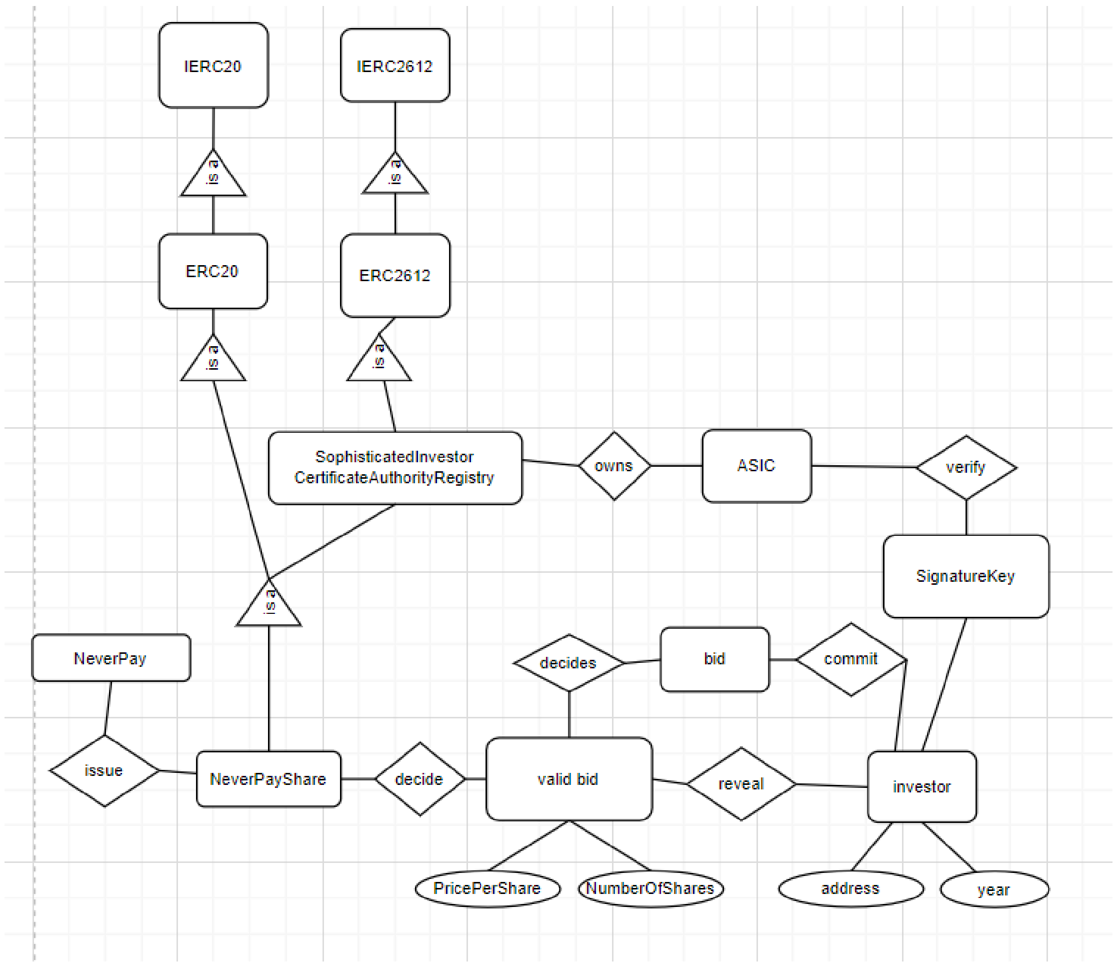
Report

1. Data Model



Because NeverPay is like publishing a Computer Share, we can call it NeverPay Share, which is essentially a token that we can implement using the ERC20 contract standard. And since we need to use off-chain signatures, this can be done using the EIP2612 standard. ASIC need a contract called SophisticatedInvestorCertificateAuthorityRegistry to inherit EIP2612 to achieve an agreement signature to verify sophisticated investors. In the first round, investors are required to provide the public key when committing the bid information and be verified to be a sophisticated investor. In the second round the investor reveals the bid, and if it matches the one submitted in the first round, a valid bid is generated. Finally through the investor's bid to confirm the allocation of shares.

2. Off-chain calculation

Before bidding, the investors should calculate encrypted messages by themselves keccak256(numberOfShares, pricePerShare, fake, secret). The operation is done off-chain.

Creating the signature also involves off-chain calculation. ASIC can do the below operation off-chain and directly introduces hash calculated.

```
PERMIT_TYPEHASH = keccak256("Permit(address owner,address spender,uint8
year,uint256 nonce,uint256 deadline)");
```

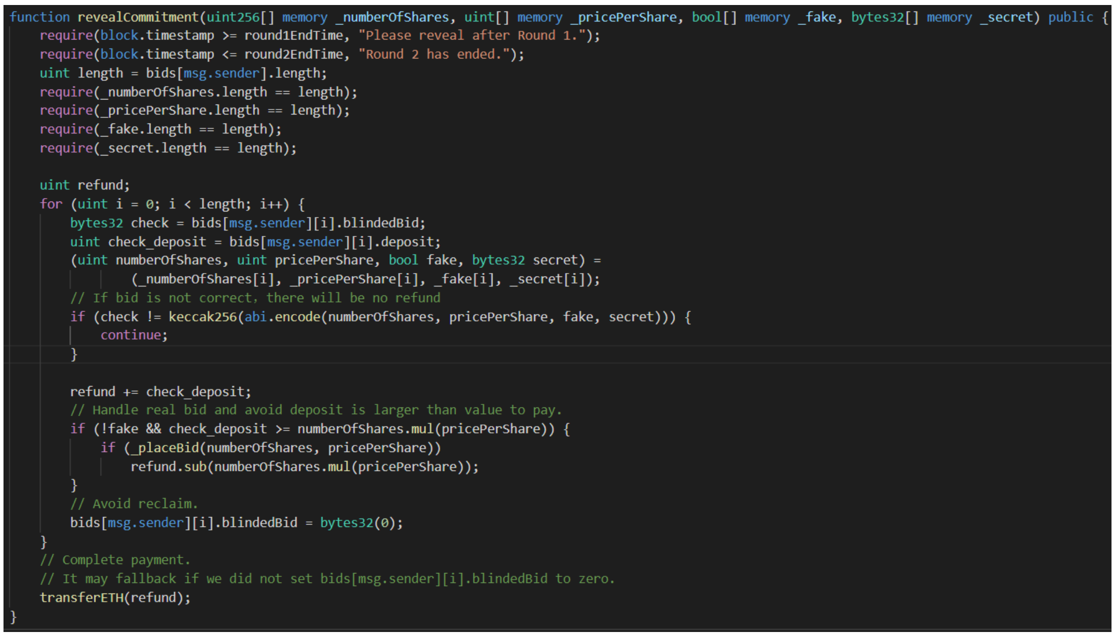
3. Requirements

3.1 A total of 10,000 shares are to be sold.

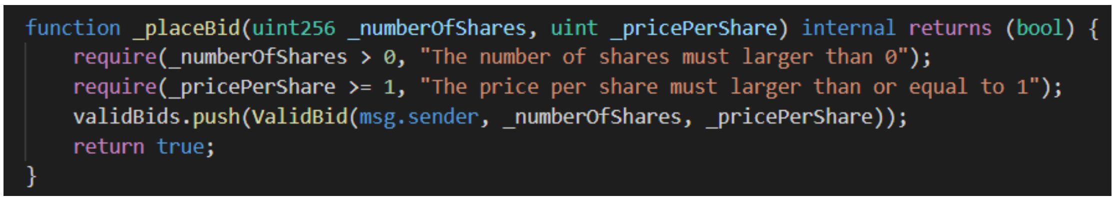
In contract ERC20, we define total_supply = 10000.

3.2 The minimum price per share is 1 Ether, any bid to buy shares for less than this amount will not be issued shares.

During Round 1, since it is blind action, we cannot check whether the price per share is larger than or equal to 1. We check it in round reveal time.



```
function revealCommitment(uint256[] memory _numberOfShares, uint[] memory _pricePerShare, bool[] memory _fake, bytes32[] memory _secret) public {
    require(block.timestamp >= round1EndTime, "Please reveal after Round 1.");
    require(block.timestamp <= round2EndTime, "Round 2 has ended.");
    uint length = bids[msg.sender].length;
    require(_numberOfShares.length == length);
    require(_pricePerShare.length == length);
    require(_fake.length == length);
    require(_secret.length == length);

    uint refund;
    for (uint i = 0; i < length; i++) {
        bytes32 check = bids[msg.sender][i].blindedBid;
        uint check_deposit = bids[msg.sender][i].deposit;
        (uint numberOfShares, uint pricePerShare, bool fake, bytes32 secret) =
                (_numberOfShares[i], _pricePerShare[i], _fake[i], _secret[i]);
        // If bid is not correct, there will be no refund
        if (check != keccak256(abi.encode(numberOfShares, pricePerShare, fake, secret))) {
            continue;
        }

        refund += check_deposit;
        // Handle real bid and avoid deposit is larger than value to pay.
        if (!fake && check_deposit >= numberOfShares.mul(pricePerShare)) {
            if (_placeBid(numberOfShares, pricePerShare))
                refund.sub(numberOfShares.mul(pricePerShare));
        }
        // Avoid reclaim.
        bids[msg.sender][i].blindedBid = bytes32(0);
    }
    // Complete payment.
    // It may fallback if we did not set bids[msg.sender][i].blindedBid to zero.
    transferETH(refund);
}
```

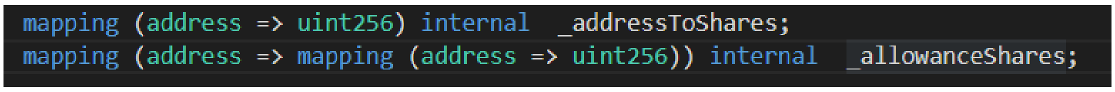


```
function _placeBid(uint256 _numberOfShares, uint _pricePerShare) internal returns (bool) {
    require(_numberOfShares > 0, "The number of shares must larger than 0");
    require(_pricePerShare >= 1, "The price per share must larger than or equal to 1");
    validBids.push(ValidBid(msg.sender, _numberOfShares, _pricePerShare));
    return true;
}
```

After we checked a bid to be a valid bid, we call _placeBid() function to store it in validBids[]. When placing Bid to ValidBid, the require statement check whether the price per share is larger than or equal to 1.

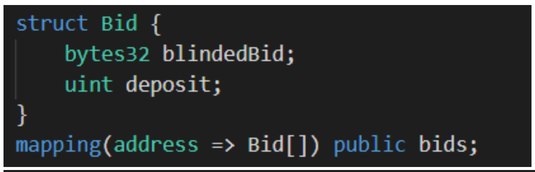
3.3 Investors should be identified by means of an anonymous Ethereum address - the system should not store personal information on the blockchain.

The system does not involve personal information. Only the address is known. And important information are defined to be internal.

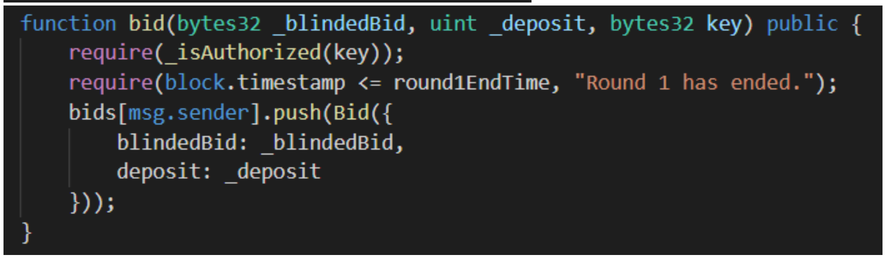


```
mapping (address => uint256) internal  _addressToShares;
mapping (address => mapping (address => uint256)) internal  _allowanceShares;
```

3.4 In order to encourage competition amongst the investors and maximize the amount of money raised, the share sale is to be conducted via a blind auction, consisting of two rounds.



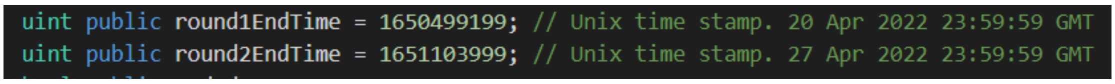
```
struct Bid {
    bytes32 blindedBid;
    uint deposit;
}
mapping(address => Bid[]) public bids;
```



```
function bid(bytes32 _blindedBid, uint _deposit, bytes32 key) public {
    require(_isAuthorized(key));
    require(block.timestamp <= round1EndTime, "Round 1 has ended.");
    bids[msg.sender].push(Bid({
        blindedBid: _blindedBid,
        deposit: _deposit
    }));
}
```

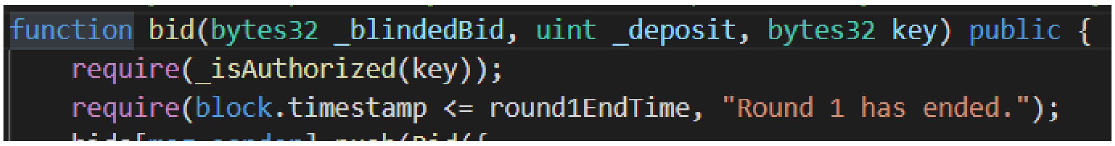
There is a struct Bid is created to store blind bid and a mapping from investor's address to bids. When the investors commit a bid, he or she need to passing 3 parameters. A hash about blinded bid information, a deposit value and a key. The value of blindedBid is the SHA256 hash value of (numberOfShares, pricePerShare, fake, secret) which is calculated off-chain. If the Boolean value of fake is set to true, it identify this is a false bid. If the Boolean value of fake is set to false, it is a true bid. The string secret is message that only the investor knows. The deposit will be returned only if the bid is correct. The key is the public key the investors kept to verify their identity of sophisticated investor.

3.5 Deadlines for the rounds are set as (Round1: April 20, 2022, Round2: April 27, 2022) at the time the smart contract is created. Actions that were supposed to have been taken in a round should be rejected after the round deadline.

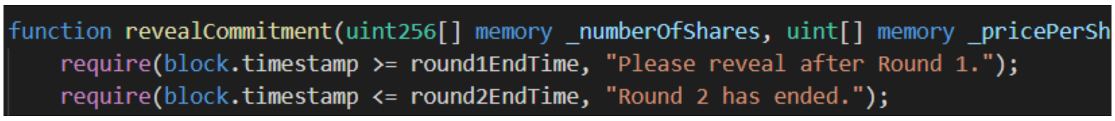


```
uint public round1EndTime = 1650499199; // Unix time stamp. 20 Apr 2022 23:59:59 GMT
uint public round2EndTime = 1651103999; // Unix time stamp. 27 Apr 2022 23:59:59 GMT
```

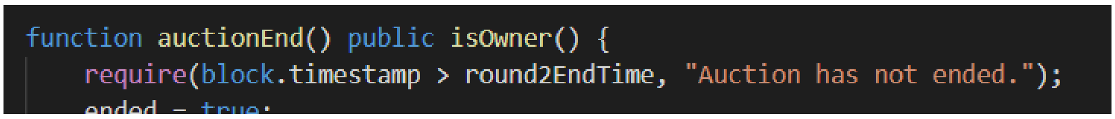
I used these two Unix stamps to represent two deadline. At each related method call we check the time stamp first, whether it is before or after the deadline.



```
function bid(bytes32 _blindedBid, uint _deposit, bytes32 key) public {
    require(_isAuthorized(key));
    require(block.timestamp <= round1EndTime, "Round 1 has ended.");
    bids[msg.sender].push(Bid({
```



```
function revealCommitment(uint256[] memory _numberOfShares, uint[] memory _pricePerSh
    require(block.timestamp >= round1EndTime, "Please reveal after Round 1.");
    require(block.timestamp <= round2EndTime, "Round 2 has ended.");
```

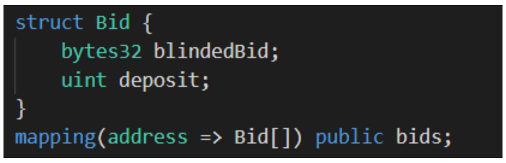


```
function auctionEnd() public isOwner() {
    require(block.timestamp > round2EndTime, "Auction has not ended.");
    ended = true;
```

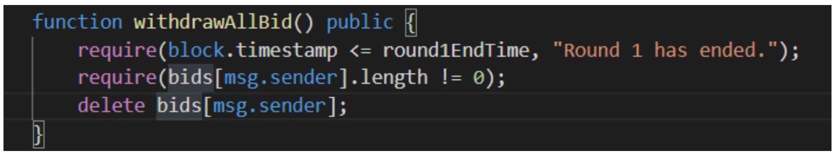
3.6 In the first round, investors wishing to buy shares should submit a blinded commitment of an offer to buy a given number of shares at a given price (in Ether) per share. While the first round is running, it should not be possible for any investor to determine from public information on the blockchain what number of shares any other investor has offered to buy, nor what price they have offered to pay.

As section 3.4 mentioned, the information is in the hash value, it is not possible for any investor to determine from public information on the blockchain what number of shares the investor has offered to buy, nor what price they have offered to pay.

3.7 An investor may submit multiple bids in the first round. Any bid may be withdrawn until the first round deadline.



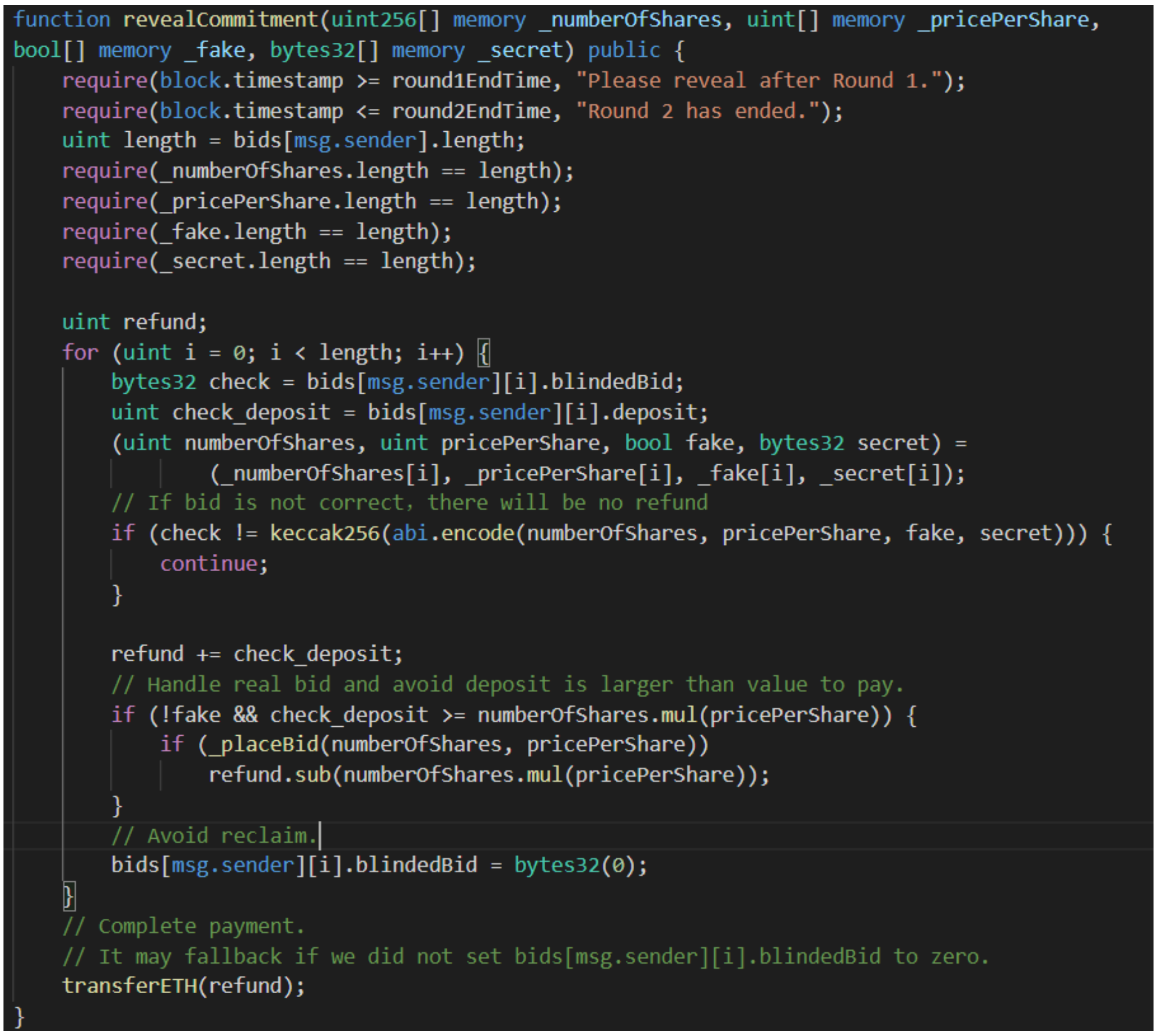
```
struct Bid {
    bytes32 blindedBid;
    uint deposit;
}
mapping(address => Bid[]) public bids;
```



```
function withdrawAllBid() public {
    require(block.timestamp <= round1EndTime, "Round 1 has ended.");
    require(bids[msg.sender].length != 0);
    delete bids[msg.sender];
}
```

A mapping can store multiple bids of one address. The investor can call withdrawAllBid() to withdraw his bids.

3.8 In the second round, the investors are required to open their commitments, revealing the number of shares in the bid and the price offered. It should not be possible for the investor to cheat in this round, revealing information that differs from what they committed to in the first round.



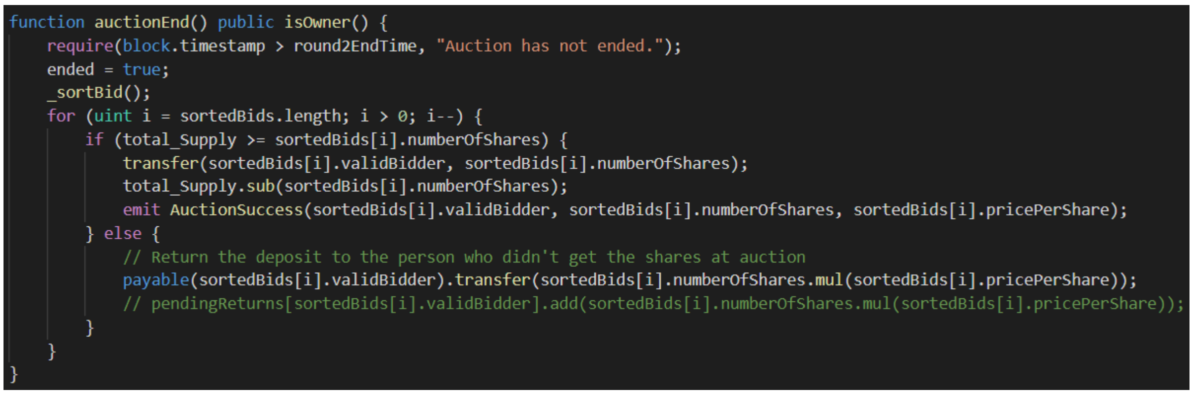
```
function revealCommitment(uint256[] memory _numberOfShares, uint[] memory _pricePerShare,
bool[] memory _fake, bytes32[] memory _secret) public {
    require(block.timestamp >= round1EndTime, "Please reveal after Round 1.");
    require(block.timestamp <= round2EndTime, "Round 2 has ended.");
    uint length = bids[msg.sender].length;
    require(_numberOfShares.length == length);
    require(_pricePerShare.length == length);
    require(_fake.length == length);
    require(_secret.length == length);

    uint refund;
    for (uint i = 0; i < length; i++) {
        bytes32 check = bids[msg.sender][i].blindedBid;
        uint check_deposit = bids[msg.sender][i].deposit;
        (uint numberOfShares, uint pricePerShare, bool fake, bytes32 secret) =
                (_numberOfShares[i], _pricePerShare[i], _fake[i], _secret[i]);
        // If bid is not correct, there will be no refund
        if (check != keccak256(abi.encode(numberOfShares, pricePerShare, fake, secret))) {
            continue;
        }

        refund += check_deposit;
        // Handle real bid and avoid deposit is larger than value to pay.
        if (!fake && check_deposit >= numberOfShares.mul(pricePerShare)) {
            if (_placeBid(numberOfShares, pricePerShare))
                refund.sub(numberOfShares.mul(pricePerShare));
        }
        // Avoid reclaim.
        bids[msg.sender][i].blindedBid = bytes32(0);
    }
    // Complete payment.
    // It may fallback if we did not set bids[msg.sender][i].blindedBid to zero.
    transferETH(refund);
}
```

In Round 2, the investor call this revealCommitment() function to reveal the number of shares in the bid and the price offered. For every bids of this address, it checks whether the hash value of the input matches the blinded Bid information committed in the first round. Then it checks bids with fake value that are false and whether the deposits are larger than current number of shares multiply current price per share. If the bid meets theses requirements, it is added to validBid.

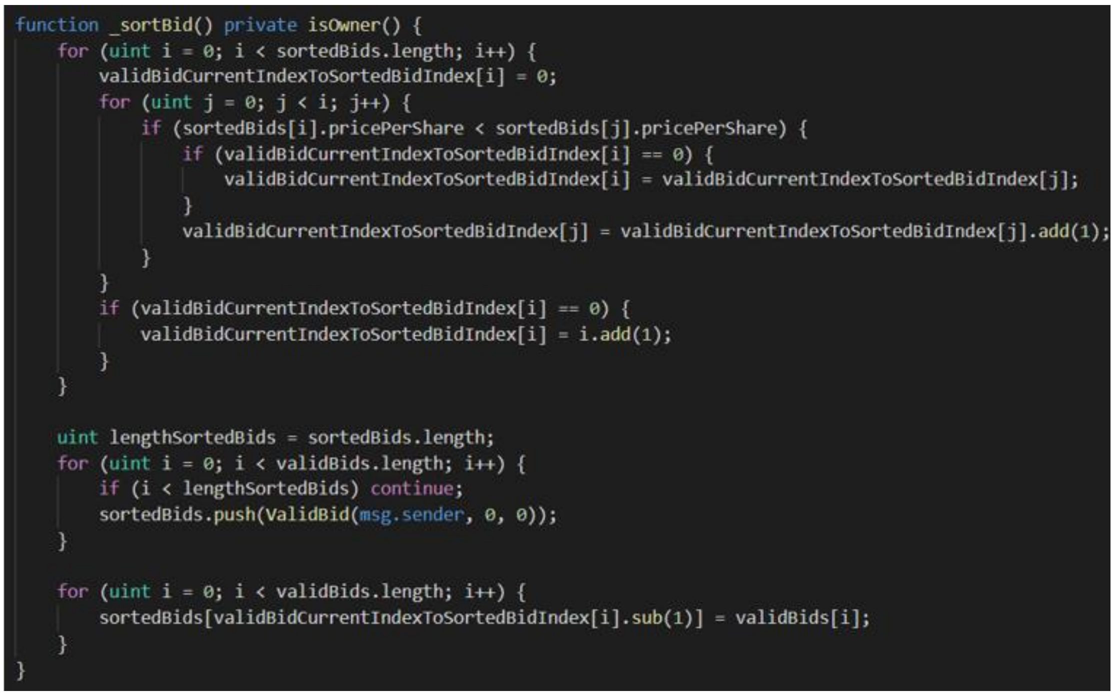
3.9 In this round, the investor is also required to submit, to the smart contract, their payment in full (number of shares × price per share) in Ether for the shares that they are offering to buy.

This was explained in Section 3.8.

3.10 After the deadline for the end of the second round has passed, any commitment that has not been opened in the second round, or which has been opened but for which the correct payment has not been provided, should be treated as an invalid, unsuccessful bid. The remaining bids are treated as valid and potentially successful (but may still be unsuccessful because the offer was oversubscribed, and there is more demand for shares than are being sold). Shares should then be issued to the investors submitting the valid bids with the highest prices. Suppose the valid bids are (i1, s1, p1), . . . ,(in, sn, pn), where each ii is an investor, each si is a number of shares, and each pi is a price, with p1 ≥ p2 ≥ pn. Then the 10,000 shares should be issued to the investors so that i1 gets s1 shares, i2 gets s2 shares, and so on, until all 10,000 shares have been issued. The final investor ik receiving shares may receive only some of the shares in their bid. If they receive s shares, then they pay s ∗ pk for these shares, and they should have the remainder (sk−s)*pk of their payment refunded. The bid of every investor ik+1 . . . in after this final successful investor ik is treated as an unsuccessful bid.



```
function auctionEnd() public isOwner() {
    require(block.timestamp > round2EndTime, "Auction has not ended.");
    ended = true;
    _sortBid();
    for (uint i = sortedBids.length; i > 0; i--) {
        if (total_Supply >= sortedBids[i].numberOfShares) {
            transfer(sortedBids[i].validBidder, sortedBids[i].numberOfShares);
            total_Supply.sub(sortedBids[i].numberOfShares);
            emit AuctionSuccess(sortedBids[i].validBidder, sortedBids[i].numberOfShares, sortedBids[i].pricePerShare);
        } else {
            // Return the deposit to the person who didn't get the shares at auction
            payable(sortedBids[i].validBidder).transfer(sortedBids[i].numberOfShares.mul(sortedBids[i].pricePerShare));
            // pendingReturns[sortedBids[i].validBidder].add(sortedBids[i].numberOfShares.mul(sortedBids[i].pricePerShare));
        }
    }
}
```



```
function _sortBid() private isOwner() {
    for (uint i = 0; i < sortedBids.length; i++) {
        validBidCurrentIndexToSortedBidIndex[i] = 0;
        for (uint j = 0; j < i; j++) {
            if (sortedBids[i].pricePerShare < sortedBids[j].pricePerShare) {
                if (validBidCurrentIndexToSortedBidIndex[i] == 0) {
                    validBidCurrentIndexToSortedBidIndex[i] = validBidCurrentIndexToSortedBidIndex[j];
                }
                validBidCurrentIndexToSortedBidIndex[j] = validBidCurrentIndexToSortedBidIndex[j].add(1);
            }
        }
        if (validBidCurrentIndexToSortedBidIndex[i] == 0) {
            validBidCurrentIndexToSortedBidIndex[i] = i.add(1);
        }
    }

    uint lengthSortedBids = sortedBids.length;
    for (uint i = 0; i < validBids.length; i++) {
        if (i < lengthSortedBids) continue;
        sortedBids.push(ValidBid(msg.sender, 0, 0));
    }

    for (uint i = 0; i < validBids.length; i++) {
        sortedBids[validBidCurrentIndexToSortedBidIndex[i].sub(1)] = validBids[i];
    }
}
```

After auction is end, it is required for NeverPay to call actionEnd() function. The _sortBid() function will sort valid bids from small to large and store them into sortedBid[]. Traversing sortedBid[] from back to front, if current totol_supply value is larger than sortedBid[i].numberOfShares. Transfer shares to this bidder. Decrement total_supply value. If current total_supply value is not enough, refund deposit to the bidder. That's not what the requirement asking for since I haven't find the proper solution.

3.11 Any unsuccessful bids should have payments that have been made to the smart contract refunded to the bidder.

This has been stated in Section 3.10.

3.12 After shares have been issued, it should be possible for investors to transfer their shares to another investor, using ERC-20 standard operations.

I implemented a ERC20 contract which inherited the interface IERC20 and override its functions. The NeverPay Shares is just like a kind of Token using ERC-20 standard. Investor can make transaction with others by calling transfer(), transferFrom(), allowance() and approve() functions.

3.13 The smart contract should be cost effective for the NeverPay to run. To the largest extent possible, transaction costs should be borne by the investors rather than by NeverPay.

NeverPay doesn't need to spend more gas, except for deployment and calling functions at the end of the auction.

4. Running costs

NeverPay:

Deploy ERC20: 967760 units of gas

Deploy contract NeverPay: 3450440 units of gas

Investor:

Transfer NeverPay shares: 21,000 units of gas

Gas costs for other operations can be calculated by referring to the Ethereum Yellow Book and it can also be calculated by using ‘msg.gas’ in code to return the current remaining gas, but each call costs an additional 2 gas.

5. Overall suitability of the Ethereum platform for this application

Token is a form of fund-raising. Ethereum can realize security, reliability, auditing and distribution of shares. With good flexibility and liquidity, the source of funds is global and fast, and investors are given timely exit opportunities through the secondary market. For the financiers, there is no equity or debt obligation beyond the promised return. A lot of investors don't want to control the company, they just want to make a financial return, and by raising money in tokens, NeverPay can skip the equity stakes and avoid the dilution of decision-making power that comes with raising money.

Shares are issued in token form, which is hard to get official recognition because they are unofficially issued. Countries like China have banned ICOs. Many governments do not allow or support token fundraising. A token is just a "capital contribution certificate", proving that you have contributed to the company or organization, but it is not an equity or debt in the current sense. Token is still blank in legal nature at present. While blockchain technology can guarantee security and openness, it cannot guarantee privacy. In addition, the high gas fee leads to high transaction costs.

6. Avoid of common Solidity security vulnerabilities like reentrancy attacks

According to the EIP-712 definition, DOMAIN_SEPARATOR should be the only DOMAIN_SEPARATOR for the contract and chain to prevent reentrancy attacks from other domains.

A common DOMAIN_SEPARATOR is like:

```
DOMAIN_SEPARATOR = keccak256(
    abi.encode(
        keccak256('EIP712Domain(string name,string version,uint256 chainId,address verifyingContract)'),
        keccak256(bytes(name)),
        keccak256(bytes(version)),
        chainid,
        address(this)
));
```

Variable nonces records the number of signatures used for reentrancy protection.

In function revealCommitment(), after couting the refund, we need to set bids[msg.sender][i].blindedBid = bytes32(0) to avoid reentrancy attacks.

7. Security considerations

ecrecover () precompilation fails silently, and signer returns only zero addresses for ill-formed messages. Therefore, when calling the permit method, be sure to confirm owner is not address(0) to avoid approval to spend zero address burn money.

If permitStake() is submitted repeatedly, the previous public key can be invalidated.

If DOMAIN_SEPARATOR contains a chainId defined at contract deployment, rather than being rebuilt for each signature, there may be a risk of reentrancy attacks in the event of a future chain split.

8. Acknowledgement

SafeMath Library

Using SafeMath library can prevent from overflows. Arithmetic operations in Solidity wrap on overflow. This can easily result in bugs, because programmers usually assume that an overflow raises an error, which is the standard behavior in high level programming languages. SafeMath restores this intuition by reverting the transaction when an operation overflows.[1]

ERC20 Standard

"ERC20" refers to a scripting standard used within the Ethereum blockchain. This technical standard dictates a number of rules and actions that an Ethereum token or smart contract must follow and steps to be able to implement it. It is perhaps easiest to think of ERC20 as a set of basic guidelines and functions that any new token created in the Ethereum network must follow.[2]

EIP2612 Standard

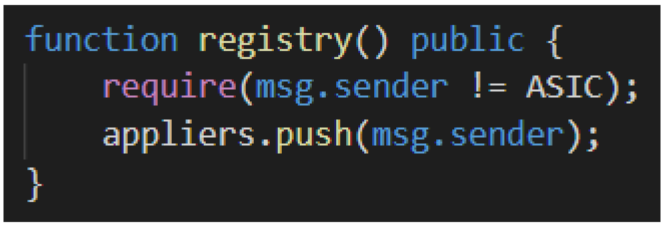
EIP2612 realized both off-chain signature and gasless approval of tokens. It has a permit() function to extend ERC20. It uses EIP-712's domain hash to structure the signature and ensures a signature is only used for our given token contract address on the correct chain id.

TokenRecipient Interface

It represents a contract which you could sent tokens to.

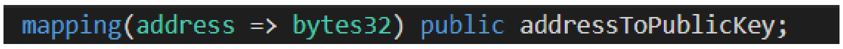
9. Stretch goal

In contract SophisticatedInvestorCertificateAuthorityRegistry inherited from ERC2612, I defined a function for investors to register to be sophisticated investor candidates. Record them in a list.

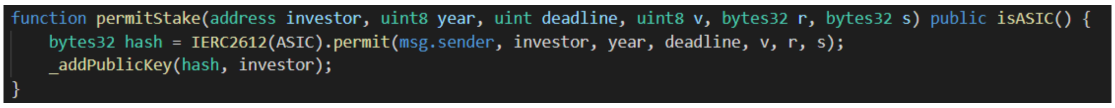


```
function registry() public {
    require(msg.sender != ASIC);
    appliers.push(msg.sender);
}
```

ASIC can check the list and call the permitStake() function to permit a investor to be a sophisticated investor.

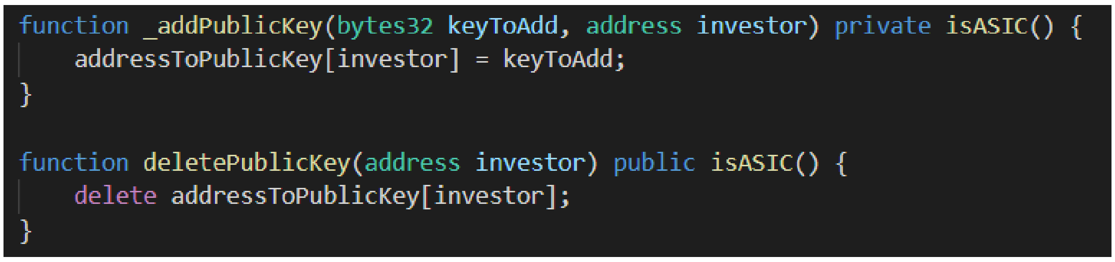


```
mapping(address => bytes32) public addressToPublicKey;
```

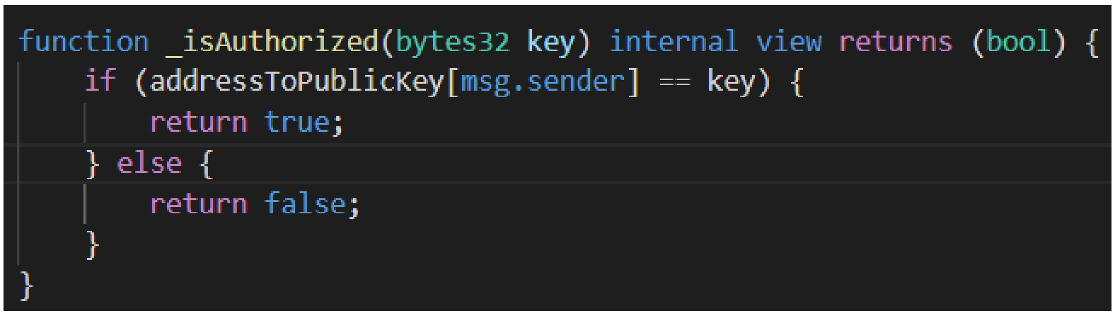


```
function permitStake(address investor, uint8 year, uint deadline, uint8 v, bytes32 r, bytes32 s) public isASIC() {
    bytes32 hash = IERC2612(ASIC).permit(msg.sender, investor, year, deadline, v, r, s);
    _addPublicKey(hash, investor);
}
```

Let the generated hash to be public key and store the investor and the key in a mapping. ASIC can add or delete the public keys. When an investor commit a bid, function _isAuthorized() is called to check whether the key is valid. In bid() function it will check whether the key is belong to this investor.



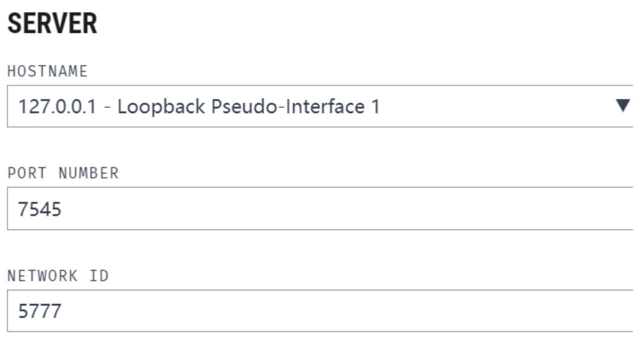
```
function _addPublicKey(bytes32 keyToAdd, address investor) private isASIC() {
    addressToPublicKey[investor] = keyToAdd;
}

function deletePublicKey(address investor) public isASIC() {
    delete addressToPublicKey[investor];
}
```



```
function _isAuthorized(bytes32 key) internal view returns (bool) {
    if (addressToPublicKey[msg.sender] == key) {
        return true;
    } else {
        return false;
    }
}
```

Testing using Truffle and Ganache

Before deploying, I should modify truffle-config.js file according to the server information in Ganache. Then I should create a file named 1_deploy_NeverPay.js for deploying. Then I write a test script. At last, I used ‘truffle migrate’ and ‘truffle test’ to deploy and test my contracts. Unfortunately, I haven’t complete my test file since I am not familiar with web3 and JavaScript.

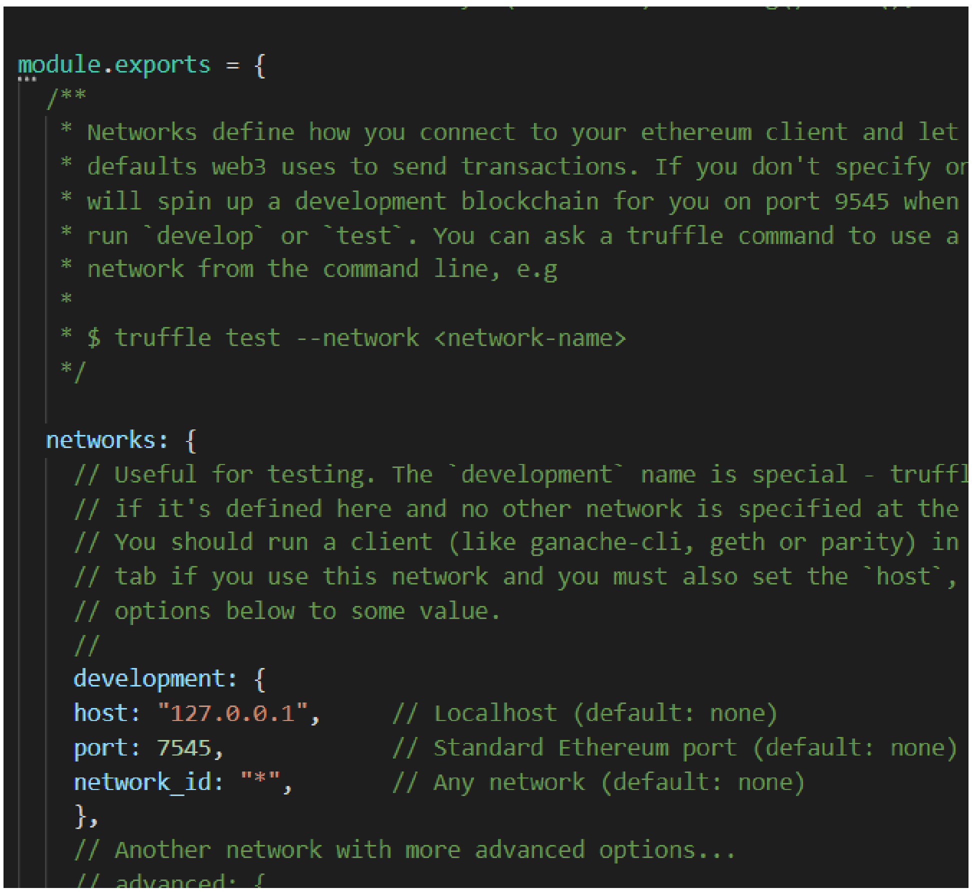


SERVER

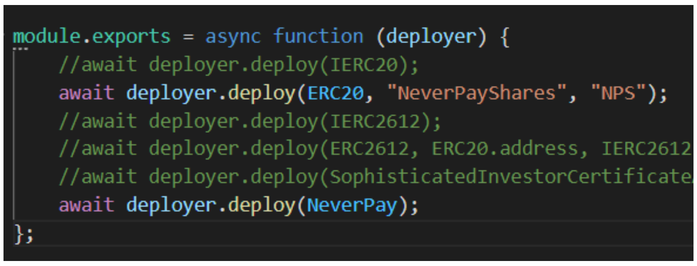
HOSTNAME
127.0.0.1 - Loopback Pseudo-Interface 1

PORT NUMBER
7545

NETWORK ID
5777



```
module.exports = {
  /**
   * Networks define how you connect to your ethereum client and let
   * defaults web3 uses to send transactions. If you don't specify or
   * will spin up a development blockchain for you on port 9545 when
   * run `develop` or `test`. You can ask a truffle command to use a
   * network from the command line, e.g
   *
   * $ truffle test --network <network-name>
   */

  networks: {
    // Useful for testing. The `development` name is special - truffl
    // if it's defined here and no other network is specified at the
    // You should run a client (like ganache-cli, geth or parity) in
    // tab if you use this network and you must also set the `host`,
    // options below to some value.
    //
    development: {
    host: "127.0.0.1",     // Localhost (default: none)
    port: 7545,            // Standard Ethereum port (default: none)
    network_id: "*",       // Any network (default: none)
    },
    // Another network with more advanced options...
    // advanced: {
```



```
module.exports = async function (deployer) {
    //await deployer.deploy(IERC20);
    await deployer.deploy(ERC20, "NeverPayShares", "NPS");
    //await deployer.deploy(IERC2612);
    //await deployer.deploy(ERC2612, ERC20.address, IERC2612
    //await deployer.deploy(SophisticatedInvestorCertificate
    await deployer.deploy(NeverPay);
};
```

References

Wiesner, T. (2022, March 28). *Add the SafeMath library safeguard mathematical operations*¶. Add SafeMath Library - Become Ethereum Blockchain Developer. Retrieved April 12, 2022, from https://ethereum-blockchain-developer.com/040-shared-wallet-project/09-add-safemath/

Reiff, N. (2022, February 8). What crypto users need to know: The erc20 standard. Investopedia. Retrieved April 12, 2022, from https://www.investopedia.com/tech/why-crypto-users-need-know-about-erc20-token-standard/