Elektro Protocol

Smart Contracts for transaction settlement, client fund management and token transfers

---

Contracts

Elektro Initialize

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

This is the registry of the `elektro-protocol`. Despite it's 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. It’s storage also holds addresses of some of the auxiliary contracts in the system (TokenManager, TokenValidator, etc.).

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`.

Elektro Event Emitter Module

Module responsible for all trade settlement related events.

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.

Utils

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

---

elektro-protocol-aux

General

Repo for auxilliary smart contracts of Elektro Protocol.

This repo contains only Ethereum smart contracts related to the Elektro Protocol and does NOT include main trading-settlement contracts. Also includes general helpers written in JS to be used for testing.

Technical architecture

Proxy Pattern and Smart contract upgradeability

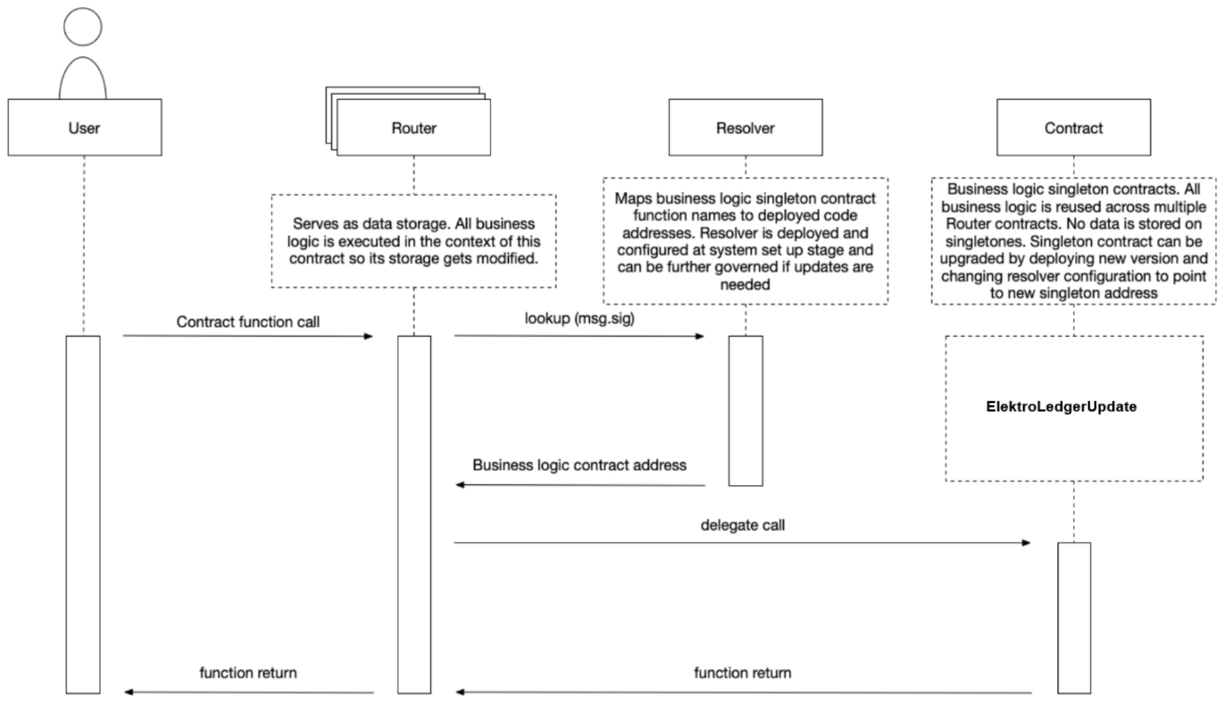
Contracts upgrades are handled using **Delegatecall-based proxy pattern** with more fine grained control over contract methods upgrades.

The key players are:

- **Router** - Keeps data storage. Delegates method invocation call to concrete contract logic implementation. Method is executed in the context of router by using `delegatecall` message call.
- **Resolver** - Maps business logic method names to address of deployed contract using first 4 bytes of Keccak hash (SHA3) of method signature.
- **Contract logic** - Concrete implementation of business logic. No data is stored in this contract. Can be upgraded by deploying new contract with modified logic and update signature in Resolver to point to new deployed contract.

Interaction

Following diagram shows contract interaction:


User
Router
Resolver
Contract
Serves as data storage. All business logic is executed in the context of this contract so its storage gets modified.
Maps business logic singleton contract function names to deployed code addresses. Resolver is deployed and configured at system set up stage and can be further governed if updates are needed
Business logic singleton contracts. All business logic is reused across multiple Router contracts. No data is stored on singletones. Singleton contract can be upgraded by deploying new version and changing resolver configuration to point to new singleton address
Contract function call
lookup (msg.sig)
ElektroLedgerUpdate
Business logic contract address
delegate call
function return
function return

1. User calls Router contract method eg. `ElektroSetters.setElektroEventEmitter(...)`
2. Router using first four bytes of Keccak hash of `setElektroEventEmitter(address)` calls Resolver to get address of deployed contract logic.
3. Resolver returns address of deployed contract logic.
4. Router use `delegatecall` in fallback method on resolved address to execute logic of `setElektroEventEmitter(...)` in Router's context (data is stored on Router, not on deployed contract)
5. `ElektroSetters.setElektroEventEmitter(...)` is executed and returns transaction data to Router.
6. Transaction data is returned from Router to calling User.

Signature conflicts

Some signatures can have same first 4 bytes of Keccak hash eg. `tgeo()` and `gsf()`.

If during upgrade we detect conflict we check Keccak Hash. If the same Keccak hash is passed as param, upgrade is allowed. If Keccak hash is different upgrade transaction is reverted.

Core contracts

Proxy Pattern Contracts

- Router
- Resolver
- Delegator

See picture and description above.

RoleManager

Access management and validation contract used by Elektro Protocol. Manages Governance and Backend access to protected Elektro functions.

Registry

Used by user with `governance` role to deploy Router for Elektro contracts logic and keep track of valid deployed contracts. Also used as storage for some important aux contract addresses of the system.

TokenManager

Responsible for collecting funds from clients, determining ETH and WETH, getting token properties. Stores ethereumAddress, weth9 address and TokenWrapper. Works in conjunction with TokenWrapper to perform safe transfers.

TokenValidator

Stores whitelist of tokens that can be used by Elektro for trading and validating settlements against this whitelist.

TokenWrapper

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.

TokenWrapper should always be used when calling operations on external tokens.

FundLock Module

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

---

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.

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)

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 **"Client withdrawal process"** 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 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:

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

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 withdraw’** can be active (funds not yet released) at a time. Set to a max of 5 requests!
- `fundsToWithdrawS` - mapping to store user **‘request to withdraw’**. 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