of 0 BTC	-	-
2) USDC collateral required	6,000	-
3) Net amount	(6,000)	-

4) Total collateral required = 6000 USDC

Trading Patterns and Collateral Implications

Trading Strategies and conditional orders

Elektro enables participants to implement diverse and complex trading strategies involving many different products using pre-packaged and structured products and most notably using conditional orders.

Conditional orders eliminate the execution risk between different orders while implementing trading strategies and position patterns. Another notable benefit of using conditional orders comes from the fact that there might be collateral optimization benefit (initially at order entry only) compared to what would be required for the corresponding orders sent simultaneously but as independent orders. When sending conditional orders, the Trading Platform will run a collateral optimization on the conditional order as a package and only the optimized collateral will be needed at order entry.

For the corresponding orders sent independently, Elektro would require the full collateral on each order independently at order submission. Any inter-dependencies between independent orders that lead to a smaller overall maximum liability compared with the linear sum of the maximum liabilities of all the legs will only take effect after the trades are executed and settled but are not being considered in the initial collateral requirements. This means that initially, at order submission collateral orders might require more collateral to be locked up than the overall portfolio will economically require. The excess collateral will be freed up again during settlement immediately after trades are executed.

Netting and Unwind trades

When a participant submits a new order, the collateral requirements for such order at the time of order entry are being calculated independently of any existing settled positions that a participant might already have in the same or any other contract.

That means that any new order that will lead to the economic unwind of an existing position will still initially require the same collateral as if it was an independent order leading to a completely new position. Similarly, orders that, once executed, will lead to an overall reduction of the economically required collateral of a participant’s portfolio, will initially require full independent collateralisation at order entry.

As soon as unwind and collateral reducing trades are executed, the Elektro Collateral Optimisation process will immediately release any excess collateral back to the participant’s FundLock cash balances. As a result of these system mechanics however, the closeout or unwind of existing positions typically do require the temporary provision of capital to fund the on-order-entry collateral requirements.

Leveraged Margin Trading

Elektro enables participants to borrow portion of the required collateral related to derivative orders and positions using Margin Loan Contracts. As described in the section on Margin Loan Trading below, such contracts can be traded as legs of a conditional order attached to derivative contract orders. Such a conditional Margin Loan order will reduce the collateral requirement of the overall conditional order according to the amount to be borrowed via the margin loan.

Similarly to unwind or close-out orders of derivatives or other types of contracts however, if and when a participant would like to trade out of an existing Margin Loan position prior to the expiry, such order to effectively ‘lend’ via a Margin Loan, will require the full notional of the margin loan to be temporarily posted as collateral on order entry.

As a consequence, the early ‘pay back’ or ‘redemption’ of a Margin Loan in the secondary market, does require full collateralization of the full loan notional at order entry for the period between the order submission and the settlement of the trade.

Maturity Settlement

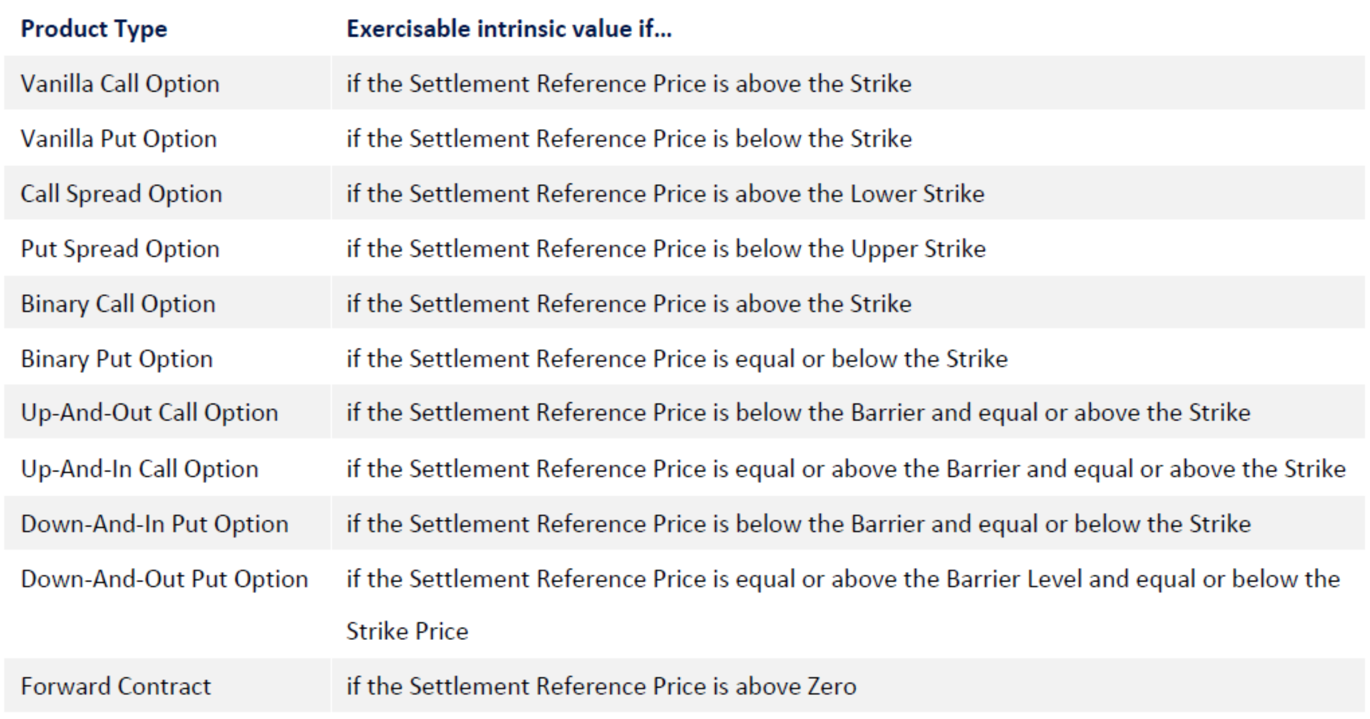
Settlement Rules

By default, all Regular and Custom Contracts are Auto-Exercised at Maturity if and only if the contract has exercisable intrinsic value. All contracts ‘cash settled’ in the sense that no party, long or short holder has to provide any additional funds during settlement. All settlement pay-outs to both long and short holders of contracts are being made from locked-up collateral amounts.
It is important to note that due to Elektro's collateral optimization mechanism, the currency of the locked-up collateral amount can be either the underlying currency or the premium currency of the underlying market and the pay-outs to long and short holders during the settlement process could happen in either currency without the information being available prior to the expiry. Participants however will be able to specify their preference settlement currency and will be able to enter into spot transactions to exchange the settlement currency into their preferred currency should they be different.

Contracts cannot be early exercised.

Contracts cannot be exercised if they do not have exercisable intrinsic value at Maturity.

Contracts with exercisable intrinsic value at Maturity must be auto-exercised.



Product Type	Exercisable intrinsic value if...
Vanilla Call Option	if the Settlement Reference Price is above the Strike
Vanilla Put Option	if the Settlement Reference Price is below the Strike
Call Spread Option	if the Settlement Reference Price is above the Lower Strike
Put Spread Option	if the Settlement Reference Price is below the Upper Strike
Binary Call Option	if the Settlement Reference Price is above the Strike
Binary Put Option	if the Settlement Reference Price is equal or below the Strike
Up-And-Out Call Option	if the Settlement Reference Price is below the Barrier and equal or above the Strike
Up-And-In Call Option	if the Settlement Reference Price is equal or above the Barrier and equal or above the Strike
Down-And-In Put Option	if the Settlement Reference Price is below the Barrier and equal or below the Strike
Down-And-Out Put Option	if the Settlement Reference Price is equal or above the Barrier Level and equal or below the Strike Price
Forward Contract	if the Settlement Reference Price is above Zero

All Contracts are being Auto-Exercised. If a contract has no exercisable intrinsic value, the Final Settlement Amount is 0.

Calculation of the Settlement Amount

If a contract has exercisable intrinsic value, the calculation of the Final Settlement Amount follows the below set steps:

Step 1 - Calculation of the Intrinsic Value

Product Type	Intrinsic Value
Vanilla Call Option	Settlement Reference Price minus Strike Price
Vanilla Put Option	Strike Price minus Settlement Reference Price
Call Spread Option	Min (Upper Strike, Settlement Reference Price) minus Lower Strike
Put Spread Option	Upper Strike minus Max (Lower Strike, Settlement Reference Price)
Binary Call Option	1
Binary Put Option	1
Up-And-Out Call Option	Settlement Reference Price minus Strike Price
Up-And-In Call Option	Settlement Reference Price minus Strike Price
Down-And-In Put Option	Strike Price minus Settlement Reference Price
Down-And-Out Put Option	Strike Price minus Settlement Reference Price
Forward Contract	Settlement Reference Price

Step 2 - Calculation of Scaled Intrinsic Value

The Scaled Intrinsic Value is calculated by multiplying the Intrinsic Value with the Contract Size and the Multiplier of the Contract.

Step 3 - Calculation of the Contract Settlement Amount

Contracts that have a different Settlement Contract than the Strike Contract require a conversion of the Scaled Intrinsic Value from the Strike Contract into the Settlement Contract using the Settlement Reference Price.

For Contract that have the same Settlement and strike Contract, the Contract Settlement Amount is equal to the Scaled Intrinsic Value.

Step 4 – Calculation of the Final Settlement Amount

The Final Settlement Amount is calculated by multiplying the Contract Settlement Amount with the quantity of the position of the Option Contract holder

Trading Facility

End of Trading

Trading in a Elektro contract will typically cease 2 hours before the Maturity of the contract. After the End of Trading the Elektro Platform will reject any new orders for such contract.

Orders and Orderbooks

All orders, BUY or SELL, are required to be Limit Orders. The platform does not allow for Market Orders to be submitted into the Orderbook.

As the Ecosystem is based on decentralized on-chain settlement of cash funds, participants are required to fulfill certain collateral and premium funding requirements before submitting any new order, or change an existing order on the platform. Orders can be valid for different time frames. A Day Order is canceled if it did not get executed by the end of the trading day on which it was submitted into the orderbook.

Unless otherwise specified, every order is a Day Order. A Good-Til-Canceled Order (GTC) will remain in the Orderbook as a live order until it is fully executed or the order is canceled. GTC orders that are not fully executed will automatically be canceled when a Contract stops trading and the orderbook is closed prior to the Contract's expiry. An Immediate-Or-Cancel (IOC) order requires all or part of the order to be executed in the next Frequent Batch Auction.

Any unexecuted parts of the order are canceled. Partial executions are accepted.

Multi-Leg Conditional Orders

Elektro allows for the submission of conditional orders.

Such orders consist of two or more legs. Each leg of the conditional order enters the matching engine as a separate order. Legs of a conditional order can be BUY or SELL orders. The order legs can be orders for any contract tradable on the platform. A conditional order can be fully executed if all its legs can be fully executed; or it can be partially executed if all its legs can be executed in the same pro-rata fraction.

A conditional order has one net limit price across all legs (or depending on the interface used, one net limit premium). A leg with a buy order is counted positively while a leg with a sell order is to be counted negatively. Therefore, in similar fashion to plain single orders, the net limit price of a conditional order represents the price of a normalized quantity of 1 unit. The normalized quantity of 1 unit for a conditional order relates to the same quantity across all legs.

Example 1:

An order to buy a 1x2 call spread can be submitted as a conditional order with two legs:

1) buy 1 contract of Call(K1)

2) sell 2 contracts of Call(K2)

The net limit price of the conditional order then relates to the price for the execution of 1/3 of Call(K1) and 2/3 of Call(K2).

If during an auction, Call(K1) clears at c1 and Call(K2) clears at c2 then the conditional order would clear at

[1 x c1 - 2 x c2] / [1 + 2]

The net limit price of a conditional order can be positive or negative. A positive net limit price indicates that execution of the conditional order will lead to the price, or less, to be paid (user is a net buyer), a negative net limit price on a conditional order will lead to the price, or more, to be received (user is a net seller)

Example 2:

An order to buy a risk reversal may be submitted as a conditional order with two legs:

1) sell 1 contract of Put (K1) 2)

buy 1 contract of Call (K2)

An order sent at a net price of -45 USDC means that the user wants to be net receiving (put over). If the Put(K1) clears at 1,100 USDC and the Call(K2) clears at 1,000 USDC then this implies a clearing net price for the risk reversal at [1,000 - 1,100] / [1 + 1] = - 50 USD and the previous conditional order would thus be filled.

The net total premium of a conditional order can be calculated by multiplying the net limit price of the order with the sum of the order quantities of all legs.

In the above risk reversal example, the net total premium would be - 45 x 2 or - 90 USDC.

Margin Trading and Liquidation Process

Under a trustless model, option sellers post collateral at the time of order submission to cover the maximum payoff of the option they are selling.

For example, a Call Seller sells a call option on 1 BTC against USDC and posts 1 BTC as collateral. This guarantees the pay-out to the counterparty. In this example, under margin trading, 1 BTC of collateral will still need to be locked up, however, part of that (say 25%) will be posted by the option seller and the remainder (75%) will be financed by a lender (the “Margin Lender”). The Margin Lender will receive a premium for protecting the derivative counterparty on the other side of the trade against the risk of collateral not being enough to satisfy his obligations.

The Margin Loan borrower will get a margin call when the value of his derivative positions falls below a certain threshold. To avoid liquidation, the borrower can transfer more funds to his Margin

Lock account. The risk for the lender is related to potential slippage when liquidating a portfolio which is underwater. We refer to that risk as Liquidation Risk.

Liquidation Risk

Liquidation Risk is defined as the risk that once the liquidation process has been initiated, the Linked Derivatives in the related Sub-Portfolio cannot be liquidated at a value that is sufficient to repay all the Margin Loan Principal. This is implemented via a requirement for the user's Margin Balance (net asset value) to cover the Maintenance Margin. All else equal, higher Maintenance Margin equates to lower Liquidation Risk.

Liquidation Engine

For Elektro's margined products, the Liquidation engine (ELE) continuously monitors users' margin balances against their maintenance margin requirements

● If a user's margin balance falls below its maintenance margin, the ELE will initiate a liquidation

● As liquidations occur when the market has moved against the user, this may involve the ELE having to liquidate an in-the-money option, where there is typically less liquidity in a typical options market

● The Elektro Matching Engine will execute these orders by replicating, for example, in-the-money puts into forwards and out-of-the money calls, thereby creating additional liquidity

SMART CONTRACTS

Proxy Pattern And Upgradability

Description of the Proxy Pattern architecture

General

Elektro's smart contract architecture is designed using the Proxy Delegate pattern (aka "Router and Resolver pattern"). State and functionality are separated. All function calls are sent to a Router address instead of the actual underlying smart contract. This design enables upgradability and extensible functionality under a single address.

Each module of the smart contract architecture is composed of multiple contracts which follow the aforementioned design pattern. This design pattern consists of 3 core concepts:

- Router (+ Delegator)
- Resolver
- Implementation Contract(-s)
- Optional Storage Contract

A `Router` is a contract that delegatecalls into multiple implementation contracts. These delegatecalls are executed in the context of the router. All storage reads/writes happen in the storage of the router. Each router has a resolver contract which maps the function signature to the address of the implementation contract. This is done to facilitate upgradeability of the system and extensible functionality under one unchanged address.

Architecture & Contracts

Contracts

Router

https://github.com/NomismaTech/elektro-protocol-aux/blob/development/contracts/delegate/Router.sol

The `Router` delegates method invocation calls to the concrete logic implementation. The function's code is executed in the context of the router by using delegatecall. A delegatecall in Ethereum works as follows: The code of the callee is executed in the context of the caller, additionally the original `msg` parameters are preserved.

Note that all data is stored in the context/storage of the Router. And Router does NOT get upgraded/redeployed.

Existing Storage layout of the implementation contracts should NOT ever change and will corrupt the data upon the upgrade if changed!

New storage slots have to be added below the existing once (appended to the end)!

All implementation contracts called by the same router have the same storage layout, and, most commonly, one Storage contract outlining common storage for all implementation contracts ("sitting" at the Router address). All implementation contracts have to inherit this Storage contract to be able to work with it and have the same storage layout. The Router defined in Router.sol is not deployed directly but inherited by the individual Router contracts which are deployed.

Every call (besides internal functionality) goes to the fallback function which determines the address of the implementation contract based on the function identifier (call to Resolver) from the call, then delegatecalls this implementation contract found.

Resolver

https://github.com/NomismaTech/elektro-protocol-aux/blob/development/contracts/delegate/Resolver.sol

Maps function signatures to the address of the implementation contracts. These entries can be updated by accounts bearing the governor role. All routers are connected to their respective resolvers. Resolver stores the first 4 bytes of keccak256 hash of a function signature and `lookup()` returns the address mapped with these 4 bytes.

Functions are registered in two ways:

- `register()` (`bulkRegister()`) - for new functions and contracts
- `update()` (`bulkUpdate()`) - for changed functions and updated contracts

Only accounts bearing Governor role can call these functions during the deploy/upgrade of the system!

Resolver contracts do NOT get upgraded/redeployed for modules existing on-chain! They are only updated through calling the above functions.

Implementation Contract(-s)

Each Proxy Module can have one or multiple implementation contracts.

In case of one contract, there's no difference between this implementation contract and any other regular Solidity contract.

In case of multiple contracts Storage has to be the same with absolutely no changes/discrepancies between the implementations within one Proxy Group (module).

Storage Contract

In case of multiple implementations or a large contract state (many storage vars) a separate `Storage` contract can be created to outline all storage (state) variable of the module.

All implementation contracts of one Proxy Module HAVE to inherit the same Storage contract and do not add to it within their own code!

Getters/Setters & Interfaces

In the case of Storage contract present, interfaces have to be created for all public state variables in order to be available to read from `Router` bound calls. These interfaces will be bound to the Router's ABI and will allow the backend to see and call all available functionality of the module.

If any storage functionality overhead is needed, Setters or Getters contracts can be created in order to provide specific logic needed.

Each implementation contract HAS to have its own interface, and in case of multiple implementations, those interfaces should be inherited by the main "umbrella" interface which during deployment will serve as an ABI for the Router, providing access to the full external functionality of the module.

> All the implementation functionality of the module is bound to the `Router`'s address, so even if we have 15 implementation contracts, we will always only call `Router` of the module, which HAS to have all this functionality outlined in one Interface!

Flow

Functionality is strictly separated. If one part needs a functionality of another part it delegatecalls into the code of the other part. This pattern is wrapped into the delegate function of the `Delegator`. The below diagram depicts the Router Pattern:

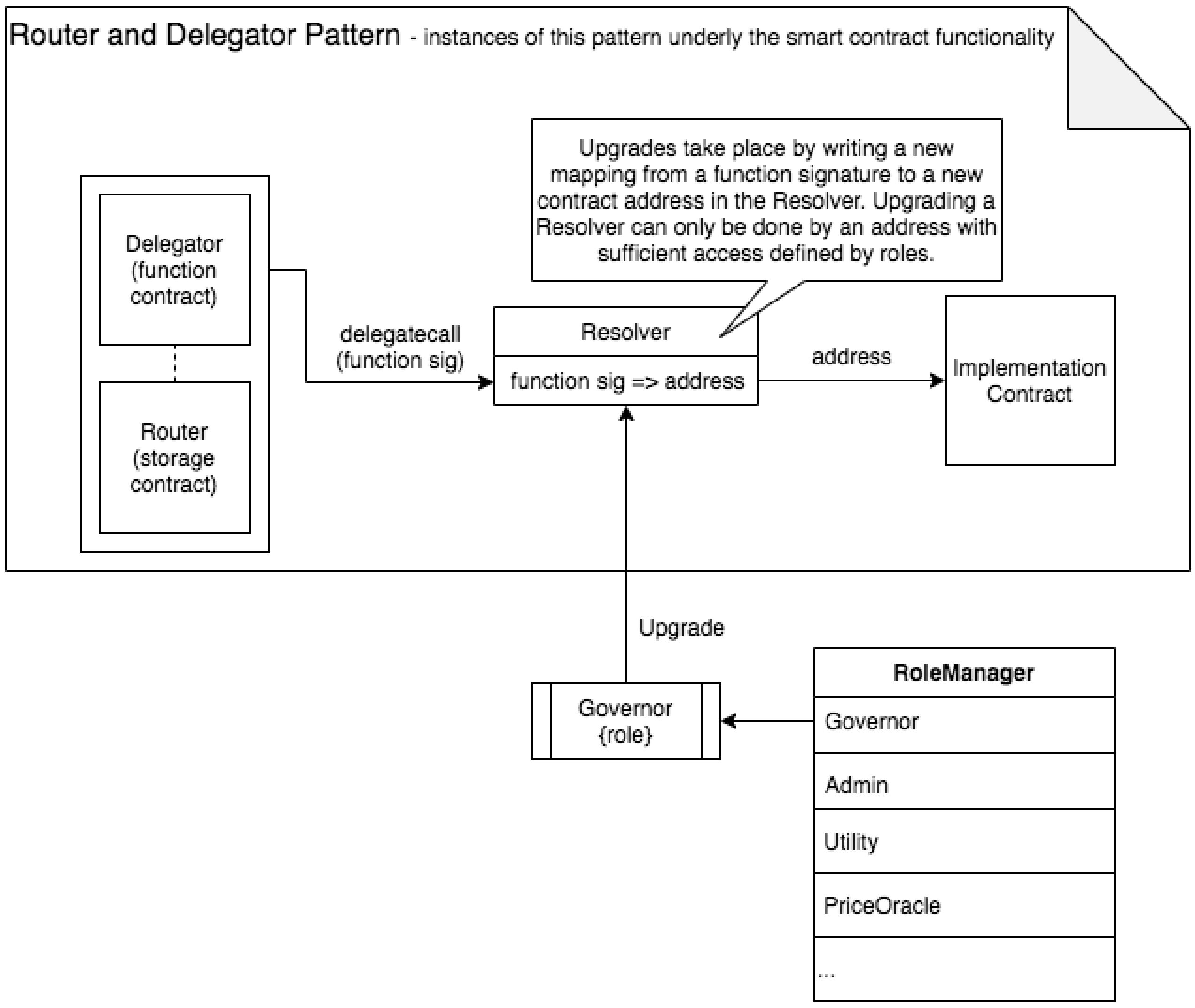

Router and Delegator Pattern - instances of this pattern underly the smart contract functionality
Upgrades take place by writing a new mapping from a function signature to a new contract address in the Resolver. Upgrading a Resolver can only be done by an address with sufficient access defined by roles.
Delegator (function contract)
Router (storage contract)
delegatecall (function sig)
Resolver
function sig => address
address
Implementation Contract
Upgrade
Governor {role}
RoleManager
Governor
Admin
Utility
PriceOracle
...

Diagram of the Proxy flow

Deployment & Initialization

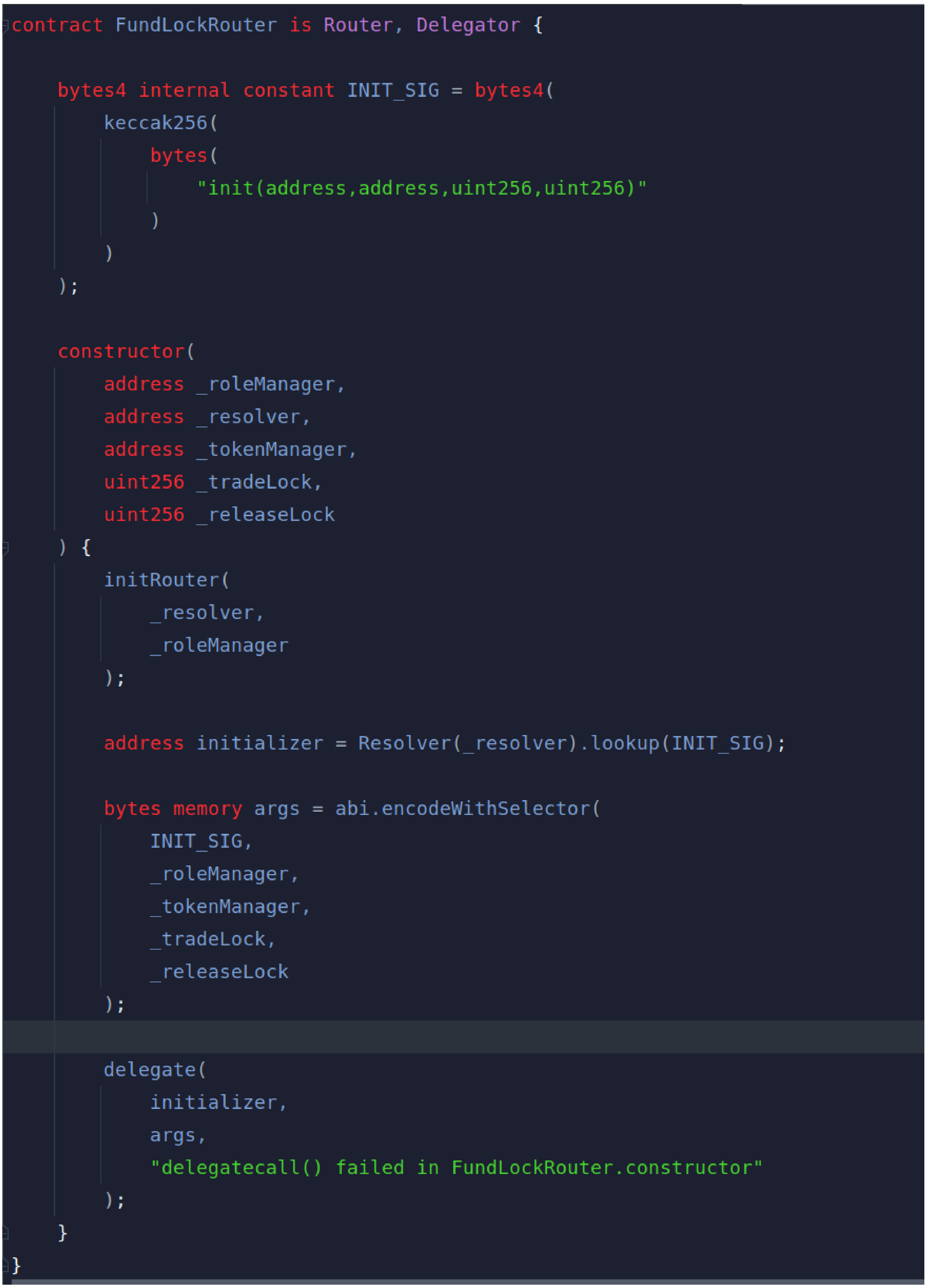
- All implementation contracts are deployed
- `Resolver` contract is deployed and initialized with all function signatures and the addresses of the implementation contracts
- `Router` is deployed and it's constructor initializes storage of the Proxy Module by calling `init()` function on the Implementation, this storage then becomes the storage of the `Router`

The individual parts of the system have their own router/resolver contracts. Each proxy group (functionality module) has one `Router` and one `Resolver` contract along with one or multiple implementation contracts.

`initialized` storage variable (boolean) is used in all proxy groups' storages to signify that a proxy group (module) has been deployed and its initial storage (state) set by calling init functions on implementations. Each `init()` function of the implementation contract is checking if this variable is "false" (otherwise – revert) and sets it to "true" upon storage initialization. This is used to prevent additional calls to init() functions and resetting the storage after deployment (this can only be done by manual calls from Governor or Admin accounts to the respective setter functions for each slot).

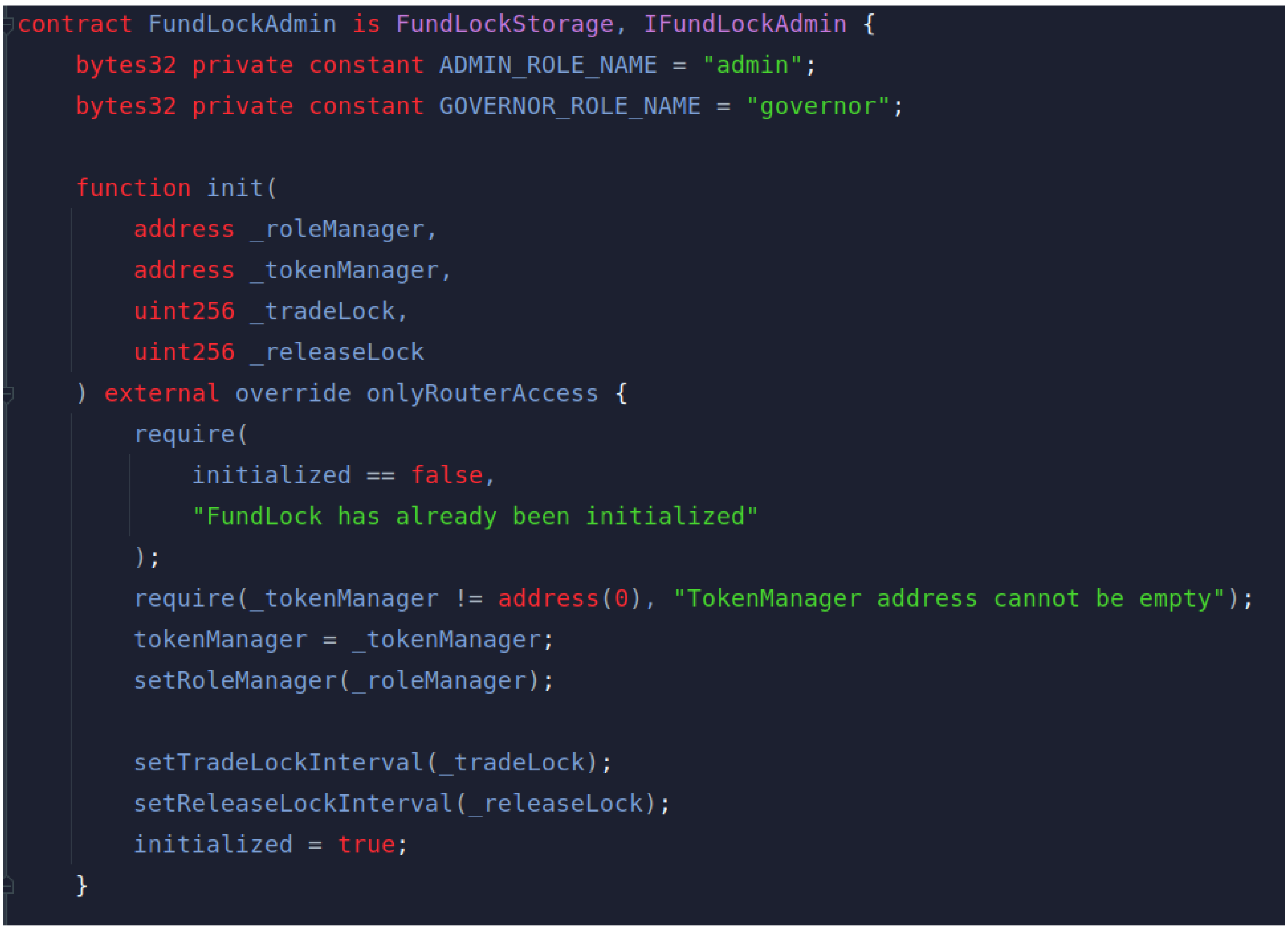
Example

An example of this pattern is the `FundLock`. `FundLock` manages the funds, but has no functionality to transfer tokens. If `FundLock` has to release tokens it does so by using the inherited `Delegator` functionality which executes a delegatecall into the `TokenWrapper` to execute the code invoking the safe transfer of the funds. The below code snippet illustrates this structure.



```
contract FundLockRouter is Router, Delegator {

    bytes4 internal constant INIT_SIG = bytes4(
        keccak256(
            bytes(
                "init(address,address,uint256,uint256)"
            )
        )
    );

    constructor(
        address _roleManager,
        address _resolver,
        address _tokenManager,
        uint256 _tradeLock,
        uint256 _releaseLock
    ) {
        initRouter(
            _resolver,
            _roleManager
        );

        address initializer = Resolver(_resolver).lookup(INIT_SIG);

        bytes memory args = abi.encodeWithSelector(
            INIT_SIG,
            _roleManager,
            _tokenManager,
            _tradeLock,
            _releaseLock
        );

        delegate(
            initializer,
            args,
            "delegatecall() failed in FundLockRouter.constructor"
        );
    }
}
```

FundLock initialization from the FundLockRouter (entry point)



```
contract FundLockAdmin is FundLockStorage, IFundLockAdmin {
    bytes32 private constant ADMIN_ROLE_NAME = "admin";
    bytes32 private constant GOVERNOR_ROLE_NAME = "governor";

    function init(
        address _roleManager,
        address _tokenManager,
        uint256 _tradeLock,
        uint256 _releaseLock
    ) external override onlyRouterAccess {
        require(
            initialized == false,
            "FundLock has already been initialized"
        );
        require(_tokenManager != address(0), "TokenManager address cannot be empty");
        tokenManager = _tokenManager;
        setRoleManager(_roleManager);

        setTradeLockInterval(_tradeLock);
        setReleaseLockInterval(_releaseLock);
        initialized = true;
    }
```

Actual implementation `init()` function to which the call is routed

`FundLockRouter` (as with any other specific Router) inherits all main Router functionality here and adds an initialization call for the storage. Initialization of the implementation contract `FundLockAdmin` and storage (`FundLockStorage`) happens at the construction of the `Router`, given it is going to maintain storage, where `Resolver.lookup()` returns an appropriate address for a hardcoded function signature of `FundLockAdmin.init()`. Arguments are assembled, encoded and passed through deletgatecall which triggers `FundLockAdmin` functionality to set all necessary storage variables.

Exceptions

The below contracts have to be called directly and are not upgradable. They are exceptions to the Proxy Delegate pattern use:

- RoleManager
- TokenWrapper
- TokenValidator

System Access

Governance, Roles & Privileged Access

https://github.com/NomismaTech/elektro-protocol-aux/blob/development/userRoles.md

The governance setup underpinning the smart contract functionality is designed to balance efficiency, security and trust. Where operations may be done by any network participant in a secure and efficient way, these functions are made public. Some functions are only operable by Elektro Team, but implemented such as not to invalidate the guarantees described in Section 1. Some operations, such as smart contract upgradability, will require the use of at least one additional party to act as a validator. Currently, all privileged operations can only be performed by Elektro.

Role Types

The governance setup is based on privileged roles. An address is granted a privileged role if it is added to the corresponding list by at least one other privileged address of the correct type. An account can hold none, one or multiple roles in the system. The roles and access control is managed by the RoleManager contract. Following roles exists:

Governor

The Governor role is the most privileged role in the system. An account bearing the Governor role can:

- add/revoke all other roles, this includes removing the Governor role of other Governors
- execute all governance functionality such as change entries in the Resolver or update the address of the contracts in the system, e.g. the Registry

Governors are assigned at construction time during deployment, new governors can be assigned by all required existing governors submitting requests for the specific address. `RoleManager` is initialized with `confirmationsRequired` state variable representing how many requests (confirmations) need to be done to assign a new governor.

`submitAddGovernorRequest()` function is called with an address of a candidate Governor by existing governors, when the `confirmationsRequired` is reached the new governor is assigned automatically with the last request submitted.

Admin

The Admin role is used to operate the system. An account bearing the admin role can:

- can set/remove the allowed tokens

- can set the time of the release lock and other parameters of the `FundLock`
- is the only role that can deploy an options market contract set

Utility

Main role for calling settlement functionality of the protocol, e.g. `updatePositions()`. This role belongs to Java Matching Engine which calls `ElektroLedger` smart contracts to settle all trades.

Role Specifics

Governor

The Governor is a top level controller role within the Elektro Smart Contracts ecosystem. It is used to interact with deployed Elektro smart contracts for configurations. Some of its functionalities include assigning roles in RoleManager, modifying contract addresses and registering and updating signatures in Router contracts. A Governor can assign any of the roles existing in Elektro smart contract.

The following tables describe the Smart Contracts and the functions within them a Governor is able to interact with.

Smart Contract	Function	Repository
RoleManager	submitAddGovernorRequest	elektro-protocol-aux
RoleManager	submitRemoveGovernorRequest	elektro-protocol-aux
RoleManager	revokeGovernorRequestConfirmation	elektro-protocol-aux
RoleManager	appointAdmins	elektro-protocol-aux
RoleManager	addRoleForAddress	elektro-protocol-aux
RoleManager	addRolesForAddresses	elektro-protocol-aux
RoleManager	removeRoleForAddress	elektro-protocol-aux
Resolver	bulkRegister	elektro-protocol-aux
Resolver	register	elektro-protocol-aux
Resolver	bulkUpdate	elektro-protocol-aux
Resolver	updateSignature	elektro-protocol-aux
Resolver	removeSignature	elektro-protocol-aux

Router	setResolver	elektro-protocol-aux
RegistryBase	setEventEmitter	elektro-protocol-aux
RegistryBase	setTokenManager	elektro-protocol-aux
RegistryBase	setInstanceResolver	elektro-protocol-aux
RegistryBase	setTokenValidator	elektro-protocol-aux
RegistryBase	setCommissionBeneficiary	elektro-protocol-aux
TokenManagerAdmin	setElektroRegistry	elektro-protocol-aux
TokenManagerAdmin	setTokenWrapper	elektro-protocol-aux
FundLock	setRegistry	elektro-protocol-aux
ElektroRegistrySetters	setFundLock	elektro-protocol-aux
ElektroRegistrySetters	postUpgradeInitialize	elektro-protocol-aux
ElektroSetters	setElektroEventEmitter	elektro-protocol

Admin

Admin is a controller level role within the Elektro Smart Contracts ecosystem. Admin role is assigned by a Governor. This role is used to modify contracts' business properties such as setting `releaseLock` and `tradeLock` and also calling initialization functions during deployment.

The following tables describe the Smart Contracts and the functions within them an Admin is able to interact with.

Smart Contract	**Function**	**Repository**
TokenManagerAdmin	setEthereumAddress	elektro-protocol-aux
TokenManagerAdmin	setWETH9Address	elektro-protocol-aux
TokenValidator	addTokensToWhitelist	elektro-protocol-aux
TokenValidator	removeTokenFromWhitelist	elektro-protocol-aux
FundLock	setReleaseLockInterval	elektro-protocol
FundLock	setTradeLockInterval	elektro-protocol

ElektroRegistry	deployElektro	elektro-protocol

Utility

Utility Account role is used for the functions execution which are called by Elektro Java Backend.

This account is assigned by Governors.

Smart Contract	Function	Repository
ElektroLedgerUpdate	updatePositions	elektro-protocol

Elektro Contract

The Elektro Contract’s role is assigned to the Elektro contract address in the Elektro Protocol.

This is not necessarily a role in the conventional sense. It is not validated by RoleManager, but validated by ElektroRegistry. We limit calls to the below functions based on Registry storage mapping which signifies if a contract has been registered as a part of the system. See `isValidContract` or `onlyAllowedContracts` modifiers.

Smart Contract	Function	Repository
TokenManagerAdmin	collectFundsToFundLock	elektro-protocol-aux
ElektroEventEmitter	emitLedgerPositionMoved	elektro-protocol
FundLock	updateBalances	elektro-protocol

Misc

All roles are trusted and expected to act correctly at all times, e.g. never making mistakes.

All other accounts & contracts: Are untrusted, should not interact with state changing functionality of the Elektro system.

Contracts interact with another. These interactions are trusted. The Registry is used and trusted to keep track of the contracts belonging to the system. This is another form of access control used within the system.

The `Resolver ` contracts of each router are assumed to be initialized correctly and trusted to return the correct address of the trusted implementation contract.

Auxiliary Contracts

Service and management contracts used by Elektro Protocol

https://github.com/NomismaTech/elektro-protocol-aux

RoleManager

`RoleManager` is the main contract of the system keeping track, assigning and validating roles and protected calls. Almost every contract in the system has `RoleManager's` address in its storage which is added as a storage slot upon inheriting a tiny `RoleAware` contract.

RoleManager's address in the storage of contracts using it ALWAYS has to be the first slot by inheriting `RoleAware` contract first in the chain!

RoleManager also holds logic for assigning new governors by accepting and managing requests. More on this in System Access

TokenManager Module

https://github.com/NomismaTech/elektro-protocol-aux/tree/development/contracts/tokens/manager

Contracts Deployed

- **TokenManagerRouter** – contract holding the address and storage of the `TokenManager` functionality module
- **TokenManagerResolver** – contract holding information about all `TokenManager` functionality mapped to specific implementation contract addresses
- **TokenManagerAdmin** – all current `TokenManager` functionality (including inherited`TokenManagerBase`). Storage of the `TokenManager` module is outlined in`TokenManagerBase`

The Token Manager manages the interaction with tokens. This includes helper functions that allow to get decimals, balances and allowances. Additionally a `collectFunds()` function is provided to safely transfer tokens to `msg.sender` using the `TokenWrapper`.

TokenWrapper

https://github.com/NomismaTech/elektro-protocol-aux/blob/development/contracts/tokens/safe-transfer/TokenWrapper.sol

The `TokenWrapper` is used to safely transfer ERC-20 tokens and is based on OpenZeppelin's SafeTransfer library. The TokenWrapper contract is deployed and used in a similar fashion as a library. Other contracts us the `Delegator` functionality to delegatecall into the code of the `TokenWrapper` contract.

TokenWrapper safe transfer functions use `IERC20.sol` OpenZeppelin library to perform safe transfers.

TokenValidator

https://github.com/NomismaTech/elektro-protocol-aux/blob/development/contracts/tokens/validation/TokenValidator.sol

The `TokenValidator` contract keeps a whitelist for tokens allowed to be used. Asset pair (underlying token/price token) are automatically whitelisted for each Elektro market created. Those token addresses are being passed at construction time to the `ElektroRegistry.deployElektro()` which in turn calls `TokenValidator` internally to whitelist them.

Tokens' whitelist is represented as a storage mapping which gets checked upon executing FundLock deposit logic and upon every update of the market.

Delisting a token would render a market obsolete, because no settlement operation for this token would be performed.

TokenValidator storage:

- **whitelistedTokens** - mapping of a token address to a Precision struct.
- **Precision struct:**
  - **precision** - value representing how many decimals are allowed for a particular token in a particular Elektro market
  - **toTokenPower** - value used in Elektro calculations representing the power of ten by which precision denominated token values need to be multiplied (30 * 10^`toTokenPower`). Is used to get proper multipliers to bring uint values to actual token denomination.

FundLock Module

Elektro Proxy Module for managing funds of all participants and their balances in the system

https://github.com/NomismaTech/elektro-protocol/tree/development/contracts/fund-lock

Contracts

- **FundLockAdmin** - Admin functionality, allows setting parameters (Registry Address, ReleaseLock, TradeLock) and initializes overall FundLock storage
- **FundLockTrade** - Functions for the Elektro to interact with the FundLock and update client balances after trade settlement
- **FundLockStateGetters** - Provides functionality to read information of the FundLocks state
- **FundLockUser** - Implements the functionality for the user to interact (deposit/withdraw/release)
- **IFundLockEvents** – interface declaring all the events used and fired by the module

- **FundLockStorage** – contract outlining the storage of the whole module (all FundLock contracts inherit this storage)

`FundLock` is designed as a joint custody solution between Elektro and the client. Clients may deposit and withdraw directly into/from `FundLock`. Only client can withdraw these funds from `FundLock`, no one from the Elektro team has access to the funds deposited/used by clients.

`FundLock` also acts as a 'gateway' for what can enter and exit the system. Clients may only deposit whitelisted tokens. The whitelisting process for token addresses is currently executable only by the Java backend. Upon expiry of a financial instrument, a client's payout is made available from `FundLock`.

FundLock serves as a main storage of clients' balances within the system. All tokens in circulation are outlined there. The FundLock holds the funds and keeps track of the balance sheet for all supported tokens. The balance sheet tracks the balances of the individual users.

Client Interactions

deposit()

Used to deposit tokens into FundLock. If Ether is used to deposit it needs to be sent with a transaction and then works under the hood as WETH. Important to know that WETH and ETH are essentially the same thing inside the Elektro Protocol trading logic. If a user doesn't have WETH, he can send regular ETH which will be converted by FundLock into WETH. The client will also get his funds back as ETH.

withdraw()

Used to send 'request to withdraw' to mark an amount of tokens client wishes to release later when it is allowed.

> No tokens are transfered at this point!
>
> A maximum of 5 withdraw requests can be stored of the same token from one client! Before adding any new ones a client must release some of the existing ones.
>
> Client's balance will change after this operation, but his marked funds can still be used to cover his trades if his remaining balance is not enough!

See "Withdrawal flow" below for more details.

release()

Used to release tokens (!) marked for withdrawal previously (!) from FundLock to the user, `withdrawTimestamp` parameter is used to determine which funds should be withdrawn as there can be multiple withdrawal requests for the same token and client (5 requests max).

Client Withdrawal Process

Withdrawing funds is a two step process split into:

1. **Withdraw** - in essence a ‘request to withdraw’ which marks funds and moves balance from one mapping to another while introducing some limitations on these funds usage (`releaseLock` and `tradeLock`).

2. **Release** - with a time lock (`releaseLock`) in between. This allows the backend to react and ensure all options are always covered. The time lock also ensures the client has eventual access to its funds if the backend were to cease being operational. The backend cannot prevent a withdrawal indefinitely.

Trade Lock Period

`tradeLock` outlines a timeframe during which funds marked for withdrawal can still be used to cover client’s trades before actual release. `tradeLock` is a state variable on `FundLockStorage` that is set at construction and can be changed later by a direct setter call.

When release() is called, there’s no option to pass an amount to the function, which means that client will get the amount that is left written to storage after everything that happened while lock timeframes were enforced.

Example

- A client has 100 WETH in his `balanceSheet.`
- He calls `withdraw()` to mark 50 WETH from his balance to be released later.
- His `balanceSheet` remainder will be 50 WETH and 50 WETH will be recorded to his `fundsToWithdraw` balance.
- `tradeLock` and `releaseLock` are enforced blocking him from calling release until the required time has passed.
- He makes a trade and settlement requires him to cover a collateral of 70 WETH.
- `FundLock` checks his `balanceSheet` first, figures out he does not have enough funds, so it checks his `fundsToWithdraw` balance where he finds 20 WETH available to add to his collateral, then it checks that `tradeLock` time is still ongoing which signifies, that 20 WETH can be subtracted from his `fundsToWithdraw`.
- `FundLock` covers his trade making his `balanceSheet` = 0 and his `fundsToWithdraw` balance = 30 WETH (remainder).
- Client calls `release()` with a timestamp of his previous withdraw request (taken from the `Withdraw` event timestamp).
- Client receives 30 WETH to his wallet.

At any point during this process if any of the requirements are not met a revert is performed negating the transaction.

Release Lock Period

`releaseLock` is a timeframe set by Java Backend to ensure that a client can not release his funds before all transactions and trades are settled. This prevents cheating, front-running and ensure that any backend or network delays will not affect trade settlement or cause errors in calculations.

This is a static, configurable variable. The backend calculates the estimated block number based on the time lock which is passed to the contract.

Client can not release his funds until releaseLock preiod has passed!

FundLock Storage

https://github.com/NomismaTech/elektro-protocol/blob/development/contracts/fund-lock/FundLockStorage.sol

All interaction with the FundLock must happen through the `FundLockRouter`. The actual logic is split across multiple separately deployed implementation contracts inheriting this contract.

State variables

- **Funds** – struct containing data for funds marked for withdrawal:
  - **value** – the amount of funds marked for withdrawal
  - **timestamp** – block timestamp for when the funds were marked (necessary for releasing them later)
- **ALLOWED_WITHDRAWAL_LIMIT** - maximum amount of 'requests to withdrawal' can be active (funds not yet released) at a time. *Set to a max of 5 requests!*
- **fundsToWithdrawS** - mapping to store user 'request to withdrawal'. User address => token address => (amount of tokens to withdraw, timestamp of withdrawal request)
- **tokenManager** - address of TokenManager contract needed for FundLock logic
- **registry** - address of ElektroRegistry contract needed for FundLock logic
- **balanceSheetS** - actual user balances of funds deposited in FundLock (user address => token address => amount). *This value does NOT include fundsToWithdraw.*
- **releaseLock** - time interval that has to be passed between calling *withdraw* and *release*
- **tradeLock** - time interval after which funds marked as to be withdrawn can be used for trading

Elektro Contracts

All business logic and registry contracts of Elektro Protocol

Contracts:

https://github.com/NomismaTech/elektro-protocol/tree/development/contracts/elektro

Functionality module responsible for settling trades. Can ONLY be called by the Java Backend bearing Utility Account role.

Consists of Initializer, Registry, EventEmitter, Ledger and Utilities contracts.

Elektro Initialize

https://github.com/NomismaTech/elektro-protocol/blob/development/contracts/elektro/init/ElektroInitialize.sol

Initializes storage used by `ElektroLedger` contracts that is tied to `ElektroRouter` by accepting a call from `ElektroRegistry` upon the deployment of a new Market (`ElektroRouter`). Calls other validation contracts like `TokenValidator` to whitelist an asset pair for a new market.

Elektro Registry Module

https://github.com/NomismaTech/elektro-protocol/tree/development/contracts/elektro/init

Main deployed contracts

https://github.com/NomismaTech/elektro-protocol-aux/tree/development/contracts/registry

Base contracts inherited by ElektroRegistry module

This is the registry of the elektro-protocol. Despite its name it does not only provide registry functionality for the multiple options markets but also features the core functionality to keep track of the base contracts of the system. Its storage also holds addresses of some of the auxiliary contracts in the system (`TokenManager`, `TokenValidator`, etc.).

Contracts

Proxy Pattern Contracts

- **ElektroRegistryRouter** - main address for all incoming calls and storage holder of the module
- **ElektroRegistryResolver** - resolver contract holding information about all external functionality to help `ElektroRegistryRouter` to route calls to implementation contracts

Implementation Contracts

- **RegistryBaseAdmin** - Provides the basic functions of the registry for the general contracts such as the `TokenManager`, initializes the `RegistryBaseStorage` of the module at construction of the `ElektroRegistryRouter` and is a base contract inherited by `ElektroRegistryAdmin`.
- **RegistryBaseSetters** - Allows accounts bearing the Governor role to update the addresses for contracts of the system and other state variables. Inherited by `ElektroRegistrySetters`.

- **ElektroRegistryAdmin** - Provides the functions of the registry for the Elektro specific contracts such as the FundLock or options market. Allows accounts bearing the Admin role to deploy, initialize and register new option markets into the system and initialize Registry Storage.
- **ElektroRegistrySetters** - Allows accounts bearing the Governor role to update the addresses of the Elektro contracts.
- **ElektroRegistryStorage** extends the `RegistryBaseStorage` and outlines the storage of the whole module which is tied to the `ElektroRegistryRouter`.
-

All `RegistryBase` contracts are a part of `elektro-protocol-aux` repository and are not being deployed directly, but inherited by `ElektroRegistry` implementation contracts.

Elektro Event Emitter Module

https://github.com/NomismaTech/elektro-protocol/tree/development/contracts/elektro/event-emitter

Main Elektro facing module

https://github.com/NomismaTech/elektro-protocol-aux/tree/development/contracts/event-emitter

Base contract inherited by ElektroEventEmitter implementation

Module responsible for all trade settlement related events.

ElektroEventEmitter does NOT include events of FundLock!

Contracts

- **ElektroEventEmitterRouter** – router of the module
- **ElektroEventEmitterResolver** – resolver of the module
- **ElektroEventEmitter** – implementation contract that fires all events related to Elektro trade settlements
- **IElektroEventEmitterEvents** – interface declaring all Elektro trading settlement related events.

The Elektro Markets

For each options market (asset pair) a separate `ElektroRouter` contract is deployed which uses the same implementation contracts as other `ElektroRouters` (markets). All `ElektroRouter` contracts share one `ElektroResolver` contract to resolve function signatures to the addresses of their respective implementation contracts. All on-chain options market functionality is to be called by the backend only.

Implementation Contracts

- **ElektroLedgerUpdate** – settlement logic for trading.

- **ElektroSetters** - This is a contract with only one setter function setElektroEventEmitter.
- **ElektroInitialize** - One function contract with Elektro market initialization logic. See "Elektro Initialize" section.

Additionally some Elektro utility contracts are deployed:

- **ElektroMultiplierAware** – utility contract to get whitelisted token precisions and token denomination multipliers.

Elektro Market Architecture And Settlement Examples

Elektro contract storage

Each `ElektroRouter` contract represents a market for a specific currency pair. When a new currency pair market is needed - new Elektro contract is deployed. Within each market trades for different contractId's which use this currency pair can be made. I.e. We can trade CALL and PUT options with different expiration dates using the same Elektro contract. The storage of each Elektro contract has respective addresses of underlying and strike currency which together represent a currency pair used by this market.

```
contract ElektroStorage {
   // ... omitted
   address public underlyingCurrency;
   address public strikeCurrency;
   // ... omitted
}
```

Within each Elektro contract client positions are represented by respective mapping. Positions are per contractId per trader. Each position denotes amount of option contracts each trader has in respective contractId (size). The size can be either negative (Sell) or positive (Buy).

```
contract ElektroStorage {
   // ... omitted
   mapping(uint32 => mapping(address => int64)) public clientPositions;
   // ... omitted
}
```

FundLock Storage

All data regarding actual funds/tokens of clients goes all the way down to FundLock and is updated in `FundLockStorage`. All clients' collateral and premium flows along with token flow from spot trades and other operations is updated in `balanceSheetS` storage mapping per user per token. This mapping is unified across all markets and trading contracts and outlines amount of funds available to client in every token used by the client for trades.

```
// client's address => tokenAddress => balanceAmount
mapping (address => mapping(address => uint256)) internal balanceSheetS;
```

Elektro contract update flow

Orders settlement flow `updatePositions` function accepts arrays of arguments. Logically these arrays can be split onto 2 virtual data structures: "Fund Movement" and "Position". Those will be represented by array groups passed as arguments.

```
contract ElektroLedgerUpdate {
 function updatePositions(
   // Position update part of arguments
   address[] memory positionClients,
   uint32[] memory positionContractIds,
   int64[] memory positionSizes,
   // Fund Movement part of arguments
   address[] memory fundMovementClients,
   int64[] memory underlyingAmounts,
   int64[] memory strikeAmounts,
   // backend tracking arg
   uint64 backendId
 ) external;
   // ... omitted
}
```

- The purpose of **Position** part is to record position information to smart contract state.
- The purpose of **FundMovement** part is to move funds in the `FundLock` without recording any information into `ElektroStorage` state. We represent things like premiums, collaterals, spot trades, margin loan underlying transfers as `Fund Movement`.

Please also note that arrays of `positionClients`, `positionContractIds`, `positionSizes` should always have the same lengths and this property is enforced by smart contract. Similarly arrays `fundMovementClients`, `underlyingAmounts`, `strikeAmounts`, would also have the same length which is also enforced within smart contract. On the contrary `positionClients` would not necessarily have the same length as `fundMovementClients` as they do represent different things. Also, it is possible for Backend to not send one part or the other (e.g. Spot Trading would only have **Fund Movements** part and arrays for positions will be empty), however smart contract will revert if both `positionClients` and `fundMovementClients` arrays are empty signifying an error on the backend or the fact that a TX is being sent with no data in it, for which we would still have to pay gas.

First three arguments represent a **Position** part with single structure sharing the same array index.

```
address[] memory positionClients,
uint32[] memory positionContractIds,
int64[] memory positionSizes,
```

The following three arguments represent **Fund Movement** part with single structure sharing the same array index.

```
address[] memory fundMovementClients,
int64[] memory underlyingAmounts,
int64[] memory strikeAmounts,
```

Lastly the `uint64 backendId` argument is meant for injecting a backend generated transaction id to be able to immediately assign it to transaction without waiting for txHash to become available.

This `backendId` is always used to emit an appropriate event in `ElektroEventEmitter` contract with the following signature:

```
event PositionsUpdated(
   uint64 indexed backendId
);
```

Naive Matching example data

A simple example of orders and expected input of `updatePositions` is as follows:

Order #	size	Price	Side	Strike	Option type	User	Match price	contractId
Order 1	20	11	BID	15	CALL	User 1	10	100500
Order 2	20	9	ASK	15	CALL	User 2	10	100500

Movement of funds need to include premium information in our case. With the following representation we have premium subtracted from account of User 1 and added to account of User 2. Importantly we introduce `Moves funds in FundLock` utility column in the following 2 tables. This column is never passed to smart contract functions as argument and is present rather to denote the fact that such row would either make `FundLock` contract to move funds or not do anything. In general all non-zero rows would move funds.

In the below example, **User 1 pays premium** to User 2 for his BID/BUY and **User 2 pays collateral** to the market for his ASK/SELL.

Please note the sign of the arguments. Here positive sign means client pays and negative means he receives.

Positions row index	positionClient	positionContractId	positionSize	Moves funds in Fundlock
0	User 1	100500	20	no
1	User 2	100500	-20	no

Fund movements row index	clientAddress	underlyingAmount	strikeAmount	Moves funds in FundLock
0	User 1	0	200	yes
1	User 2	20	-200	yes

Position change example data

Current smart contract state is important for validating incoming position's data. We provide a number of examples which describe situations when Users exit from their respective positions or are swapping long position for short one and vice-versa. After these examples we summarize the rules that apply to validate the input data and explain why those validations are appropriate. All following examples assume a 2 step flow. The second step is provided in each of the respective examples. Setup data for all the cases below is shared and similar to Naive Matching example data. After Naive Matching example data is sent to the contract the state is as follows:

Elektro State

```
{
 clientPositions: {
   100500: {
     'User 1': 20,
     'User 2': -20,
   }
 },
}
```

FundLock State

```
{
 balanceSheetS: {
   'User 1': {
     'underlyingAddress': IB`,
     'strikeAddress': IB` - 200
   },
   'User 2': {
     'underlyingAddress': IB` - 20,
     'strikeAddress': IB` + 200
   }
 }
}



` IB - Initial Balance
```

Example 1. Short position partial and full exit.

The second step is when User 2 submits an order for long position with same contractId:

Order #	size	Price	Side	Strike	Option type	User	Match price	contractId
Order 1	10	11	BID	15	CALL	User 2	10	100500
Order 2	10	9	ASK	15	CALL	User 3	10	100500

Such submission would result in the following input data to the `updatePositions` function:

Positions row index	clientAddress	positionContractId	positionSize	Moves funds in Fundlock
0	User 2	100500	10	no

1	User 3	100500	-10	no

Fund movements row index	fundMovementClient	underlyingAmount	strikeAmount	Moves funds in FundLock
0	User 2	-10	100	yes
1	User 3	10	-100	yes

After these changes are applied to state it would look as follows:

Elektro State

```
{
 clientPositions: {
   100500: {
     'User 1': 20,
     'User 2': -10,
     'User 3': -10
   }
 },
}
```

FundLock State

```
{
 balanceSheetS: {
   'User 1': {
     'underlyingAddress': IB`,
     'strikeAddress': IB` - 200
   },
   'User 2': {
     'underlyingAddress': IB` - 20 + 10,
     'strikeAddress': IB` + 200 - 100
   }
   'User 3': {
     'underlyingAddress': IB` - 10,
     'strikeAddress': IB` + 100
   }
 }
}


` IB - Initial Balance
```

As a result of this execution User 2 would get 10 underlying currency tokens back to his account and would have to pay 100 strike tokens in premium. User 3 would have to, in-turn, deposit 10 underlying currency tokens to back newly formed option.

Full exit example data is almost the same as the one provided in the above example. Distinction is only in the size of the order which would be 20. The input data for positions and the resulting state would be as follows:

Positions row index	positionClient	positionContractId	positionSize	Moves funds in Fundlock
0	User 3	100500	-20	no
1	User 2	100500	20	no

```
{
 clientPositions: {
   100500: {
     'User 1': 20,
     'User 2': 0,
     'User 3': -20
   }
 },
}
```

Example 2. Long position partial and full exit.

The second step is when User 1 submits an order for short position with same contractId:

Order #	size	Price	Side	Strike	Option type	User	Match price	contractId
Order 1	10	11	BID	15	CALL	User 3	10	100500
Order 2	10	9	ASK	15	CALL	User 1	10	100500

Such submission would result in the following input data to the `updatePositions` function:

Positions row index	clientAddress	contractId	positionSize	Moves funds in Fundlock
0	User 3	100500	10	no
1	User 1	100500	-10	no

Fund movements row index	clientAddress	underlyingAmount	strikeAmount	Moves funds in FundLock
0	User 3	0	100	yes
1	User 1	0	-100	yes

After these changes are applied to state it would look as follows:

Elektro State

```
{
 clientPositions: {
   100500: {
     'User 1': 10,
     'User 2': -20,
     'User 3': 10
   }
 },
}
```

FundLock State

```
{
 balanceSheetS: {
   'User 1': {
     'underlyingAddress': IB`,
     'strikeAddress': IB` - 200 + 100
   },
   'User 2': {
     'underlyingAddress': IB` - 20,
     'strikeAddress': IB` + 200
   },
   'User 3': {
     'underlyingAddress': IB`,
     'strikeAddress': IB` - 100
   }
 }
}


` IB - Initial Balance
```

As a result of this execution User 1 would have to post 0 underlying currency tokens as collateral even though he is engaging in a short position. He would also receive 100 underlying tokens in premium. There is no collateral modification in Elektro contract when this trade executes since all necessary collateral is already provided during the first trade.

To summarize the validation which would be applied: When user has a positive position in Elektro state and engages in negative position trade as a result the absolute value for his position's state would be decreased both of the collateral values for his position row should be zero. Full exit example data is almost the same as the one provided in this example. Distinction is only in the size of the order which would be 20. The input data for positions and the resulting state would be as follows:

Positions row index	positionClient	positionContractId	positionSize	Moves funds in Fundlock
0	User 3	100500	20	no
1	User 1	100500	-20	no

```
{
 clientPositions: {
   100500: {
     'User 1': 0,
     'User 2': -20,
     'User 3': 20
```

```
    }
  },
}
```

Example 3. Short position switch to long position.

The second step is when User 2 submits an order for long position with same contractId and the size of this position is larger then the size of the short position currently owned by User 2:

Order #	size	Price	Side	Strike	Option type	User	Match price	contractId
Order 1	30	11	BID	15	CALL	User 2	10	100500
Order 2	30	9	ASK	15	CALL	User 3	10	100500

Such submission would result in the following input data to the `updatePositions` function:

Positions row index	positionClient	positionContractId	positionSize	Moves funds in Fundlock
0	User 2	100500	30	no
1	User 3	100500	-30	no

Fund movements row index	fundMovementClient	underlyingAmount	strikeAmount	Moves funds in FundLock
0	User 2	-20	300	yes
1	User 3	30	-300	yes

After these changes are applied to Elektro state it would look as follows:

```
{
  clientPositions: {
    100500: {
      'User 1': 20,
      'User 2': 10,
      'User 3': -30
    }
  },
}
```

As a result of this execution User 2 would get 20 underlying currency tokens back to his account and would have to pay 300 strike tokens in premium. User 3 would have to in-turn deposit 30 underlying currency tokens to back newly formed option. User 1 and User 2 are now paired on the long side with User 3.

Example 4. Long position switch to short position.

The second step is when User 1 submits an order for short position with same contractId and the size of this position is larger then the size of the long position currently owned by User 1:

Order #	size	Price	Side	Strike	Option type	User	Match price	contractId
Order 1	30	11	BID	15	CALL	User 3	10	100500
Order 2	30	9	ASK	15	CALL	User 1	10	100500

Such submission would result in the following input data to the `updatePositions` function:

Positions row index	positionClient	positionContractId	positionSize	Moves funds in Fundlock
0	User 3	100500	30	no
1	User 1	100500	-30	no

Fund movements row index	clientAddress	underlyingAmount	strikeAmount	Moves funds in FundLock
0	User 3	0	300	yes
1	User 1	10	-300	yes

```
{
clientPositions: {
100500: {
'User 1': -10,
'User 2': -20,
'User 3': 30
}
},
}
```

NatSpec Technical Documentation

NatSpec documentation on each contract in the system

Proxy Pattern Base Logic

Inheritable contracts from other sources used to help with internal flow

Resolvable

View Source: @nomisma/elektro-protocol-aux/contracts/delegate/Resolvable.sol
↘ **Derived Contracts: ElektroStorage, RegistryBaseStorage, TokenManagerBase**
Resolvable

Contract is used in all concrete contracts that should have the same memory layout as Router All concrete contracts that are accessed through Router must inherit from this contract

Contract Members
Constants & Variables
```
contract Resolver public resolver;
```

Resolver

View Source: @nomisma/elektro-protocol-aux/contracts/delegate/Resolver.sol
↗ **Extends: RoleAware**

Resolver
Contract is used to store mapping between keccak signatures and address of deployed contract. It is used by router to resolve and call correct contract address. This contract is an essential part of Elektro's Proxy Pattern architecture! It does NOT get redeployed during a system on-chain upgrade, only internal storage gets updated to store data for newly deployed/upgraded implementations contracts throughout the system.

Contract Members
Constants & Variables
```
bytes32 private constant GOVERNOR_ROLE_NAME;
```

```
mapping(bytes4 => address) internal pointers;
```

IMPORTANT: This is a signature of the native Router function that can NOT be shadowed by any implementation contract. We check a keccak signature against it for every `register()` to make sure this doesn't happen. In the case of shadowing, Resolver's `lookup()` will return an incorrect address, since with any call in this case only {Router.setResolver()} will be called.
```
bytes32 internal constant SET_RESOLVER_SIG;
```

SignatureRegistered

Event fired for every newly registered function signature on Resolver.

Parameters

Name	Type	Description
keccakSignature	bytes32	
destination	address	

SignatureUpdated

Event fired for every updated (e.g. arguments changed after upgrade) function signature on Resolver.

Parameters

Name	Type	Description
keccakSignature	bytes32	
destination	address	

SignatureRemoved

Event fired for every updated (e.g. arguments changed after upgrade) function signature on Resolver.

Parameters

Name	Type	Description
keccakSignature	bytes32	

Functions

- (address roleManager)
- bulkRegister(bytes32[] keccakSignatures, address[] destinations)
- register(bytes32 keccakSignature, address destination)
- bulkUpdate(bytes32[] keccakSignatures, address[] destinations)
- updateSignature(bytes32 keccakSignature, address destination)
- removeSignature(bytes32 keccakSignature)
- lookup(bytes4 signature)
- stringToSig(string signature)
- _register(bytes32 keccakSignature, address destination)
- _updateSignature(bytes32 keccakSignature, address destination)
- assertSignatureAndDestination(bytes32 keccakSignature, address destination)

Constructor setting {RoleManager} contract to storage

```
function (address roleManager) public nonpayable
```

Arguments

Name	Type	Description
roleManager	address	address of {RoleManager} contract

bulkRegister

Function to register mapping of multiple signatures to corresponding smart contract addresses.

```
function bulkRegister(bytes32[] keccakSignatures, address[] destinations) public
nonpayable onlyRole
```

Arguments

Name	Type	Description
keccakSignatures	bytes32[]	array of signatures to be registered
destinations	address[]	array of contract addresses that signatures will point to

register

Function to register single `keccakSignature` to address mapping

```
function register(bytes32 keccakSignature, address destination) public nonpayable
onlyRole
```

Arguments

Name	Type	Description
keccakSignature	bytes32	signature to be registered
destination	address	contract address that signature will point to See {_register}

bulkUpdate

Function to update existing signatures in bulk during a SC upgrade

```
function bulkUpdate(bytes32[] keccakSignatures, address[] destinations) public
nonpayable onlyRole
```

Arguments

Name	Type	Description
keccakSignatures	bytes32[]	signatures to be updated
destinations	address[]	contract addresses that signatures will point to

updateSignature

Function to update existing `keccakSignature` to address mapping, used during SC upgrade

```
function updateSignature(bytes32 keccakSignature, address destination) public nonpayable onlyRole
```

Arguments

Name	Type	Description
keccakSignature	bytes32	signature to be updated
destination	address	contract address that signature will point to

removeSignature

Function to remove single `keccakSignature`.

```
function removeSignature(bytes32 keccakSignature) public nonpayable onlyRole
```

Arguments

Name	Type	Description
keccakSignature	bytes32	signature to be removed

lookup

View to check address of contract for given first 4 bytes of keccak `signature`.

```
function lookup(bytes4 signature) public view
returns(address)
```

Arguments

Name	Type	Description
signature	bytes4	

stringToSig

Converts string signature to first 4 bytes of keccak `signature`.

```
function stringToSig(string signature) public pure
returns(bytes4)
```

Arguments

Name	Type	Description
signature	string	

_register

Function to register single `keccakSignature` to address mapping and emit a `SignatureRegistered` event that can be found in the transaction events.

```
function _register(bytes32 keccakSignature, address destination) internal nonpayable
```

Arguments

Name	Type	Description
keccakSignature	bytes32	
destination	address	

_updateSignature

```
function _updateSignature(bytes32 keccakSignature, address destination) internal nonpayable
```

Arguments

Name	Type	Description
keccakSignature	bytes32	
destination	address	

assertSignatureAndDestination

```
function assertSignatureAndDestination(bytes32 keccakSignature, address destination) internal view
```

Arguments

Name	Type	Description
keccakSignature	bytes32	
destination	address	

Router

View Source: @nomisma/elektro-protocol-aux/contracts/delegate/Router.sol
↗ **Extends: RoleAware** ↘ **Derived Contracts: ElektroEventEmitterRouter, ElektroRegistryRouter, ElektroRouter, FundLockRouter, TokenManagerRouter**
Router

Contracts that extend {Router} contract serves as data storage. {Router} delegates all calls based on mapping found in Resolver. All calls are executed in context of calling router and for every

functionality/proxy modules, Router's address is being used as an entry point and represents functionality of the module (full module's ABI is tied to the respective Router's address). This contract is not being deployed, but inherited by specific Router contracts of each module.
IMPORTANT: A signature for any external or public function added to this contract needs to be present in Resolver contract as a constant and be checked against during `register()` process. {Router} functions can NOT be shadowed by any other contracts!

Contract Members

Constants & Variables

```
bytes32 private constant GOVERNOR_ROLE_NAME;
```

```
contract Resolver public resolver;
```

Functions

- ()
- ()
- setResolver(address _resolver)
- initRouter(address _resolver, address _roleManager)

Default fallback functions that intercepts/accepts all calls. Method signature found in Resolver is used to get address of contract to call. Next `delegatecall` is executed to call resolved contract address.

IMPORTANT: This function gets called ONLY when no functions with the needed name are present in this contract!

```
function () external payable
```

Arguments

```
function () external payable
```

Arguments

setResolver

Setting {Resolver} contract used for routing calls. This address is one of the main key points of each Router contract, since this address will be used to figure out where to route calls for each module.

IMPORTANT! This function should not exist on implementation interfaces or implementation logic contracts! This would cause a potential shadowing problem that might arise. We should ONLY call this function on a particular {Router} contract DIRECTLY, and not through an implementation interface it is used with!

```
function setResolver(address _resolver) public nonpayable onlyRole
```

Arguments

Name	Type	Description

_resolver	address	address of {Resolver} contract to be set for {Router}

initRouter

{Router} initialization function that sets resolver and governance role

```
function initRouter(address _resolver, address _roleManager) internal nonpayable
```

Arguments

Name	Type	Description
_resolver	address	address of {Resolver} contract to be set for {Router}
_roleManager	address	

Delegator

View Source: @nomisma/elektro-protocol-aux/contracts/delegate/Delegator.sol

↘ **Derived Contracts: ElektroEventEmitterRouter, ElektroRegistryAdmin, ElektroRegistryRouter, FundLockRouter, FundLockStorage, TokenManagerAdmin**

Delegator

Contract contains extracted common logic for call delegation. Contracts that are delegating calls are extending Delegator contract

Functions

- delegate(address calleeContract, bytes parameters, string errorMsg)

delegate

Delegates call to specified contract. If delegatecall fails, transaction is reverted Additional check is done to verify that contract to call exists

```
function delegate(address calleeContract, bytes parameters, string errorMsg)
internal nonpayable
returns(bytes)
```

Returns

result of delegatecall as bytes array. Those bytes can be used in contract that is delegating call

Arguments

Name	Type	Description
calleeContract	address	address of contract to delegate call to

parameters	bytes	function signature and parameters encoded to bytes
errorMsg	string	error message to revert in case of failed delegatecall execution

OnlyRouterAccess

View Source: @nomisma/elektro-protocol-aux/contracts/utils/OnlyRouterAccess.sol
↘ **Derived Contracts: ElektroEventEmitter, ElektroStorage, FundLockStorage, RegistryBaseStorage**

OnlyRouterAccess

Contract used to ensure that function call can be only executed by {Router} contract Some functions needs to have restricted access that can be only accessible by {Router} calls, In order to use onlyRouterAccess modifier, contract needs to inherit from OnlyRouterAccess contract See {Router} docs

Contract Members
Constants & Variables
```
bool internal isImplementationContract;
```

Modifiers
- onlyRouterAccess

onlyRouterAccess

Checks if calling contract is not implementation contract.
```
modifier onlyRouterAccess() internal
```

Arguments
Functions
- ()

Constructor initializes isImplementationContract var to true Contracts that extend OnlyRouterAccess will always have this var set to true without possiblity to change

```
function () public nonpayable
```

Arguments

Role Management

RoleAware

View Source: @nomisma/elektro-protocol-aux/contracts/access/RoleAware.sol
↘ **Derived Contracts: ElektroStorage, FundLockStorage, RegistryBaseStorage, Resolver, Router, TokenManagerBase, TokenValidator**

RoleAware
Provides the ability for heirs to restrict the use of functions by specific roles using the `onlyRole` modifier. See {IRoleManager}.

Contract Members
Constants & Variables
```
contract IRoleManager internal roleManager;
```

Modifiers
- onlyRole

onlyRole

IMPORTANT: This function can not be changed after Router contracts have been deployed, since their logic is not upgradable, and changing this logic which will be inherited in implementation contracts might result in vulnerability since all Routers on chain would have this current (unchanged) logic.

```
modifier onlyRole(bytes32 roleName) internal
```

Arguments

Name	Type	Description
roleName	bytes32	

Functions
- setRoleManager(address _roleManager)

setRoleManager
```
function setRoleManager(address _roleManager) internal nonpayable
```

Arguments

Name	Type	Description
_roleManager	address	

Roles

View Source: @nomisma/elektro-protocol-aux/contracts/backward-compatibility/Roles.sol

Roles

Library for managing addresses assigned to a Role. Contract defines struct `Role` that keeps mapping for which address has requested Role

Structs

Role
```
struct Role {
mapping(address => bool) bearer
}
```

Functions
- add(struct Roles.Role role, address account)
- remove(struct Roles.Role role, address account)
- has(struct Roles.Role role, address account)

add

Function to grant role to account. New address is added to `role` mapping
```
function add(struct Roles.Role role, address account) internal nonpayable
```

Arguments

Name	Type	Description
role	struct Roles.Role	Role mapping of current roles to addresses
account	address	to grant role to

remove

Function to grant role to account. Address is removed from `role` mapping
```
function remove(struct Roles.Role role, address account) internal nonpayable
```

Arguments

Name	Type	Description
role	struct Roles.Role	Role mapping of current roles to addresses
account	address	to remove from the mapping

has

Function to check if address is present in `roleMapping`
```
function has(struct Roles.Role role, address account) internal view
returns(bool)
```

Arguments

Name	Type	Description
role	struct Roles.Role	Role mapping of current roles to addresses
account	address	to check if its present in the mapping

RoleManager

Central contract in the system that keeps and manages account roles and access to the system. Most crucial logic like setting address, changing {Resolver}'s signatures, contract initizlizing logic, trade settlement are restricted to accounts that have valid roles. See docs/SystemAccess.md for more details about roles (RoleManager.sol)

View Source: @nomisma/elektro-protocol-aux/contracts/access/RoleManager.sol
↗ **Extends: IRoleManager** ↘ **Derived Contracts: RoleManagerProxy**

RoleManager

Enums

OperationType

```
enum OperationType {
AddGovernor,
RemoveGovernor
}
```

Structs

GovernorRequest

```
struct GovernorRequest {
uint256 confirmationCount,
address[] confirmations
}
```

Contract Members

Constants & Variables

```
bytes32 public constant GOVERNOR_ROLE_NAME;
```

```
bytes32 public constant ADMIN_ROLE_NAME;
```

```
bytes32 public constant UTILITY_ACCOUNT_ROLE_NAME;
```

All add-governor requests already submitted to the system. Maps governor candidate address to {GovernorRequest} struct holding data of all requests for a particular candidate.

```
mapping(address => struct RoleManager.GovernorRequest) public govRequests;
```

Variable set upon construction representing how many request are required to assign a new governor for the system.

```
uint256 private confirmationsRequired;
```

Maps role name to {Roles} library to access all base role related functionality.

```
mapping(bytes32 => struct Roles.Role) private roles;
```

RoleAdded
Event fired upon adding a role for an address.

Parameters

Name	Type	Description
roleName	bytes32	- bytes32 representation of a role name
user	address	- address of a user that is assigned a role above

RoleRemoved
Event fired upon removing a role for an address.

Parameters

Name	Type	Description
roleName	bytes32	- bytes32 representation of a role name
user	address	- address of a user that for which the above role is removed

RequestConfirmed

Event fired upon submission/confirmation of a governor addition/removal request.See {submitGovernorRequest}.

Parameters

Name	Type	Description
sender	address	- person who called {submitGovernorRequest} and sent the request
govAddress	address	- address of the governor candidate to appoint/remove the role for

operationType	enum RoleManager.OperationType	- an {OperationType} enum value representing if it's an addition or removal call

ConfirmationRevoked

Event fired upon revoking a request for addition/removal of a governor.See {revokeGovernorRequestConfirmation}.

Parameters

Name	Type	Description
sender	address	- person who called {revokeGovernorRequestConfirmation} and sent the request
govAddress	address	- address of a governor candidate for whom request is being revoked

GovernorAdded

Event fired upon successful addition of a new governor.

Parameters

Name	Type	Description
govAddress	address	- address of a new governor added

GovernorRemoved

Event fired upon successful removal of a governor.

Parameters

Name	Type	Description
govAddress	address	- address of a new governor removed

Modifiers

- onlyRole

onlyRole

Modifier preventing unauthorized addresses from calling protected functions.

```
modifier onlyRole(bytes32 roleName) internal
```

Arguments

Name	Type	Description
roleName	bytes32	

Functions

- (address[] governors, uint256 _confirmationsRequired)
- appointAdmins(address[] admins)
- addRoleForAddress(address addr, bytes32 roleName)
- addRolesForAddresses(address[] addresses, bytes32[] rolesArr)
- removeRoleForAddress(address addr, bytes32 roleName)
- submitAddGovernorRequest(address govAddress)
- submitRemoveGovernorRequest(address govAddress)
- revokeGovernorRequestConfirmation(address govAddress)
- checkRole(address addr, bytes32 roleName)
- hasRole(address addr, bytes32 roleName)
- addRole(address addr, bytes32 roleName)
- removeRole(address addr, bytes32 roleName)
- submitGovernorRequest(address govAddress, enum RoleManager.OperationType operationType)
- acceptGovernorRequest(address govAddress, enum RoleManager.OperationType operationType)
- addGovernorRole(address addr)
- removeGovernorRole(address addr)
- isGovernorRequestConfirmed(address govAddress)
- isNotConfirmedBySender(address govAddress)

Constructor that assigns accounts with `governor` role Governor role can be only added during contract construction and when {_confirmationsRequired} number of governors {submitAddGovernorRequest}. Note that `governors.length >= _confirmationsRequired`

```
function (address[] governors, uint256 _confirmationsRequired) public nonpayable
```

Arguments

Name	Type	Description

governors	address[]	array of addresses to be added as governors
_confirmationsRequired	uint256	number of confirmations required to add/remove governor

appointAdmins

Adds an `ADMIN_ROLE_NAME` for the provided array of `admins`.

```
function appointAdmins(address[] admins) external nonpayable onlyRole
```

Arguments

Name	Type	Description
admins	address[]	array of addresses to be added as admins

addRoleForAddress

Function to assign a role to an address

```
function addRoleForAddress(address addr, bytes32 roleName) external nonpayable
onlyRole
```

Arguments

Name	Type	Description
addr	address	address to add role to
roleName	bytes32	name of the role to add

addRolesForAddresses

Assigns multiple roles to addresses.

```
function addRolesForAddresses(address[] addresses, bytes32[] rolesArr) external
nonpayable onlyRole
```

Arguments

Name	Type	Description
addresses	address[]	array of user addresses for which roles should be added

rolesArr	bytes32[]	array of respective roles

removeRoleForAddress

Removes a `roleName` for the provided `addr`.

```
function removeRoleForAddress(address addr, bytes32 roleName) external nonpayable
onlyRole
```

Arguments

Name	Type	Description
addr	address	address to remove role from
roleName	bytes32	name of the role to remove See {removeRole}.

submitAddGovernorRequest

Multisig implementation for adding new governor. In order to add new governor at least `confirmationsRequired` amount of governors need to call this function for same `govAddress`. For the first governor that calls this function with new `govAddress`, new `GovernorRequest` is created and stored in RoleManager. Only after last governor executed this function, `governor` role is added for `govAddress`.

```
function submitAddGovernorRequest(address govAddress) external nonpayable onlyRole
```

Arguments

Name	Type	Description
govAddress	address	account address for which `governor` role is requested to be added

submitRemoveGovernorRequest

Multisig implementation for removing governor. In order to remove governor at least `confirmationsRequired` amount of governors need to call this function for same `govAddress`. For the first governor that calls this function with new `govAddress`, new `GovernorRequest` is created and stored in {RoleManager.govRequests}. Only after last governor executed this function, `governor` role is removed for `govAddress`.

```
function submitRemoveGovernorRequest(address govAddress) external nonpayable
onlyRole
```

Arguments

Name	Type	Description
govAddress	address	account address for which `governor` role is requested to be removed

revokeGovernorRequestConfirmation

Function to provide ability for current governors to revoke previously granted confirmation of adding or removing a new governor.

```
function revokeGovernorRequestConfirmation(address govAddress) external nonpayable
onlyRole
```

Arguments

Name	Type	Description
govAddress	address	account address for which confirmation was granted before and needs to be removed

checkRole

Checks the existence of a `roleName` value for the given `addr`. Function reverts if account does not have required role

```
function checkRole(address addr, bytes32 roleName) public view
```

Arguments

Name	Type	Description
addr	address	address to check role for
roleName	bytes32	name of the role to be checked

hasRole

See {Roles.has}.

```
function hasRole(address addr, bytes32 roleName) public view
returns(bool)
```

Arguments

Name	Type	Description
addr	address	

roleName	bytes32	

addRole

Adds a `roleName` for the provided `addr` and emits a `RoleAdded` event that can be found in the transaction events. See {Roles.add}.

```
function addRole(address addr, bytes32 roleName) internal nonpayable
```

Arguments

Name	Type	Description
addr	address	
roleName	bytes32	

removeRole

Removes a role for the provided `addr` and emits a `RoleRemoved` event that can be found in the transaction events. See {Roles.remove}.

```
function removeRole(address addr, bytes32 roleName) internal nonpayable
```

Arguments

Name	Type	Description
addr	address	
roleName	bytes32	

submitGovernorRequest

Internal function with base functionality for both adding and removing governors.Checks if this is a first request, in which case it would initialize a new slot in {govRequests} adding request data or, in case of subsequent requests, would add a new request to an existing mapping slot. Also fires an event and checks if the required number of requests reached, in which case it will add or remove new governor automatically.

```
function submitGovernorRequest(address govAddress, enum RoleManager.OperationType
operationType) internal nonpayable
```

Arguments

Name	Type	Description
govAddress	address	- address of a governor candidate to add or remove

operationType	enum RoleManager.OperationType	- enum value signifying type of the operation (add or remove)

acceptGovernorRequest

Function that adds/removes a governor.This function is launched automatically when the number of requests required is reached.

```
function acceptGovernorRequest(address govAddress, enum RoleManager.OperationType
operationType) internal nonpayable
```

Arguments

Name	Type	Description
govAddress	address	- address of a governor candidate to add or remove
operationType	enum RoleManager.OperationType	- enum value signifying type of the operation (add or remove)

addGovernorRole

Adds a `GOVERNOR_ROLE_NAME` for the provided `addr` and emit a `RoleAdded` event that can be found in the transaction events. Can only be run as a part of governor request/confirmation flow.

```
function addGovernorRole(address addr) internal nonpayable
```

Arguments

Name	Type	Description
addr	address	address to check governor role for See {Roles.add}.

removeGovernorRole

Removes a `GOVERNOR_ROLE_NAME` for the provided `addr` and emit a `RoleRemoved` event that can be found in the transaction events. Can only be run as a part of governor request/confirmation flow.

```
function removeGovernorRole(address addr) internal nonpayable
```

Arguments

Name	Type	Description
addr	address	address to check governor role for See {Roles.remove}.

isGovernorRequestConfirmed

Function to check if governor request for `govAddress` has enough confirmations to be executed

```
function isGovernorRequestConfirmed(address govAddress) internal view
returns(bool)
```

Returns
true if at least confirmationsRequired number of requests are confirmed for `govAddress`

Arguments

Name	Type	Description
govAddress	address	address identifier of governor requests

isNotConfirmedBySender

Function to check if governor request has been already confirmed by this `msg.sender`

```
function isNotConfirmedBySender(address govAddress) internal view
returns(bool)
```

Returns
true if `govAddress` is not already confirmed by `msg.sender`

Arguments

Name	Type	Description
govAddress	address	address identifier of governor requests

Token Management

TokenProperties

View Source: @nomisma/elektro-protocol-aux/contracts/utils/TokenProperties.sol

TokenProperties
Library responsible for getting decimals and symbol values from tokens
Functions

- getTokenDecimals(address token)
- getSymbol(address token)

getTokenDecimals

Helper functions that get decimals property for specified tokenAssembly is used to get decimals from a token. Some tokens do not have decimals() function. If decimals() function is not found default value 18 is returned. Otherwise decimals value is taken from the token

```
function getTokenDecimals(address token) internal view
returns(decimals uint256)
```

Returns
decimals - decimals value for requested token

Arguments

Name	Type	Description
token	address	address of asset to get decimals

getSymbol

```
function getSymbol(address token) internal view
returns(string)
```

Arguments

Name	Type	Description
token	address	

TokenWrapper

View Source: @nomisma/elektro-protocol-aux/contracts/tokens/safe-transfer/TokenWrapper.sol
↘ **Derived Contracts: TokenWrapperProxy**
TokenWrapper

Contract used for safe transferring non compatible {ERC20} tokens. According to article, some {ERC20} tokens do not return value for transfer, {transferFrom}, approve functions. {TokenWrapper} implements safe methods that can handle non-compliant tokens. Using assembly we check if method execution was successful, even if no value was returned.

Functions

- safeTransfer(address _token, address _to, uint256 _value)
- safeTransferFrom(address _token, address _from, address _to, uint256 _value)
- safeApprove(address _token, address _spender, uint256 _value)

safeTransfer

```
function safeTransfer(address _token, address _to, uint256 _value) external payable
returns(bool)
```

Arguments

Name	Type	Description
_token	address	
_to	address	
_value	uint256	

safeTransferFrom

```
function safeTransferFrom(address _token, address _from, address _to, uint256
_value) external payable
returns(bool)
```

Arguments

Name	Type	Description
_token	address	
_from	address	
_to	address	
_value	uint256	

safeApprove

```
function safeApprove(address _token, address _spender, uint256 _value) external
payable
returns(bool)
```

Arguments

Name	Type	Description
_token	address	
_spender	address	
_value	uint256	

TokenValidator

View Source: @nomisma/elektro-protocol-aux/contracts/tokens/validation/TokenValidator.sol
↗ **Extends: RoleAware** ↘ **Derived Contracts: TokenValidatorProxy**
TokenValidator

Contract used for whitelisting tokens used in Elektro Markets Only user with "admin" role can add or remove tokens to whitelist If token is not valid, transaction is reverted Additionally contract is storing precision values for tokens. Precision is used in trade settlement calculations

Structs

Precision

```
struct Precision {
uint256 precision,
uint256 toTokenPower
}
```

Contract Members

Constants & Variables

```
bytes32 private constant ADMIN_ROLE_NAME;
```

Mapping holding data for Elektro whitelisted token precisions.Maps token address to a {Precision} struct holding precision values. Notice that a zero precision can NOT be passed since it serves as a sign that a precision for a certain token has not been initialized. See {Precision}

```
mapping(address => struct TokenValidator.Precision) public whitelistedTokens;
```

TokenWhitelisted

Event fired upon whitelisting a token for Elektro usage.

Parameters

Name	Type	Description
whitelistedToken	address	- address of a whitelisted token
precision	uint256	- amount of decimals used/important for Elektro during rounding of values (always <= token decimals!)
toTokenPower	uint256	- difference between decimals and precision, this value is used for calculations to convert (normalize) precision based amount to actual token denomination 10^(decimals-precision)

TokenRemovedFromWhitelist

Event fired upon token delisting.

Parameters

Name	Type	Description
deletedToken	address	

Functions

- (address roleManager)
- validateTokens(address[] tokens)
- validateToken(address token)
- getTokenPrecision(address token)
- addTokensToWhitelist(address[] tokens, uint256[] precisions)
- removeTokenFromWhitelist(address token)

Constructor setting {RoleManager} contract

```
function (address roleManager) public nonpayable
```

Arguments

Name	Type	Description
roleManager	address	address of {RoleManager} contract

validateTokens

Function to bulk check if tokens can be used in Elektro as deposit in {FundLock} and traded with

```
function validateTokens(address[] tokens) public view
```

Arguments

Name	Type	Description
tokens	address[]	addresses of tokens to be checked

validateToken

Function to check if token can be used in Elektro as deposit in {FundLock} and traded with

```
function validateToken(address token) public view
```

Arguments

Name	Type	Description
token	address	address of token to be checked

getTokenPrecision

Function to get precision for token.

```
function getTokenPrecision(address token) public view
returns(precision uint256, toTokenPower uint256)
```

Returns
precision of a requested token

Arguments

Name	Type	Description
token	address	address of token to get precision for

addTokensToWhitelist

Function to add addresses of tokens to be whitelisted for Elektro

```
function addTokensToWhitelist(address[] tokens, uint256[] precisions) public
nonpayable onlyRole
```

Arguments

Name	Type	Description
tokens	address[]	array of tokens addresses to be added to {TokenValidator}'s whitelist
precisions	uint256[]	array of precision values to store in {TokenValidator} for tokens

removeTokenFromWhitelist

Function to remove token address from {TokenValidator} whitelist

```
function removeTokenFromWhitelist(address token) public nonpayable onlyRole
```

Arguments

Name	Type	Description
token	address	address of token to be removed from {TokenValidator} whitelist

TokenManagerBase

View Source: @nomisma/elektro-protocol-aux/contracts/tokens/manager/TokenManagerBase.sol ↗ **Extends: RoleAware, Resolvable ↘ Derived Contracts: ITokenManagerMain, TokenManagerAdmin**

TokenManagerBase

Base contract inherited by {TokenManagerAdmin} and {TokenManagerAssets}. Keeps addresses of contracts needed for {TokenManager} functionality

Contract Members
Constants & Variables

Address of the {ElektroRegistryRouter}

```
address public elektroRegistry;
```

Address of Ether (ETH). This is used internally to differentiate between ERC20 tokens and ETH. Any fund related operation check token address against this one to verify if the token in the operation is ETH or not. If it is - we need to do a different flow for managing it.

```
address public ethereumAddress;
```

Address of the {TokenWrapper}

```
address internal tokenWrapper;
```

Address of Wrapped Ethereum (WETH). This is used internally to differentiate between other ERC20 tokens and WETH. Any fund related operation check token address against this one to verify if the token in the operation is WETH or not. If it is - we need to do a different flow for managing it.

```
address internal weth9Address;
```

BankRegistrySet

Parameters

Name	Type	Description
bankRegistryAddress	address	

ElektroRegistrySet

Parameters

Name	Type	Description
elektroRegistryAddress	address	

EthereumAddressSet

Parameters

Name	Type	Description
etherAddress	address	

WETH9AddressSet

Parameters

Name	Type	Description
weth9Address	address	

TokenWrapperSet

Parameters

Name	Type	Description
tokenWrapperAddress	address	

Modifiers

- onlyAllowedContracts

onlyAllowedContracts

Only contracts that are verified by Registry are allowed

```
modifier onlyAllowedContracts(address _registry) internal
```

Arguments

Name	Type	Description
_registry	address	

TokenManagerAdmin

View Source: @nomisma/elektro-protocol-aux/contracts/tokens/manager/TokenManagerAdmin.sol ↗ **Extends: Delegator, TokenManagerBase, ITokenManagerAdmin** ↘ **Derived Contracts: TokenManagerAdminProxy**

TokenManagerAdmin

Contract is responsible for managing interactions with tokens. It provides helper functions that allows to get decimals, balances and allowances of tokens. {collectFunds} function is used to safely transfer tokens to `msg.sender` by using {TokenWrapper} contract which implements {safeTransferFrom} function to transfer non compatible {ERC20} tokens

Contract Members

Constants & Variables

```
bytes32 private constant GOVERNOR_ROLE_NAME;
```

```
bytes32 public constant ADMIN_ROLE_NAME;
```

Functions

- setElektroRegistry(address _registry)
- setEthereumAddress(address _ethereumAddress)
- setWETH9Address(address _weth9Address)
- setTokenWrapper(address _wrapper)
- collectFundsToFundLock(address from, address tokenAddress, uint256 amount)
- getEthereumAddress()
- getWETH9Address()
- getTokenWrapper()
- getTokenDecimals(address token)
- getTokenSymbol(address token)
- getTokenProperties(address[] tokens, address spender)
- collectFundsCommon(address from, address tokenAddress, uint256 amount)

setElektroRegistry

Setting Elektro registry in {TokenManager}

```
function setElektroRegistry(address _registry) external nonpayable onlyRole
```

Arguments

Name	Type	Description
_registry	address	address of {ElektroRegistry} contract

setEthereumAddress

Setting ethereum asset address in {TokenManager}

```
function setEthereumAddress(address _ethereumAddress) external nonpayable onlyRole
```

Arguments

Name	Type	Description
_ethereumAddress	address	address of ethereum asset

setWETH9Address

Setting {WETH9} asset address in {TokenManager}

```
function setWETH9Address(address _weth9Address) external nonpayable onlyRole
```

Arguments

Name	Type	Description
_weth9Address	address	address of {WETH9} contract

setTokenWrapper

Setting {TokenWrapper} address in {TokenManager}

```
function setTokenWrapper(address _wrapper) external nonpayable onlyRole
```

Arguments

Name	Type	Description
_wrapper	address	address of {TokenWrapper} contract

collectFundsToFundLock

Function that allows to safely transfer tokens to {FundLock} from an external source. This function is always called by {FundLock}, so msg.sender is address of {FundLock} contract Can be used only by verified Elektro contracts.

```
function collectFundsToFundLock(address from, address tokenAddress, uint256 amount)
external nonpayable onlyAllowedContracts
```

Arguments

Name	Type	Description
from	address	address of account to transfer tokens from
tokenAddress	address	address of the token to transfer

amount	uint256	of tokens to be transferred

getEthereumAddress

Getting address of Ether set in {TokenManager}

```
function getEthereumAddress() external view
returns(address)
```

Arguments

getWETH9Address

Getting address of WETH9 set in {TokenManager}

```
function getWETH9Address() external view
returns(address)
```

Arguments

getTokenWrapper

Getting address of {TokenWrapper} contract set in {TokenManager}

```
function getTokenWrapper() external view
returns(address)
```

Arguments

getTokenDecimals

Helper functions that gets decimals property for specified token

```
function getTokenDecimals(address token) external view
returns(uint256)
```

Arguments

Name	Type	Description
token	address	address of asset to get decimals

getTokenSymbol

Helper functions that gets symbol property for specified token

```
function getTokenSymbol(address token) external view
returns(string)
```

Arguments

Name	Type	Description
token	address	address of asset to get symbol for

getTokenProperties

Helper functions that gets multiple balances and allowances for specified tokens

```
function getTokenProperties(address[] tokens, address spender) external view
returns(uint256[], uint256[])
```

Arguments

Name	Type	Description
tokens	address[]	address of assets to get balances and allowances
spender	address	address of account used to get allowance

collectFundsCommon

See {collectFunds}.

```
function collectFundsCommon(address from, address tokenAddress, uint256 amount)
internal nonpayable
```

Arguments

Name	Type	Description
from	address	
tokenAddress	address	
amount	uint256	

TokenManagerRouter

View Source: @nomisma/elektro-protocol-aux/contracts/tokens/manager/TokenManagerRouter.sol
↗ **Extends: Router** ↘ **Derived Contracts: TokenManagerRouterProxy**

TokenManagerRouter

Router that redirects calls to concrete {TokenManager} contract based on resolver signature to address mapping

Functions

- (address _roleManager, address _resolver)

Constructor to create new Router contract. {RoleManager} and {Resolver} are needed to set role access and resolve the address of the called signature.

```
function (address _roleManager, address _resolver) public nonpayable
```

Arguments

Name	Type	Description
_roleManager	address	{RoleManager} contract address
_resolver	address	{Resolver} contract address

Registry

RegistryBaseStorage

Used to store most critical contract addresses

View Source: @nomisma/elektro-protocol-aux/contracts/registry/RegistryBaseStorage.sol
↗ **Extends: RoleAware, Resolvable, OnlyRouterAccess** ↘ **Derived Contracts: ElektroRegistryStorage, RegistryBaseAdmin, RegistryBaseSetters**

RegistryBaseStorage

Contract is used as a single place to keep addresses of major contracts used in Elektro. Instead of passing all needed contract addresses to contracts that need them, only Registry contract is passed. This approach ensures that change of contract address is done in one place, instead of multiple updates in all the contracts.

Structs

ContractStorage

```
struct ContractStorage {
uint256 length,
mapping(uint256 => address) _contractsByIdx,
mapping(address => bool) _contractsByAddress
}
```

Contract Members

Constants & Variables

Address of {TokenManagerRouter}

```
address public tokenManager;
```

Address of {EventEmitterRouter}

```
address public eventEmitter;
```

Address of {TokenValidator}

```
address public tokenValidator;
```

Address of {ElektroResolver} - Resolver for the Elektro market module.

```
contract Resolver public elektroResolver;
```

Variable to prevent subsequent initializations of the module's storage. Can only be set once during the construction of the higher level contract. Setting this var to 'true' should be done in the higher level contract since there might be extra functionality that can potentially fail after initRegistryBase() is complete

```
bool public initialized;
```

See {ContractStorage} struct

```
struct RegistryBaseStorage.ContractStorage internal contractStorage;
```

Modifiers

- onlySelf

onlySelf

Modifier protecting functions that can only be called by the same module.

```
modifier onlySelf() internal
```

Arguments

RegistryBaseAdmin

View Source: @nomisma/elektro-protocol-aux/contracts/registry/RegistryBaseAdmin.sol
↗ **Extends: RegistryBaseStorage, IRegistryBaseAdmin** ↘ **Derived Contracts: RegistryBaseAdminProxy**

RegistryBaseAdmin
Contract is used to register contracts used in our systems and set/initialize storage of the module. Some methods can be only called by registered contracts. It prevents unwanted external method call by third party client Initialization contract for Registry.

Functions

- initRegistryBase(address _elektroResolver, address _tokenManager, address _tokenValidator)

- verify(address contractToVerify)
- isValidContract(address _contract)
- isValidContractOrUtilityBase(address _contract)
- contractsLength()
- getContractByIdx(uint256 idx)
- _isValidContract(address _contract)

initRegistryBase

Initialization function that can be called only once during initial contract setup `initalized` state variable ensures that this function cannot be called again. Initializes the storage of the Registry module. Notice the `_elektroResolver` arg that will save this address to storage to point to the ElektroResolver contract for all Elektro markets.

```
function initRegistryBase(address _elektroResolver, address _tokenManager, address
_tokenValidator) external nonpayable onlyRouterAccess
```

Arguments

Name	Type	Description
_elektroResolver	address	address of Resolver contract to be set in
_tokenManager	address	address of TokenManager contract to be set in
_tokenValidator	address	address of TokenValidator contract to be set in

verify

Adding a new contract to the list of registered contracts.

```
function verify(address contractToVerify) external nonpayable onlyRouterAccess
onlySelf
returns(instanceId uint256)
```

Arguments

Name	Type	Description
contractToVerify	address	address of contract to add to the Registry

isValidContract

View that checks if contract is present in Registry storage

```
function isValidContract(address _contract) external view
returns(bool)
```

Arguments

Name	Type	Description
_contract	address	address of contract to validate

isValidContractOrUtilityBase

View that checks if a contract is present in Registry storage. This provides an access point to a higher level function from ElektroRegistryAdmin.

```
function isValidContractOrUtilityBase(address _contract) public view
returns(bool)
```

Arguments

Name	Type	Description
_contract	address	address of contract to validate

contractsLength

View that returns amount of registered contracts

```
function contractsLength() public view
returns(uint256)
```

Arguments

getContractByIdx

Returning registered contract by idx

```
function getContractByIdx(uint256 idx) public view
returns(address)
```

Arguments

Name	Type	Description
idx	uint256	address of registered contract

_isValidContract

```
function _isValidContract(address _contract) internal view
returns(bool)
```

Arguments

Name	Type	Description
_contract	address	

RegistryBaseSetters

View Source: @nomisma/elektro-protocol-aux/contracts/registry/RegistryBaseSetters.sol
↗ **Extends: RegistryBaseStorage, IRegistryBaseSetters** ↘ **Derived Contracts: RegistryBaseSettersProxy**

RegistryBaseSetters

Contract provides functionality to set major contract addresses in Registry Governor role is needed to set any address in Registry

Contract Members

Constants & Variables

```
bytes32 private constant GOVERNOR_ROLE_NAME;
```

Functions

- setEventEmitter(address _eventEmitter)
- setTokenManager(address _tokenManager)
- setElektroResolver(address _resolver)
- setTokenValidator(address _tokenValidator)

setEventEmitter

Registers {EventEmitter} contract by assigning it's address to the `eventEmitter` variable. Only `governor` can call this function.

```
function setEventEmitter(address _eventEmitter) external nonpayable onlyRole
onlyRouterAccess
```

Arguments

Name	Type	Description
_eventEmitter	address	

setTokenManager

Registers {TokenManager} contract by assigning it's address to the `tokenManager` variable. Only `governor` can call this function.

```
function setTokenManager(address _tokenManager) external nonpayable onlyRole
onlyRouterAccess
```

Arguments

Name	Type	Description
_tokenManager	address	

setElektroResolver

Setting Instance Resolver contract

```
function setElektroResolver(address _resolver) external nonpayable onlyRole
onlyRouterAccess
```

Arguments

Name	Type	Description
_resolver	address	address of Resolver contract to be set in

setTokenValidator

Setting {TokenValidator} contract

```
function setTokenValidator(address _tokenValidator) external nonpayable onlyRole
onlyRouterAccess
```

Arguments

Name	Type	Description
_tokenValidator	address	address of {TokenValidator} contract to be set in

ElektroRegistryStorage

View Source: contracts/elektro/init/ElektroRegistryStorage.sol
↗ **Extends: RegistryBaseStorage** ↘ **Derived Contracts: ElektroRegistryAdmin, ElektroRegistrySetters, IElektroRegistry**

ElektroRegistryStorage

Top level storage declaration for Elektro Registry Module.

Contract Members

Constants & Variables

```
address public fundLock;
```

ElektroRegistryAdmin

View Source: contracts/elektro/init/ElektroRegistryAdmin.sol
↗ **Extends: ElektroRegistryStorage, Delegator, IElektroRegistryAdmin**

ElektroRegistryAdmin

Registry contract of Elektro Protocol. Provides contract registration and verification if contract has been registered. Is responsible for deployment of each Elektro market ({ElektroRouter}).

Contract Members
Constants & Variables

```
bytes32 private constant ADMIN_ROLE_NAME;
```

Signature of the {RegistryBase.initRegistryBase} which will initialize {RegistryBaseStorage} which is a part (inherited) of the {ElektroRegistryStorage} Is needed for delegate calls.

```
bytes4 private constant BASE_INIT_SIG;
```

Functions

- deployElektro(address _underlyingCurrency, address _strikeCurrency, uint256 _precisionUnderlyingCurrency, uint256 _precisionStrikeCurrency)
- initElektroRegistry(address _tokenManager, address _fundLock, address _elektroResolver, address _tokenValidator)
- isValidContractOrUtility(address _contract)

deployElektro

Deployment of each Elektro market ({ElektroRouter}). First Router contract is deployed, next - Router is added as verified contract. Finally deployment of Elektro contract is executed.

```
function deployElektro(address _underlyingCurrency, address _strikeCurrency,
uint256 _precisionUnderlyingCurrency, uint256 _precisionStrikeCurrency) external
nonpayable onlyRole onlyRouterAccess
returns(address)
```

Arguments

Name	Type	Description

_underlyingCurrency	address	address of underlyingCurrency asset used in Elektro contract
_strikeCurrency	address	address of strikeCurrency asset used in Elektro contract
_precisionUnderlyingCurrency	uint256	min precision of base token
_precisionStrikeCurrency	uint256	min precision of underlying token

initElektroRegistry

function that sets crucial contracts addresses used by Registry

```
function  initElektroRegistry(address  _tokenManager,  address  _fundLock,  address
_elektroResolver, address _tokenValidator) external nonpayable onlyRouterAccess
returns(bool)
```

Arguments

Name	Type	Description
_tokenManager	address	address of TokenManager contract to be set in
_fundLock	address	address of FundLock contract to be set in
_elektroResolver	address	address of ElektroResolver contract to be set in
_tokenValidator	address	

isValidContractOrUtility

Function used by some modifiers or to just check if the a certain contract is registered as a part of Elektro Protocol. Protects from unauthorized calls from external contracts outside of our domain.

```
function isValidContractOrUtility(address _contract) external view
returns(bool)
```

Arguments

Name	Type	Description
_contract	address	- address of the contract to check

ElektroRegistrySetters

View Source: contracts/elektro/init/ElektroRegistrySetters.sol
↗ **Extends: ElektroRegistryStorage, IElektroRegistrySetters**

ElektroRegistrySetters

Storage Setter contract for Elektro Registry Proxy Module. Sets state variables declared in the {ElektroRegistryStorage} and stored on {ElektroRegistryRouter}.

Contract Members

Constants & Variables

```
bytes32 private constant GOVERNOR_ROLE_NAME;
```

Functions
- setFundLock(address _fundLock)
- postUpgradeInitialize()

setFundLock

Setter for {FundLockRouter} address

```
function setFundLock(address _fundLock) external nonpayable onlyRole
onlyRouterAccess
```

Arguments

Name	Type	Description
_fundLock	address	

postUpgradeInitialize

Setter for `initialized` state var preventing reinitialization of storage past the point of deployment. This is needed to be able to reset the var during Proxy Pattern smart contracts upgrade.

```
function postUpgradeInitialize() external nonpayable onlyRole onlyRouterAccess
```

Arguments

ElektroRegistryRouter

View Source: contracts/elektro/init/ElektroRegistryRouter.sol
↗ **Extends: Router, Delegator**

ElektroRegistryRouter

Contract used to route calls to ElektroRegistry contracts and functions.

Contract Members

Constants & Variables

Signature of the initialization function of the ElektroRegistry module. Is called by {ElektroRegistryRouter} during its construction to initialize {ElektroRegistryStorage} (set crucial state variables). It is a top part of multiple initialization calls, the lower parts being {RegistryBase.initRegistryBase()}.

```
bytes4 internal constant INIT_SIG;
```

Functions

- (address _resolver, address _roleManager, address[4] addressData)

Constructor initiating Proxy Pattern storage initialization. Besides setting main vars on the Router itself, makes delegatecall to {ElektroRegistryAdmin} to initialize {ElektroRegistryStorage}.

```
function (address _resolver, address _roleManager, address[4] addressData) public nonpayable
```

Arguments

Name	Type	Description
_resolver	address	- address of the ElektroRegistryResolver Notice that contract with this name is not present as code - we just deploy regular Resolver contract and name it ElektroRegistryResolver in the DB to differentiate between it and Resolvers of other proxy modules.
_roleManager	address	- address of the {RoleManager}
addressData	address[4]	- array of address arguments that need to be passed to {initElektroRegistry()} function to make delegatecall for storage initialization.

Funds Management

FundLockStorage

View Source: contracts/fund-lock/FundLockStorage.sol
↗ Extends: RoleAware, Delegator, OnlyRouterAccess, IFundLockStorage ↘ Derived Contracts: FundLockAdmin, FundLockBase, FundLockStateGetters, IFundLock

FundLockStorage

All storage of the `FundLock` module declared.

Structs

Funds

```
struct Funds {
uint256 value,
uint256 timestamp
}
```

Contract Members

Constants & Variables

Variable used for upgrades. Represents the fact that this storage was already initalized. If this variable is "true", {FundLockAdmin.init} will revert, not letting subsequent initializations.

```
bool public initialized;
```

The maximum amount of {FundLockUser.withdraw} requests one client can make per one token.

```
uint256 public constant ALLOWED_WITHDRAWAL_LIMIT;
```

Address of the {TokenManagerRouter}

```
contract ITokenManagerMain public tokenManager;
```

Address of the {ElektroRegistryRouter}

```
contract IElektroRegistry public registry;
```

This interval represents the time that user has to wait after he called {FundLockUser.withdraw} function before he can {FundLockUser.release} funds to his wallet.This allows the backend to react and ensure all options are always covered. The time lock also ensures the client has eventual access to its funds if the backend were to cease being operational. The backend cannot prevent a withdrawal indefinitely.

```
uint256 public releaseLock;
```

Time period during which user's {fundsToWithdrawS} can still be used for funding Elektro (new trades or covering old trades).

```
uint256 public tradeLock;
```

Mapping representing all {FundLockUser.withdraw} requests a client made. userAddress => tokenAddress => array of `Funds` structs

```
mapping(address => mapping(address => struct FundLockStorage.Funds[5])) internal
fundsToWithdrawS;
```

Main storage mapping representing all Elektro clients balances at any given time in the tokens of Elektro. beneficiaryAddress => tokenAddress => amount

```
mapping(address => mapping(address => uint256)) internal balanceSheetS;
```

IFundLockEvents

View Source: contracts/fund-lock/IFundLockEvents.sol
↘ **Derived Contracts: FundLockMock, IElektroEvents, IFundLockStorage**

IFundLockEvents

FundLockDeposit

Event fired after successful deposit to {FundLock}.

Parameters

Name	Type	Description
depositor	address	- address of the depositor (can be contract or user)
token	address	- address of the token deposited
amount	uint256	- amount of the token deposited See {FundLockBase._deposit()}

FundsToBeWithdrawn

Event fired with user's {withdraw()} request, which marks funds for release from FundLock.Do not confuse with {release()}!

Parameters

Name	Type	Description
beneficiary	address	- address of a user withdrawing

token	address	- address of the token to be marked for release
amount	uint256	- amount of token to be marked for release
totalSlotsUsed	uint256	- the amount of withdrawal requests (slots) used (out of 5 max) Each user is only allowed to make 5 withdrawal requests at once. See {FundLockUser.withdraw()}

FundsReleased

Event fired upon releasing funds (tokens) from FundLock to a user's wallet.

Parameters

Name	Type	Description
beneficiary	address	- address of the user to whom funds are released
token	address	- address of the token released
amount	uint256	- amount of token released
withdrawTimestamp	uint256	- time in seconds at which {withdraw()} request has been made See {FundLockUser.release()}

ReleaseLockSet

Event fired upod setting the {ReleaseLock} interval.

Parameters

Name	Type	Description
interval	uint256	- the amount of time in seconds which user has to wait between {withdraw()} and {release()}

TradeLockSet

Event fired upod setting the {TradeLock} interval.

Parameters

Name	Type	Description
interval	uint256	- the amount of time in seconds during which user's funds can still be used for trading after {withdraw()} request

BalanceUpdated

Event fired upon successful client balance update, which signifies the end of the settlement transaction.

Parameters

Name	Type	Description
user	address	- client's address for which balance has been updated
amount	int256	- amount by which balance has been updated
token	address	- token address for which balance has been updated
backendId	uint64	- Java Backend batch ID used for tracking

FundLockStateGetters

State Getters of the FundLock module.

View Source: contracts/fund-lock/FundLockStateGetters.sol
↗ **Extends: FundLockStorage**

FundLockStateGetters

Specific getter functionality for the ease of use. Overriding (covering) some of the {FundLockStorage} getters to provide more appropriate/parsed data.

Functions

- balanceSheet(address userAddress, address tokenAddress)
- fundsToWithdraw(address userAddress, address tokenAddress, uint256 index)
- fundsToWithdrawTotal(address beneficiary, address token)
- isEtherToken(address tokenAddress)
- _getIsEtherAndAssetAddress(address tokenAddress)

balanceSheet

Getter returning the correct {balanceSheetS} value for the correct token and particular client.In cases of `ETH` usage it will convert the `ETH` address to `WETH9` address and read the appropriate balance. See {getIsEtherAndAssetAddress}

```
function balanceSheet(address userAddress, address tokenAddress) external view
returns(uint256)
```

Returns
uint256 - balance amount

Arguments

Name	Type	Description
userAddress	address	- address of the user to check balance for
tokenAddress	address	- top level token address used (`ETH` for cases with Ether)

fundsToWithdraw

Getter used to pull data about client's withdraw requests.Wraps autogenerated getter for {fundsToWithdrawS} to find a proper token address (in the case of Ether) and read the correct storage slot of the mapping. This will return only ONE request at an index provided.

```
function fundsToWithdraw(address userAddress, address tokenAddress, uint256 index)
external view
returns(value uint256, timestamp uint256)
```

Returns
value - the amount of funds marked for withdrawal

Arguments

Name	Type	Description
userAddress	address	- address of the client who sent withdrawal request(s)

tokenAddress	address	- address of the token withdrawn
index	uint256	- index of the particular withdrawal request (max of 5)

fundsToWithdrawTotal

Returns the total amount of token flagged for withdrawal for a particular user that still has active {tradeLock}. This is the SUM of all withdraw requests for a particular token and user.

```
function fundsToWithdrawTotal(address beneficiary, address token) external view
returns(uint256)
```

Returns
uint256 - total amount of client's funds marked for withdrawal

Arguments

Name	Type	Description
beneficiary	address	- address of the user to get balance for
token	address	- address of the token to get balance for

isEtherToken

Checks if the provided token is Ether.

```
function isEtherToken(address tokenAddress) internal view
returns(bool)
```

Arguments

Name	Type	Description
tokenAddress	address	

_getIsEtherAndAssetAddress

Internal function used by other getters to determine if the provided token is Ether and return a proper token address used by {FundLockStorage}.

```
function _getIsEtherAndAssetAddress(address tokenAddress) internal view
returns(isEther bool, assetAddress address)
```

Arguments

Name	Type	Description
tokenAddress	address	

FundLockBase

View Source: contracts/fund-lock/FundLockBase.sol
↗ Extends: FundLockStorage ↘ Derived Contracts: FundLockTrade, FundLockUser

FundLockBase

Base contract, which functionality could be used by several contracts inside the FundLock group.

Functions

- _deposit(address tokenAddress, uint256 value)
- _withdraw(address tokenAddress, uint256 value)

_deposit

Internal function used by {FundLockUser.deposit()}. Does general depositing logic: check if the asset is Ether, validates amount and token, performs token transfer to FundLock, updates {balanceSheetS} storage and fires {FundLockDeposit} event.

```
function _deposit(address tokenAddress, uint256 value) internal nonpayable
returns(bool)
```

Arguments

Name	Type	Description
tokenAddress	address	- address of the token to deposit
value	uint256	- the amount of token to deposit

_withdraw

Marking funds that need to be withdrawn by the client. This function does NOT do any transfers. It only marks funds by creating `Funds` structs and mapping them to user and token addresses. Also changes `balanceSheet` and emits event. Created for security purposes.

```
function _withdraw(address tokenAddress, uint256 value) internal nonpayable
returns(bool)
```

Arguments

Name	Type	Description
tokenAddress	address	- address of the token to be marked for withdrawal
value	uint256	- amount of the token to be marked for withdrawal

FundLockAdmin

View Source: contracts/fund-lock/FundLockAdmin.sol
↗ **Extends: FundLockStorage, IFundLockAdmin**

FundLockAdmin

Administrative contract that initializes storage of FundLock group and sets main state variables.

Contract Members
Constants & Variables

```
bytes32 private constant ADMIN_ROLE_NAME;
```

```
bytes32 private constant GOVERNOR_ROLE_NAME;
```

Functions
- init(address _roleManager, address _tokenManager, uint256 _tradeLock, uint256 _releaseLock)
- setRegistry(address _registry)
- setReleaseLockInterval(uint256 interval)
- setTradeLockInterval(uint256 interval)

init

Main initialization function that checks if initialization has been previously done and sets storage vars.

```
function init(address _roleManager, address _tokenManager, uint256 _tradeLock,
uint256 _releaseLock) external nonpayable onlyRouterAccess
```

Arguments

Name	Type	Description
_roleManager	address	- address of the {RoleManager} contract
_tokenManager	address	- address of the {TokenManager} contract

_tradeLock	uint256	- time period during which user's {fundsToWithdrawS} can still be used for funding trades
_releaseLock	uint256	- time period after withdrawal request a user has to wait to release his funds (between {withdraw()} and {release()})

setRegistry

Setter for {registry} storage var which holds {ElektroRegistryRouter} address

```
function setRegistry(address _registry) external nonpayable onlyRouterAccess
onlyRole
```

Arguments

Name	Type	Description
_registry	address	- {ElektroRegistryRouter} address to be set

setReleaseLockInterval

Sets {releaseLock} interval variable on the contract that will be used for all {fundsToWithdraw}. This interval represents time that user has to wait after he called {withdraw()} function before he can {release()} funds. Can only be called by the Admin account.

```
function setReleaseLockInterval(uint256 interval) public nonpayable
onlyRouterAccess onlyRole
returns(bool)
```

Arguments

Name	Type	Description
interval	uint256	- {releaseLock} interval to be set

setTradeLockInterval

Sets {tradeLock} interval variable on the contract that will be used for all {fundsToWithdraw}. This interval represents time during which user's {fundsToWithdraw} can still be used to cover trades. Starts after {withdraw()} is called, at the same time as {releaseLock}, but is shorter. Can only be called by the Admin account.

```
function setTradeLockInterval(uint256 interval) public nonpayable onlyRouterAccess
onlyRole
```

```
returns(bool)
```

Arguments

Name	Type	Description
interval	uint256	- {tradeLock} interval to be set

FundLockUser

View Source: contracts/fund-lock/FundLockUser.sol
↗ **Extends: FundLockBase, IFundLockUser**

FundLockUser

FundLock contract responsible for all user related token operations like deposits, withdrawals and funds release to a wallet.

Contract Members

Constants & Variables

```
bytes32 private constant UTILITY_ACCOUNT_ROLE_NAME;
```

Functions

- deposit(address tokenAddress, uint256 value)
- withdraw(address tokenAddress, uint256 value)
- release(address tokenAddress, uint256 withdrawTimestamp)

deposit

User facing function for depositing funds to be used in Elektro. `value` deposited will go to the client's {balanceSheetS}.

```
function deposit(address tokenAddress, uint256 value) external payable
onlyRouterAccess
returns(bool)
```

Arguments

Name	Type	Description
tokenAddress	address	- address of the token to be deposited
value	uint256	- amount of the token to be deposited

withdraw

Marking funds that need to be withdrawn by the depositor. This function does NOT do any transfers. It only marks funds by creating `Funds` structs and mapping them to user and token addresses. Also changes {balanceSheetS} and emits event. Created for security purposes.

```
function withdraw(address tokenAddress, uint256 value) external nonpayable
onlyRouterAccess
returns(bool)
```

Arguments

Name	Type	Description
tokenAddress	address	address of the token to be marked for withdrawal
value	uint256	amount of the token to be marked for withdrawal

release

Actual withdrawal through releasing funds and making transfer to the client who called the function. In order to be released, funds first have to be marked as {fundsToWithdrawS} by the {withdraw()} function and time interval set as {releaseLock} has to pass since withdraw() was called. Uses different transfer logic depending on which token is used (ETH/WETH/ERC20).

```
function release(address tokenAddress, uint256 withdrawTimestamp) external
nonpayable onlyRouterAccess
returns(bool)
```

Arguments

Name	Type	Description
tokenAddress	address	- address of the token to be released
withdrawTimestamp	uint256	- timestamp associated with a certain withdraw request (can be found in {Funds} struct included in the {fundsToWithdrawS} mapping).

FundLockTrade

Elektro triggered the functionality of the FundLock.

View Source: contracts/fund-lock/FundLockTrade.sol
↗ **Extends: FundLockBase, IFundLockTrade**

FundLockTrade

Elektro trading related contract of FundLock group. Updates {balanceSheetS} and {fundsToWithdrawS} upon trade settlement, to finalize and manage client funds. Last stop for any trade settlement transaction.All functions besides balance getter can only be called by a registered Elektro contract.

Modifiers
- isValidContract

isValidContract

Uses `isValidContractOrUtility` function of `RegistryBase` contract to verify that a caller of the function using this modifier is a valid Elektro contract.

```
modifier isValidContract() internal
```

Arguments

Functions
- updateBalances(address[] traders, int256[] amounts, address[] tokens, uint64 backendId)
- fundFromWithdrawn(address beneficiary, address token, uint256 toFundAmt)
- _updateBalance(address trader, address assetAddress, int256 amount)

updateBalances

Public function used by Elektro to update `balanceSheetS` for all types of accounts. Used by LedgerUpdate flows. Updates trader balances by calling single update function {_updateBalance}, emits event.

```
function updateBalances(address[] traders, int256[] amounts, address[] tokens,
uint64 backendId) external nonpayable onlyRouterAccess isValidContract
returns(bool)
```

Arguments

Name	Type	Description
traders	address[]	- array of all clients accounts of the settlement batch
amounts	int256[]	- array of amounts of funds to update client balances with

tokens	address[]	- array of token addresses for each update row
backendId	uint64	- identificator created by Java Backend to track settlement progress

fundFromWithdrawn

Private function that is a part of {_updateBalance}. This function is only called when user's funds in {balanceSheets} are not enough and his {fundsToWithdrawS} can be used.Checks every struct bound to user and a particular token for trade availability (if {tradeLock} interval hasn't yet passed). Changes values of all necessary structs according to funding amount. Zeros out {balanceSheetS} if all structs have been changed and all funds are used.

```
function fundFromWithdrawn(address beneficiary, address token, uint256 toFundAmt)
internal nonpayable
returns(funded bool)
```

Returns

funded - a boolean representing if there were enough funds to fully pay for what's needed, if it's false, the transaction using this flow will fail because this call is usually wrapped in a require()

Arguments

Name	Type	Description
beneficiary	address	- address of a client whose funds we're using.
token	address	- address of a token to use.
toFundAmt	uint256	- total amount to fund from all {fundsToWithdrawS}.

_updateBalance

Internal low level function to update a single entry in the {balanceSheetS} for a client. Checks if a client has enough funds in his {balanceSheetS} for a particular token, if so - that balance is used to cover his trade, if not - call to {fundFromWithdrawS} to check and use his funds marked for withdrawal that are still viable for usage ({tradeLock} is on). If those funds are not enough as well - revert, otherwise, his {fundFromWithdrawS} becomes smaller.

```
function _updateBalance(address trader, address assetAddress, int256 amount)
private nonpayable
```

Arguments

Name	Type	Description

trader	address	- client's address to update balance for
assetAddress	address	- address of the token used
amount	int256	- signed int which is used to update balance

FundLockRouter

View Source: contracts/fund-lock/FundLockRouter.sol
↗ **Extends: Router, Delegator**

FundLockRouter

Main address called for all `FundLock` functionality. Initializes storage and routes calls to any `FundLock` related functions.

Contract Members

Constants & Variables

Function signature to delegatecall() {FundLockAdmin} implementation contract since we need a "construction" at a later time to tie it to the {FundLockRouter}

```
bytes4 internal constant INIT_SIG;
```

Functions

- (address _roleManager, address _resolver, address _tokenManager, uint256 _tradeLock, uint256 _releaseLock)

Constructor which will initialize parent {Router} contract and trigger initialization of the implementation {FundLockAdmin} contract and {FundLockStorage} through delegatecall.

```
function (address _roleManager, address _resolver, address _tokenManager, uint256
_tradeLock, uint256 _releaseLock) public nonpayable
```

Arguments

Name	Type	Description
_roleManager	address	- address of the {RoleManager} contract
_resolver	address	- address of the respective {FundLockResolver}
_tokenManager	address	- address of the {TokenManagerRouter}

_tradeLock	uint256	- interval to use {fundsToWithdrawS} before {release} see {tradeLock}
_releaseLock	uint256	- interval to lock funds before {release} see {releaseLock}

Elektro Market

ElektroStorage

View Source: contracts/elektro/base/ElektroStorage.sol
↗ **Extends: RoleAware, Resolvable, OnlyRouterAccess** ↘ **Derived Contracts: ElektroAbstract, ElektroInitialize, ElektroLedgerCommon, ElektroLedgerUpdate, ElektroSetters**

ElektroStorage

Main Storage contract for each Elektro market. One of these is used for every market (asset pair) deployed. This stores data necessary for any Elektro market functionality and is tied to every {ElektroRouter}.

Structs

CurrencyPair

```
struct CurrencyPair {
address underlyingCurrency,
address strikeCurrency,
uint256 srcMultiplier,
uint256 destMultiplier
}
```

Contract Members

Constants & Variables

Boolean used to block subsequent initializations for Elektro market storage. Makes sure contract's storage has been initialized and check it upon every call to init().

```
bool public initialized;
```

```
contract ITokenManagerMain public tokenManager;
```

```
contract IElektroEventEmitter public elektroEventEmitter;
```

```
contract IElektroRegistry public registry;
```

```
address public underlyingCurrency;
address public strikeCurrency;
```

Mapping storing client positions for a specific Elektro market. Map<{uint32 contractId, address clientAddress}, int256 amount>

```
mapping(uint32 => mapping(address => int64)) public clientPositions;
```

Functions

- getChainID()

getChainID

Returns CHAINID from the opcode by using inline assembly.

```
function getChainID() external pure
returns(uint256)
```

Returns

uint256 - chainid representing which Ethereum network (chain) this contract belongs to (Ropsten, Mainnet, etc.)

Arguments

ElektroSetters

View Source: contracts/elektro/base/ElektroSetters.sol
↗ **Extends: ElektroStorage, IElektroSetters**

ElektroSetters

Contract Members

Constants & Variables

Important all role strings have to match across contracts!

```
bytes32 private constant GOVERNOR_ROLE_NAME;
```

Functions

- setElektroEventEmitter(address eventEmitter)

setElektroEventEmitter

```
function setElektroEventEmitter(address eventEmitter) external nonpayable onlyRole
onlyRouterAccess
```

Arguments

Name	Type	Description
eventEmitter	address	

ElektroInitialize

View Source: contracts/elektro/init/ElektroInitialize.sol
↗ **Extends: ElektroStorage, IElektroInitialize**

ElektroInitialize
Contract repsonsible for initialization of {ElektroStorage}.

Functions
- init(address _underlyingCurrency, address _strikeCurrency, address _eventEmitter, address _tokenManager)

init

Main initialization function of the storage for each Elektro market ({ElektroRouter}). Will set all necessary state variables in {ElektroStorage}. It is called by {ElektroRegistry} during the construction/creation of each Elektro market ({ElektroRouter}).

```
function init(address _underlyingCurrency, address _strikeCurrency, address
_eventEmitter, address _tokenManager) external nonpayable onlyRouterAccess
returns(bool)
```

Arguments

Name	Type	Description
_underlyingCurrency	address	- the underlying asset of Elektro market
_strikeCurrency	address	- the strike/price asset of Elektro market
_eventEmitter	address	- address of the {ElektroEventEmitterRouter}
_tokenManager	address	- address of the {TokenManagerRouter}

ElektroLedgerCommon

View Source: contracts/elektro/ledger/ElektroLedgerCommon.sol
↗ **Extends: ElektroStorage, ElektroMultiplierAware** ↘ **Derived Contracts:**

ElektroLedgerUpdate

ElektroLedgerCommon

Some base functionality separated into a separate contract inherited by {ElektroLedgerUpdate}. Can be used to add common functionality in the case of multiple implementation Ledger contracts.

Contract Members

Constants & Variables

```
bytes32 internal constant UTILITY_ACCOUNT_ROLE_NAME;
```

Functions

- getCurrencyPair()

getCurrencyPair

Getter for a {CurrencyPair} struct. Returns addresses for the asset pair and their respective multipliers needed to normalize token amounts - from precision denominated values to actual token decimals denominated values.

```
function getCurrencyPair() internal view
returns(currencyPair struct ElektroStorage.CurrencyPair)
```

Arguments

ElektroLedgerUpdate

View Source: contracts/elektro/ledger/ElektroLedgerUpdate.sol
↗ **Extends: ElektroStorage, ElektroLedgerCommon, IElektroLedgerUpdate**

ElektroLedgerUpdate

Main functionality of an Elektro market. Contract for settling trades based on the data passed from Java Backend Engine. Some data validation, calculations and calls to FundLock Module to make transfers and update token balances.

Functions

- updatePositions(address[] positionClients, uint32[] positionContractIds, int64[] positionSizes, address[] fundMovementClients, int64[] underlyingAmounts, int64[] strikeAmounts, uint64 backendId)
- validateAndCountAmounts(int64[] underlyingAmounts, int64[] strikeAmounts)
- processPositionUpdates(address[] clientAddresses, uint32[] positionContractIds, int64[] positionSizes)
- processFundMovement(address[] fundMovementClients, int64[] underlyingAmounts, int64[] strikeAmounts, uint64 backendId, uint256 transferCount)
- initializeData(address[] fundMovementClients, int64[] underlyingAmounts, int64[] strikeAmounts, uint256 transferCount)

updatePositions

Entry point for all settlement logic and the function that Java Backend calls to trigger trade settlement. Checks arguments, updates position balances for a market in {ElektroStorage}, normalizes values to token denomination and calls FundLock to update token balances for each client. Arguments are broken down in two groups:

1. Position update in ElektroStorage,
2. Fund movements/client balances update in FundLockStorage.

Please note: Signs of the values for both of the below amount args are inverted, meaning that a negative amount would be added to an existing client balance, while a positive value is going to be subtracted! Negative means client receives tokens, positive means he pays tokens.

```
function updatePositions(address[] positionClients, uint32[] positionContractIds, int64[] positionSizes, address[] fundMovementClients, int64[] underlyingAmounts, int64[] strikeAmounts, uint64 backendId) external nonpayable onlyRole onlyRouterAccess
```

Arguments

Name	Type	Description
positionClients	address[]	- addresses of the traders for whom positions are being changed
positionContractIds	uint32[]	- contractIds for each position (representation of a trade contract under which we are settling)
positionSizes	int64[]	- difference number representing how a position is going to change Negative position is going to be subtracted from an existing one, while a positive is going to be added.
fundMovementClients	address[]	- addresses of the client for which funds are going to be moved/updated
underlyingAmounts	int64[]	- signed amounts of fund movements in underlying asset
strikeAmounts	int64[]	- signed amounts of fund movements in strike/price asset
backendId	uint64	- identificator created by Java Backend to track settlement progress

validateAndCountAmounts

Function that validates that there are no fully zero rows present and counts how many transfers will be needed when reaching FundLock, so we can create correct length transfer arrays for FundLock.

```
function validateAndCountAmounts(int64[] underlyingAmounts, int64[] strikeAmounts)
internal pure
returns(transferCount uint256)
```

Returns
transferCount - the amount of non zero amounts that would make a valid transger in FundLock

Arguments

Name	Type	Description
underlyingAmounts	int64[]	- see {updatePositions()}
strikeAmounts	int64[]	- see {updatePositions()}

processPositionUpdates

Function for updating `clientPositions` mapping in {ElektroStorage}

```
function processPositionUpdates(address[] clientAddresses, uint32[]
positionContractIds, int64[] positionSizes) internal nonpayable
```

Arguments

Name	Type	Description
clientAddresses	address[]	- addresses of the traders for whom positions are being changed
positionContractIds	uint32[]	- contractIds for each position (representation of a trade contract under which we are settling)
positionSizes	int64[]	- difference number representing how a position is going to change

processFundMovement

Preparation of the "fund movement" part of the data where we take initial arrays and convert them into arrays that FundLock can understand. Creating new arrays, inverting and normalizing values to token denomination for the upcoming transfers in FundLock + sending these arrays to FundLock for further settlement.

```
function processFundMovement(address[] fundMovementClients, int64[] underlyingAmounts, int64[] strikeAmounts, uint64 backendId, uint256 transferCount) internal nonpayable
```

Arguments

Name	Type	Description
fundMovementClients	address[]	- addresses of the client for which funds are going to be moved/updated
underlyingAmounts	int64[]	- signed amounts of fund movements in underlying asset
strikeAmounts	int64[]	- signed amounts of fund movements in strike/price asset
backendId	uint64	- identificator created by Java Backend to track settlement progress
transferCount	uint256	- the amount of transfers between balances that will be performed by FundLock, needed to create non-dynamic memory arrays to fill with prepared data

initializeData

The actual data preparation function in order to move settlement further to FundLock. Creates new arrays, gets {CurrencyPair} to further perform amounts conversion to token denomination and inverts signs for proper balance updates. Only non-negative values are used.

```
function initializeData(address[] fundMovementClients, int64[] underlyingAmounts, int64[] strikeAmounts, uint256 transferCount) internal view
returns(traders address[], amounts int256[], tokens address[])
```

Returns
traders - array of clients/traders for whom FundLock balances will be updated

Arguments

Name	Type	Description

fundMovementClients	address[]	- addresses of the client for which funds are going to be moved/updated
underlyingAmounts	int64[]	- signed amounts of fund movements in underlying asset
strikeAmounts	int64[]	- signed amounts of fund movements in strike/price asset
transferCount	uint256	- the amount of transfers between balances that will be performed by FundLock, needed to create non-dynamic memory arrays to fill with prepared data

ElektroMultiplierAware

View Source: contracts/elektro/utils/ElektroMultiplierAware.sol
↘ **Derived Contracts: ElektroLedgerCommon**

ElektroMultiplierAware

Functionality to get multipliers and precision for market assets. src - underlying asset dest - strike (premium) asset multiplier - 10^(decimals - precision) These functions get values from TokenValidator contract by asset address.

Structs
PrecisionsMultipliers

```
struct PrecisionsMultipliers {
uint256 srcMultiplier,
uint256 destMultiplier,
uint256 srcPrecision,
uint256 destPrecision
}
```

Functions
- getDestSrcPrecisions(address underlyingCurrency, address strikeCurrency, address tokenValidator)

getDestSrcPrecisions

```
function getDestSrcPrecisions(address underlyingCurrency, address strikeCurrency,
address tokenValidator) internal view
returns(precisionDest uint256, toTokenPowerDest uint256, precisionSrc uint256,
toTokenPowerSrc uint256)
```

Arguments

Name	Type	Description
underlyingCurrency	address	
strikeCurrency	address	
tokenValidator	address	

ElektroRouter

View Source: contracts/elektro/init/ElektroRouter.sol
↗ **Extends: Router**

ElektroRouter
The official holder of Elektro address. Each Elektro is represented by ElektroRouter address and if inside any Elektro contract we write `address(this)` we will get address of this contract

Functions
- (address _resolver, address _roleManager)

```
function (address _resolver, address _roleManager) public nonpayable
```

Arguments

Name	Type	Description
_resolver	address	
_roleManager	address	

BaseEventEmitter

View Source: @nomisma/elektro-protocol-aux/contracts/event-emitter/BaseEventEmitter.sol
↘ **Derived Contracts: ElektroEventEmitter**

BaseEventEmitter

Base event emitter functionality that is extended/inherited by Elektro Protocol. Contains initialization function that ensures initialization is done only once and sets Registry contract address.

Contract Members

Constants & Variables

```
address internal registryAddress;
```

```
bool private initialized;
```

Modifiers
- isValidContract

isValidContract

```
modifier isValidContract() internal
```

Arguments
Functions
- initBaseEventEmitter(address _registry)

initBaseEventEmitter

Initializes base event emitter by providing registry address

```
function initBaseEventEmitter(address _registry) public nonpayable
```
Arguments

Name	Type	Description
_registry	address	address of {Registry} contract

ElektroEventEmitter

View Source: contracts/elektro/event-emitter/ElektroEventEmitter.sol
↗ **Extends: BaseEventEmitter, OnlyRouterAccess, IElektroEventEmitter**

ElektroEventEmitter

Main and the only contract of the Elektro for firing events. Simple structure where each function is firing a specific event. Made for the purpose of centralization of event management. Events for all existing and all new Elektro contracts should project, declare and fire events through this contract ONLY. See For all events docs please see {IElektroEventEmitterEvents}

Functions
- initElektroEventEmitter(address _registry)
- emitLedgerPositionMoved(uint64 backendId, uint32[] contractIds, address[] users, int64[] positionSizes)

initElektroEventEmitter

Function to initialize storage through Proxy Pattern module construction.

```
function initElektroEventEmitter(address _registry) external nonpayable
onlyRouterAccess
```

Arguments

Name	Type	Description
_registry	address	- address of {ElektroRegistryRouter}

emitLedgerPositionMoved

Function that emits 2 types of events upon each trade settlement. {PositionsUpdated} event is fired once per each settlement for a `backendId`. {LedgerPositionMoved} event fired for each row (user funds transfer) in a batch.

```
function emitLedgerPositionMoved(uint64 backendId, uint32[] contractIds, address[]
users, int64[] positionSizes) external nonpayable isValidContract
returns(bool)
```

Arguments

Name	Type	Description
backendId	uint64	- special identifier created by Java Backend to track settlement progress for each batch.
contractIds	uint32[]	- array of contractIds of the whole settlement batch
users	address[]	- address array for all user participating in the settlement
positionSizes	int64[]	- quantities of each position for every user to be settled

IElektroEventEmitterEvents

View Source: contracts/elektro/event-emitter/IElektroEventEmitterEvents.sol
↘ **Derived Contracts: IElektroEventEmitter, IElektroEvents**

IElektroEventEmitterEvents

Interface with declarations of all events specific to Elektro Markets. This does NOT include FundLock events. For those, see {IFundLockEvents}

LedgerPositionMoved

Includes information about each particular position change for every client/position participating in the settlement.

Parameters

Name	Type	Description
contractId	uint32	- speficial identificator signifying a trade contracts, that includes asset pair, strike price, maturity date and such, created and outlined on the Java Backend Engine.
user	address	- address of a client who's position is being settled
positionSize	int64	- outlines the position amount diff by which position is changed

PositionsUpdated

Signifies a successful settlement, emitted once per settlement batch. Shows that a settlement batch with a particular `backendId` has been settled.

Parameters

Name	Type	Description
backendId	uint64	- special identificator created by Java Backend to track settlement progress

ElektroEventEmitterRouter

View Source: contracts/elektro/event-emitter/ElektroEventEmitterRouter.sol

↗ **Extends: Router, Delegator**

ElektroEventEmitterRouter

Router for ElektroEventEmitter group (currently 1 impl contract). Created to eliminate possibilities of changing ElektroEventEmitter address in the event of redeployment (upgrade). Since Router contracts do not get redeployed (only implementation contracts do) even when changing functionality or deploying new implementation contracts the address of ElektroEventEmitter saved in Elektro will not change and will keep being this contract's address. This will let us keep all of the events no matter how many changes will be performed to implementations. Initializes Router first then initializes ElektroEventEmitter with registry address through `delegatecall()`. This initialization is a substitute for constructor because Proxy Pattern is used.

Contract Members

Constants & Variables

```
bytes4 internal constant INIT_SIG;
```

Functions

- (address _resolver, address _roleManager, address _registry)

```
function (address _resolver, address _roleManager, address _registry) public nonpayable
```

Arguments

Name	Type	Description
_resolver	address	
_roleManager	address	
_registry	address	