Features

1. **Ownable**. Each account contract has one owner. The owner could be EOA or another smart contract agent.
2. **Upgradable**. The user can upgrade its account contract to a new implementation.
3. **Fixed code**. The address of each account is fixed even after upgrading.
4. **Native gas payment**. The user can pay gas with ETH the account contract holds.
5. **Gas sponsorship**. The user can specify the gas sponsor to pay gas for the user.
6. (**Optional**) Social Recovery. The user can reset the contract owner from guardians he/she selects.
7. (**Optional**) Users add add/remove functionalities by adding/removing plugins.

Implementation

Address Computation

We use **CREATE2**(EIP-1014) to deploy our account contract for users.

```
keccak256( 0xff ++ address ++ salt ++ keccak256(init_code))[12:]

address: the address of deployer contract
salt: the hash of the identifier(email address, twitter handle e.t.c)
init_code: the code of the account contract
```

All these three items need to be finalized since the address cannot be changed once we release our very first version.

Clone

To minimize the cost to deploy a contract, we take advantage of EIP-1167. The code deployed would be:

```
363d3d373d3d3d363d73<20 bytes logic contract
address>5af43d82803e903d91602b57fd5bf3
```

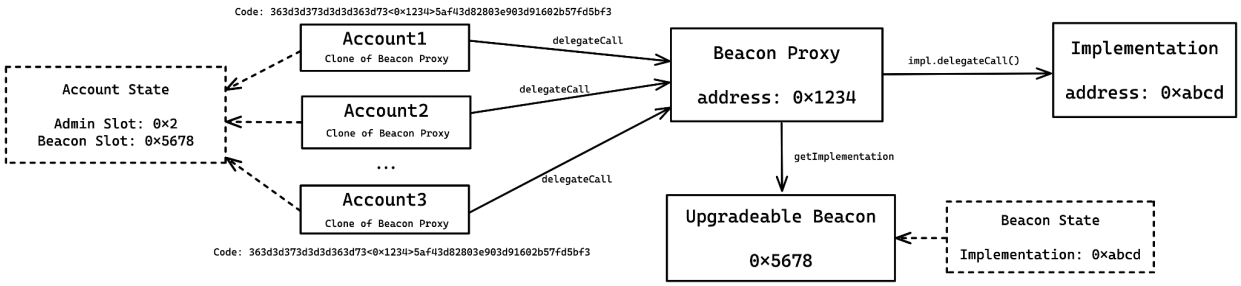
All requests will be forwarded to the logic contract with delegate calls. The logic contract address cannot be changed once it's deployed.

Beacon Proxy Model

We use the beacon proxy model to manage all our contracts. In this way, we can decouple the implementation contract and the account contract to deploy. The beacon proxy contract is immutable but users can reset the beacon to any contract they are comfortable with.

We use beacon proxy mode so we can upgrade the default implementation for any feature enhancement. Once the account contract implementation is stabilized, we will remove our capability of upgrading the default implementation so only users themselves will be able to upgrade their account contracts.

The implementation follows EIP-1967.



Upgrading

There are two different types of upgrades: the global upgrade and per account upgrade.

At early stages, Hexlink is able to upgrade the default implementation by resetting the implementation contract at the upgradeable beacon contract. In this case, all beacon proxy contracts pointing to the upgradable beacon will be affected. All future contracts(not deployed yet) will also be affected and point to the new implementation. Once the implementation is stabilized, Hexlink will disable the global upgrade feature permanently after which only per account upgrade is allowed.

Each user can also upgrade their contract to a different implementation by pointing the proxy to a customized beacon contract, after which it will use the implementation referenced by the customized beacon contract.

Ownership

Each beacon proxy has an admin which is stored at the admin slot. The admin has the authority to

1. update the beacon.
2. update the admin.
3. execute specific functions of the implementation contract, such as token transfer

The owner of the account contract could be either contract or externally owned account.

Contract Account Admin

We can validate the ownership by two ways:

1. If msg.sender == _getAdmin(), the transaction is valid
2. If msg.sender != _getAdmin(), we should validate the transaction signature with EIP-1271.

Externally Owned Account Admin

We can validate the ownership by two ways:

1. If msg.sender == _getAdmin(), the transaction is valid
2. If msg.sender != _getAdmin(), we should validate if the signature is signed by the admin account.

Trusted Forwarder/Entrypoint contract

Each contract can set up a trusted forwarder or entrypoint contract, which can help to call the account contract functions and manage gas payment. The account contact will blindly trust the calls from the entrypoint contract.

Instead of sending the contract by users themselves, in this case, they will just sign the transaction and send the transaction to Hexlink so Hexlink can aggregate the transactions and send them on-behalf of these users. Hexlink will pay the gas fee first and then be refunded by users.

The signature is signed over an UserOp struct. We will calculate the hash of the UserOp struct following EIP-712 and generate the request id as `keccak256(hashStruct(UserOp), address(entrypoint), block.chainId)`. The signature is signed over the request id following EIP-712 as well.

Multi-Sig Support

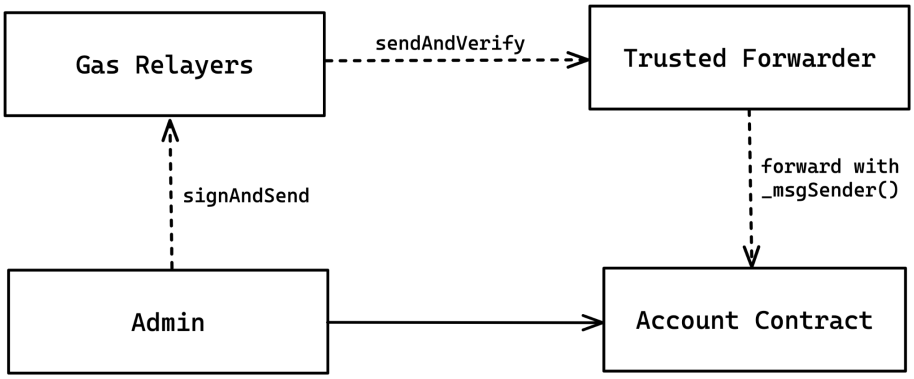
The admin could be a contract with multisig support. Users can authenticate with an admin contract with multisig first after which the admin contract can forward the transaction to the account contract.

Gas Payment Model

Meta Transactions (EIP-2771)

Reference: MinimalForwarder.sol, Gasless Transactions

Meta transactions can be used for gas sponsorship. The account contract will receive any transactions from a trusted forwarder. Users only need to sign transactions and send it to gas relayers. The gas relayer will pay gas for the transaction.



In reality, we can use [OpenZeppelin Defender Relay] as gas relayers for hexlink users and expose http API for the front end. Also, the account contract can serve as the trusted forwarder as long as it can verify signatures. Users can deposit money into their Hexlink accounts and pay gas with dollars directly.

To support native gas payment, the account contract can pay the gas to the gas relayer(Hexlink) automatically after executing the transaction. The design of this is TBD. This could be similar to EIP-4337.

If the admin is a contract, then the user is responsible for sending transactions by themselves. Users can not use hexlink gas relayers in this case.

Account Abstraction (ERC-4337)

Account abstraction can be used for both native gas payment and gas sponsorship. The contract implementation could be an instance of BaseWallet.sol. Hexlink can deposit at the entrypoint and serve as the pay master for hexlink contracts.

Final Interface

State

- admin
- beacon
- entryPoint(for ERC4337)
- nonce
- initialized

Functions

View Functions

- Basic: beacon(), admin()
- EIP4337: validateUserOps(), nonce(), entryPoint()

State Update Functions

- Basic: upgradeBeaconToAndCall(), changeAdmin()
- EIP4337: updateEntryPoint()

Exec Functions

- exec(), execBatch()

Exploration: Plugins

EIP-2535 provides another way to upgrade contracts, with which you can cut/add one function freely while keeping the other functions untouched. It relies on a map from function selector to implementation contract. It can be used to support plugins: each user can add/remove plugins, which is a set of functions, to their contract freely with one transaction to override the function to contract mapping.

Implementation Reference

Gnosis Safe: A smart contract wallet with multi-sig
GnosisSafe.sol, GnosisSafeL2.sol

Account Abstraction: EIP-4337 Compatible wallet implementation
BaseWallets.sol, SimpleWallet.sol

ERC-2535 Mode: The diamond mode where you can add/remove functions from smart contract dynamically, EIP-2535, this is perfect for dynamic plugins.
Proxy.sol

Proxies: EIP-1967, Proxies Explanation
Proxy.sol, BeaconProxy.sol, TransparentUpgradableProxy.sol, UUPSUpgradable.sol

Clone contract: EIP-1167
Clones.sol