Table of Contents

Usage Document

Intro

Elight is the project designed to automate thermo logistics processes.

It includes the possibility of interaction between temperature data logger device, mobile app and blockchain, where Smart Contracts implementation is the key feature.

To get started download the application Expo for Android or iOS. Then go to the ElightMobile web site and scan the QR-code with your mobile Expo app.

Please follow this link to get animated functionality performance.

Authors

Sergey Ankarenko (@ankarenkosergey#3570), Leonid Markizov (@Leonid#3318).

Content

Idea

The market

A cold chain is a temperature-controlled supply chain. It poses an uninterrupted series of refrigerated production, storage and distribution activities, along with associated equipment and logistics, which maintain the desired low-temperature range. It is used to preserve and to extend and ensure the shelf life of temperature-sensitive products. Such products, during transport and when in transient storage, are sometimes called **cool cargo**.

One of the most common and valuable types of cool cargo are **vaccines and biologic medical products**. In these industries, the conventional temperature range for a cold chain is **2 to 8 ºC**, although the specific temperature tolerances may depend on the actual product being shipped. The temperature range is a part of the cold chain distribution process, which poses an extension of the **good manufacturing practice (GMP)** environment that all drugs and biological products are required to adhere to, enforced by the various health regulatory bodies and governments.

The problem of delivery management for this kind of cool cargo has two dimensions:

1) **Economical dimension.** The valuable cargo has the high risk to be contaminated due to temperature variations which may lead to financial losses.

2) **Social aspect.** Disruption of a cold chain may produce consequences such as decease outbreaks, as the vaccines were inactivated due to excess exposure to heat. Patients that think they were being immunized, in reality, are put at greater risk due to the inactivated vaccines they received. In another case, overheated (or overcooled) vaccines may be dangerous due to their toxicity. Anyway, according to World Health Organization reports, up to 25% vaccines come to patients contaminated due to failures in cold chains.

The problem

We have observed the business practice at this market and find out two points we would like to improve.

Problem #1: ***imperfections of temperature monitoring equipment***

To keep a watch on a temperature of a cool cargo, transport operators use special devices called **temperature data loggers (TDL)**. It is a portable measurement instrument that is capable of autonomously recording temperature over a defined period. Two types of TDLs are being widely used, and both have major disadvantages:

	+	**-**
Vehicle body built-in TDL	Ability for real-time monitoring; Monitoring over the whole batch	Low precision of measurement; Tied to one vehicle
TDL placed in a cool cargo package	High precision of measurement; Follows cargo throughout the entire cold chain	One cannot read recordings until the cargo was unpacked

Problem #2: *mistrust and imperfections of modern business practice*

As our survey taken on Russian market shows, distrust of transport operators is widely spread among pharmaceutical companies, because most of the SME-companies cannot take financial responsibility for the cases of damage to costly cargo called by temperature. Insurance policies **do not usually cover temperature risks** (for instance, see Institute Cargo Clauses by The International Underwriting Association of London) ссылка Therefore, for SME-companies the way to the market of logistics of vaccines or other cool cargos is generally blocked. Pharmaceutical companies prefer to contract with big transport companies or invest in their own truck fleet and equipment.

Furthermore, mistrust between contractors usually takes place in business especially in cases of their farness or lack of successful experience of collaboration.

To solve these problems, we come up with our **solution**.

Solution

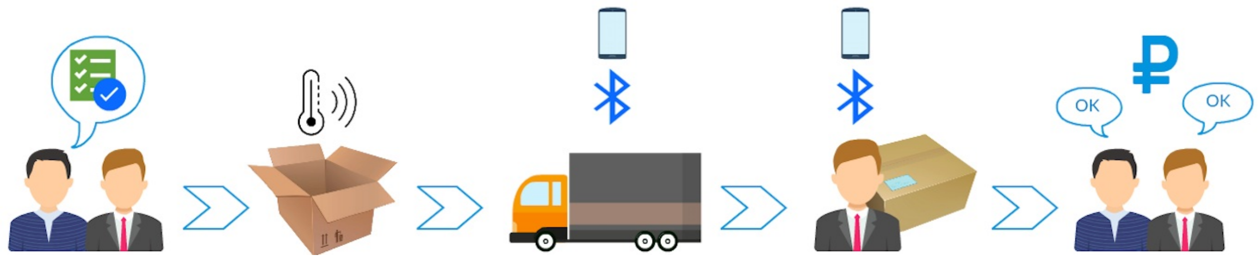
Elight system

Elight system consists of:

1) **Temperature data logger** with Bluetooth module;

2) Mobile platform for detecting data and creation **Smart Contracts**.

How does it work?



For example, there is a deal between three contractors, the Consignor, the Transport operator and the Consignee.

Step 1. They create a Smart contract and program all the key terms of treaty including **temperature range** and **payment conditions**.

Step 2. Transport operator **places money to an account of the Smart contract** as a security deposit to guarantee discharge of obligations of transportation and provision temperature range.

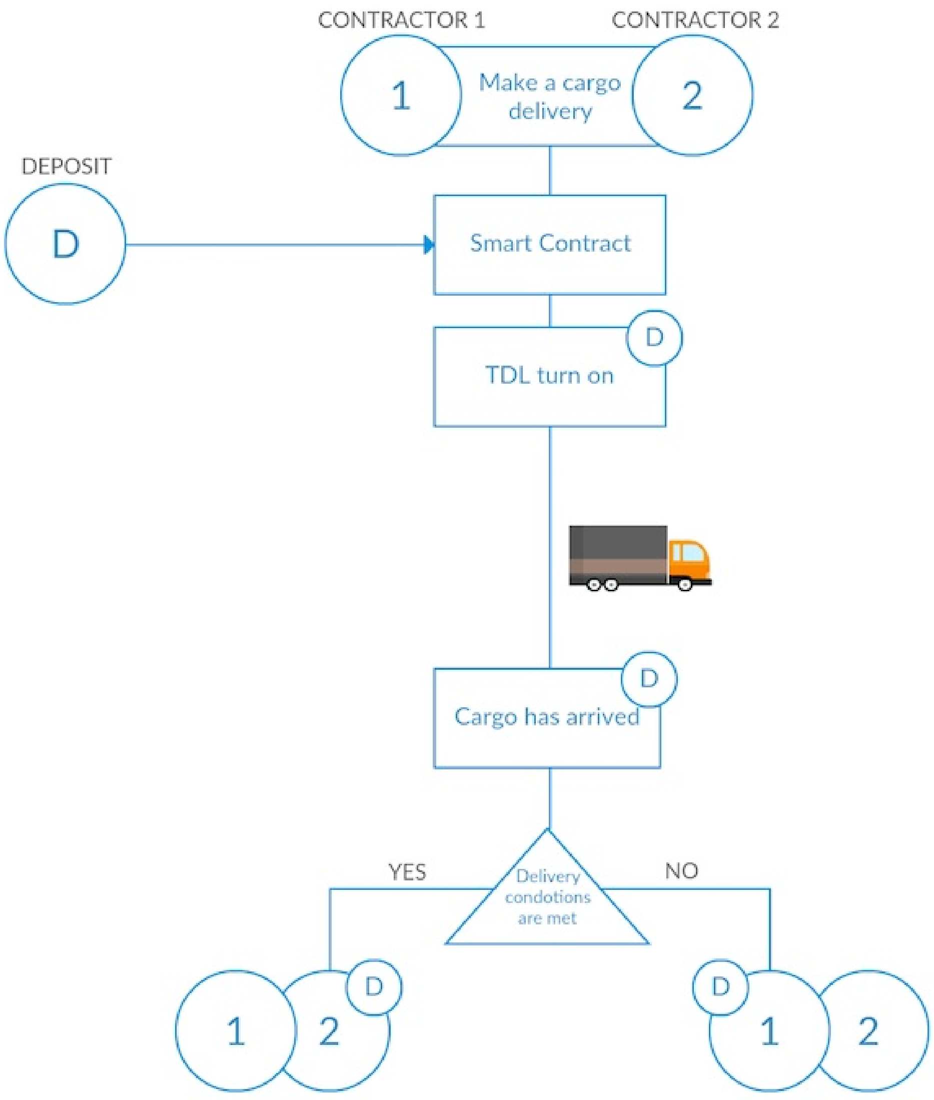
Step 3. The Consignor places TDLs in the boxes with goods and loads a truck.

Step 4. While the truck is running, TDLs measure and record the temperature of goods. A truck driver may **connect to TDLs via his mobile phone (by Bluetooth)** and carry out the real-time monitoring.

Step 5. The truck arrives at the Consignee's depot. A warehouse agent connects to the TDLs via his mobile phone and downloads the temperature logs. These **logs are automatically uploaded to the blockchain.**

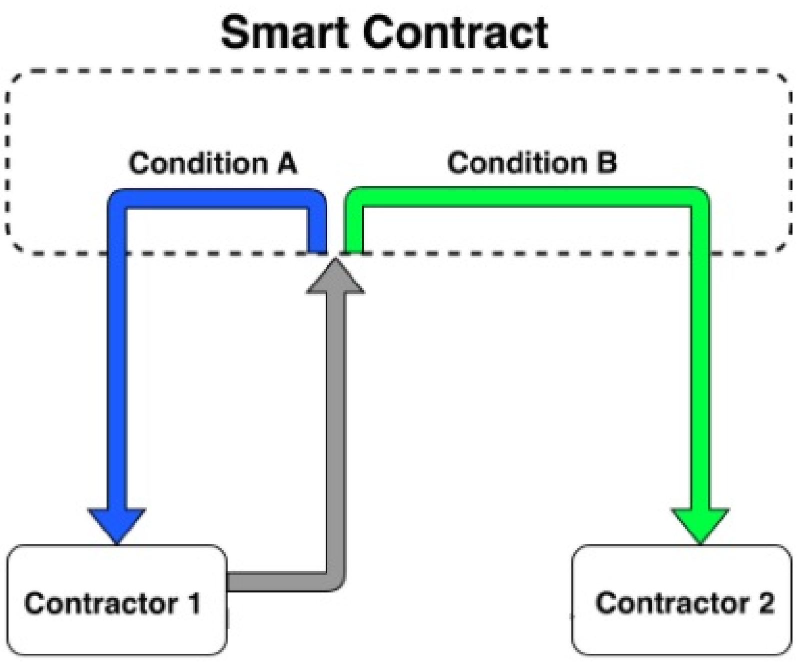
Step 6. The Smart contract checks the uploaded data and reacts in accordance with its program. If there are no temperature fluctuations or other breaches made by Transport operator, the security deposit goes back to him. Otherwise, the security deposit goes to

Consignor as a loss compensation.



This was just one case of how Elight system can be used. The structure of Blockchain-based Smart Contract is agile and provides ability to involve any number of contractors and to set many alternative terms of payment between them. Anyway, the idea of our Smart Contract is to set strong cause-and-effect relationship between the fact of successful delivery (without temperature fluctuations) and the payment.

The general scheme of how the Smart Contract works is represented on the picture below:

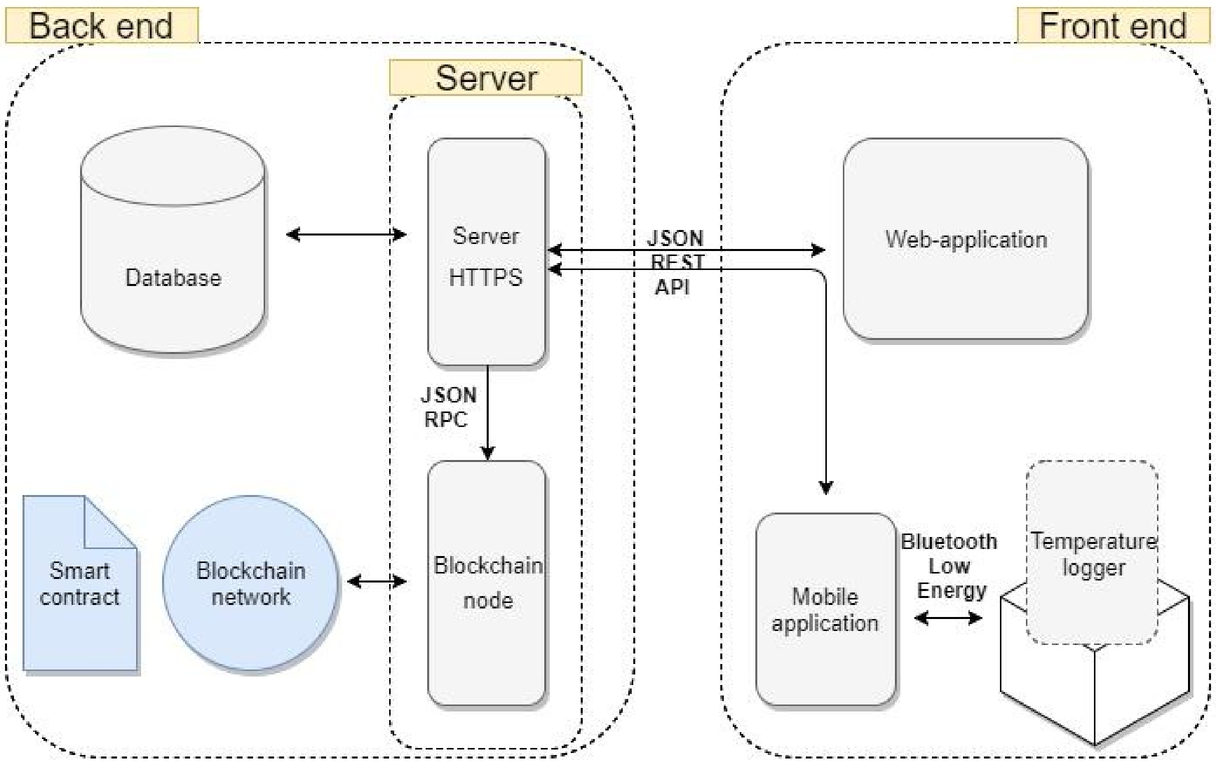


Advantages of Elight system

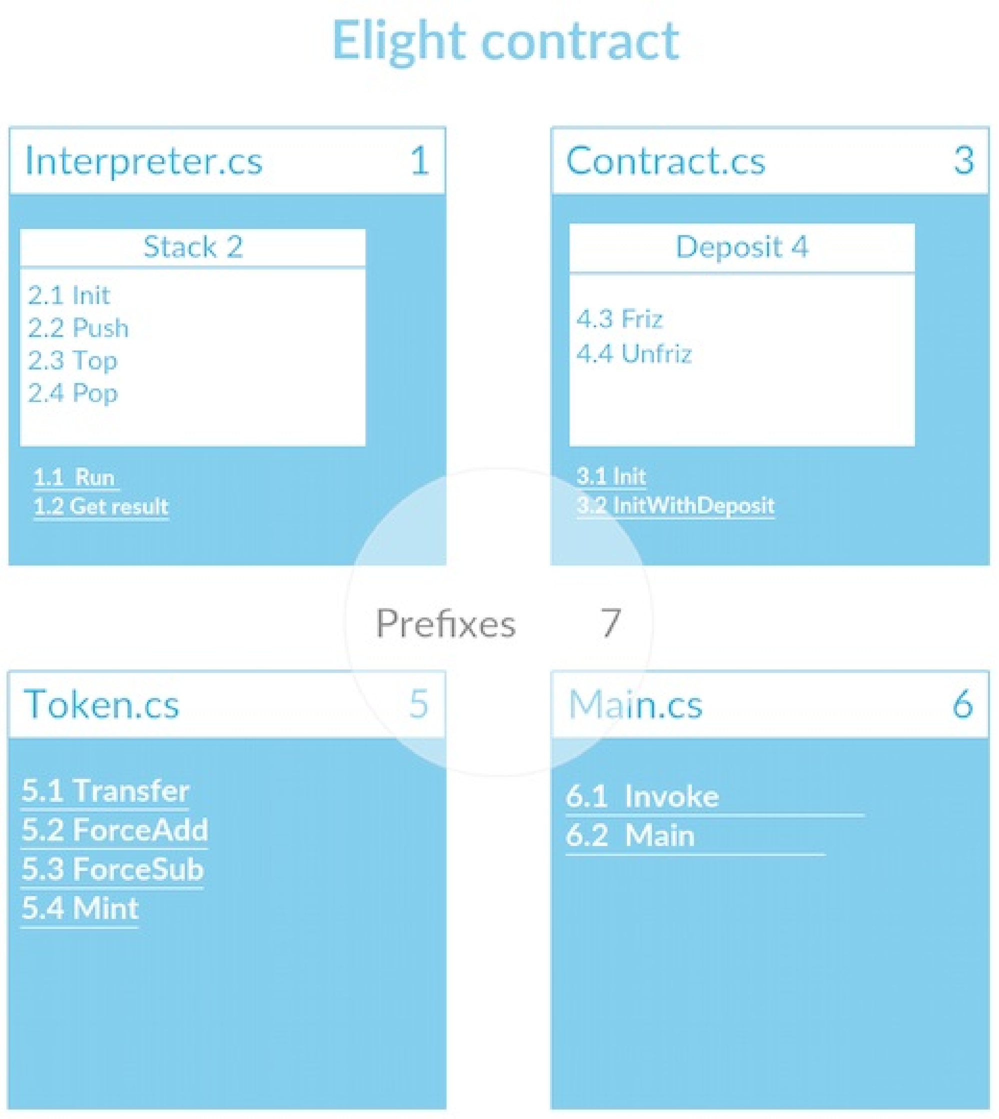
- Real-time temperature monitoring without unpacking the cargo;
- Reducing labor costs and time losses;
- Agile structure of the Smart Contract provides ability to involve any number of contractors and to set many alternative terms of payment between them;
- Ability to establish complicated supply chains;
- Provides additional guarantee to all contractors. May replace bank letter of credit;
- Easy to implement and integrate. No any additional equipment. Only one contractor bears the cost of implementation (for buying TDLs).

Realization

This is the general scheme of Elight platform work:



When developing the smart contract, we wanted to use all the advantages of C# language, so we've split our Smart Contract into small pieces/components. It allowed us to build a scalable and complex application, which is easy to understand, test and extend. There are a diagram and detailed explanation of all the components used in our smart contract.



Main.cs

The starting point of our contract. It has a method for invoking contracts:

```
public static bool Invoke(string carrierHash, BigInteger i, byte[] arg)
```

It invokes early created contracts. You should specify the index of contract, the hash script of carrier and arguments represented as the byte array.

Interpreter.cs

It's very expensive to create a smart NEO contract (about 490 gas) for each cargo delivery (as it can be done in Ethereum, for example). So we decided to write byte code programs which are interpreted by smart contract. So Interpreter.cs file represents interpreter for executing a very simple set of programs. It's possible to implement more complex one. For example, we could implement the interpreter for stack language like forth to enhance contract's functionality.

Interpreter class has two methods...

It accepts contract object with bytecode program in it and executes it with specified parameters (args variable):

```
//note: you can write another byte code program in args
public static Interpreter Run(Interpreter interpreter, Contract contract, byte[] arg)
```

After execution is completed you can get an output by calling `GetResult` method. Its signature looks like this:

```
public static Int32 GetResult(Interpreter Interpreter)
```

It gives you the last element from stack.

Stack.cs

Interpreter uses stack data structure for executing programs. It's a very famous data structure, so we don't want to dive into it.

Stack class operates with 4 byte signed numbers `(Int32)`

```
public static Stack Pop(Stack stack)           //remove element from stack
public static Stack Push(Stack stack, Int32 v) //push element onto stack
public static Int32 Top(Stack stack)           //get copy of element that lays onto s
tack
```

Contract.cs

It represents a contract between contractors. It consists of information in JSON format and a program that written in byte code. Interpreter understands this code and it's able to execute it.

So there're two main methods:

```
public static Contract Init(byte[] info, byte[] source)
```

`source` is a byte code, available at the moment opcodes are described below

```
public enum OPCODES
{
    NEG  = 0x7FFFFFFF, // MULTIPLY BY -1
    SUM  = 0x7FFFFFFE, // a, b => a + b
    SUB  = 0x7FFFFFFD, // a, b => a - b
    MUL  = 0x7FFFFFFC, // a, b => a * b
    ACC  = 0x7FFFFFFB, // a => acc_register + a
    CMP  = 0x7FFFFFFA, // a, b, c => b < a < c
};
```

`info` represents an additional contract data, usually this data is represented in JSON format like so ...

```
{
    "Name": "...",
    "Description": "...",
    "From": "...",
    "To": "...",
    "Goods": ""
}
```

Also it's possible to add deposit to the contract. It brings guarantee and security to contractors.

```
public static Contract InitDeposit(Contract contract, byte[] carrierHash, byte[] clien
tHash, BigInteger amount)
```

It has the same signature as `Init` method except two additional parameters: hash of client and amount to freeze as a deposit.

Deposit.cs

It represents guarantee between contractors. It stores hash their hash addresses and amounts of deposits.

When Contract.cs is initialized, the deposit is frozen.

When some particular conditions are met. The sensor is arrived with cargo and submits all the measurements to the blockchain, tokens are releasing in favor of client or carrier. It depends on the measurements and conditions, that was written in the contract before sending.

```
public static bool Freeze(Deposit deposit)

public static bool Unfreeze(Deposit deposit, bool isOk) //isOk determines reciever
```

Token.cs

It represents our currency which is used mainly for making deposits and adding them to contracts.

It has a couple of useful methods...

For transferring Elight Coin (EC) between contractors...

```
public static bool Transfer(byte[] from, byte[] to, BigInteger value) //'from' and 'to
' are script hashes
```

For exchanging EC on NEO tokens...

```
public static bool MintTokens()
```

Also there are some helper methods. They help us to add and remove specific amount of tokens from address without proving an identity. It mainly used for freezing and unfreezing deposits.

```
ForceAdd(byte[] from, BigInteger amount)
ForceSub(byte[] from, BigInteger amount)
```

Prefixes.cs

All the keys, that store any data in blockchain are computed like so...

```
private static string GetContractKey(string carrierHash, BigInteger index)
{
    string main = Prefixes.CONTRACT_PREFIX + carrierHash;
    return main + index;
}
```

Every key has its own prefix, so it's important to store all the prefixes in one place, otherwise it may lead to Runtime errors which are very hard to find and debug.

For these purposes we created `Prefixes.cs` file. It stores all the prefixes.

Smart Contract communication

In this section we'll give you some instructions on how to communicate with the Smart Contract.

Neo-python cli

The Elight contract's hash is `7ddf3c47eb6ca27e472067c110f4f435478b5fff`

As we discussed earlier, there are two scenarios when creating a contract. With and without the deposit. Will cover both of them.

I. Without depositing

First of all, we importing our smart contract like so:

```
neo> import contract contract.avm 0705 05 True True
```

The next step is to init the contract:

```
neo> testinvoke {script_hash} init ["AK2nJJpJr6o664CWJKi1QRXjqeic2zRp8y","1<x<26",b'00
0000010000001a7ffffffa']
```

Then we emulate sensor's behavior by typing:

```
neo> testinvoke script_hash invoke ["AK2nJJpJr6o664CWJKi1QRXjqeic2zRp8y",1,b'0000000a'
] //1 < 10 < 26 true
neo> testinvoke script_hash invoke ["AK2nJJpJr6o664CWJKi1QRXjqeic2zRp8y",1,b'0000001a'
] //1 < 26 < 26 false
```

II. With depositing

Very similar to previous scenario. First of all, we should mint some tokens, to make a deposit:

```
neo> testinvoke script_hash mint [] --attach-neo=1  //mint 10 EC
```

You'll see something like this:

```
[SmartContract.Storage.Get] AK2nJJpJr6o664CWJKi1QRXjqeic2zRp8y -> 0
[SmartContract.Storage.Put] AK2nJJpJr6o664CWJKi1QRXjqeic2zRp8y -> 10
```

Then we'll be able to initialize a contract with depositing option like so:

```
testinvoke script_hash initDeposit ["AK2nJJpJr6o664CWJKi1QRXjqeic2zRp8y","1<x<26",b'00
0000010000000a7ffffffa',"AKDVzYGLczmykdtRaejgvWeZrvdkVEvQ1X",1]
```

Afterwards, you'll see something like this:

```
[SmartContract.Storage.Get] AK2nJJpJr6o664CWJKi1QRXjqeic2zRp8y -> 10  //freezing token
s
[SmartContract.Storage.Put] AK2nJJpJr6o664CWJKi1QRXjqeic2zRp8y -> 9
```

Then we invoke a contract, pretending that we are hardware sensor:

```
neo> testinvoke script_hash invoke ["AK2nJJpJr6o664CWJKi1QRXjqeic2zRp8y",1,b'0000000a'
] //1 < 10 < 26 true
```

In this case, carrier gets back his deposit, because he's transferred the cargo successfully. You'll immediately see:

```
[SmartContract.Storage.Get] AK2nJJpJr6o664CWJKi1QRXjqeic2zRp8y -> 9  //unfreezing toke
ns in favor of carrier
[SmartContract.Storage.Put] AK2nJJpJr6o664CWJKi1QRXjqeic2zRp8y -> 10
```

Let's look at another scenario:

```
neo> testinvoke script_hash invoke ["AK2nJJpJr6o664CWJKi1QRXjqeic2zRp8y",1,b'0000001a'
] //1 < 26 < 26 false
```

In this case, carrier has failed his mission, so his tokens are sent to client automatically. You'll se something like this:

```
[SmartContract.Storage.Get] AKDVzYGLczmykdtRaejgvWeZrvdkVEvQ1X -> 0  //unfreezing toke
ns in favor of carrier
[SmartContract.Storage.Put] AKDVzYGLczmykdtRaejgvWeZrvdkVEvQ1X -> 1
```

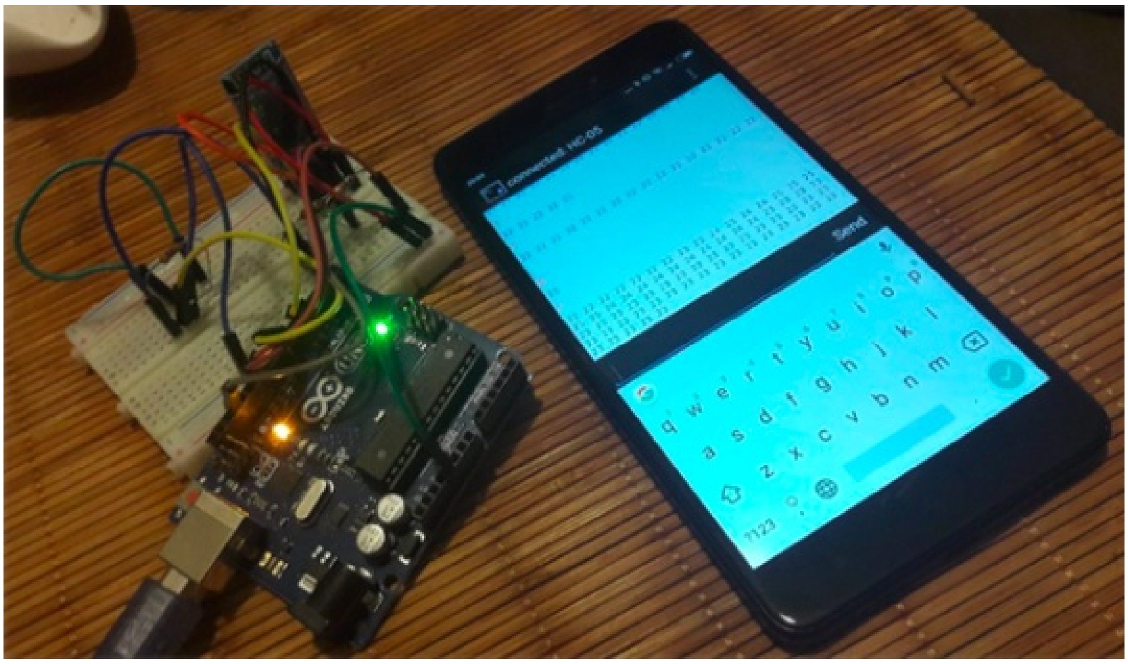
Javascript

There were a lot of complaints on Discord chat related to invoking smart contract using neon.js library, so here is an example, which is used in our project:

```
import Neon from '@cityofzion/neon-js';  //"@cityofzion/neon-js": "^3.3.0"
...

invoke(operation, gas, ...args) {
    return new Promise(
      (res, rej) => {
        const { net, address, scriptHash, privKey } = this;
        getBalance(net, address)
          .then(balances => {

            const intents = [
              {
                assetId: CONST.ASSET_ID.GAS,
                value: 0.000001,
                scriptHash
              }
            ];

            if (operation === Token.OPERATIONS.MINT) {
              intents.push(
                {
                  assetId: CONST.ASSET_ID.NEO,
                  value: args[0],
                  scriptHash
                })
            }

            const invoke = { operation, args, scriptHash };
            const unsignedTx = Neon.create.invocationTx(balances, intents, invoke, gas
);
            const signedTx = Neon.sign.transaction(unsignedTx, privKey);
            const hexTx = Neon.serialize.tx(signedTx);

            return rpc.queryRPC(CONST.DEFAULT_RPC.TEST, {
              method: 'sendrawtransaction', params: [hexTx], id: 1
            });
          })
          .then(m => res(m))
          .catch(err => rej(err));
      }
    );
}
```

Temperature data logger

Detector Device



The device includes the following components:

- Microprocessor
- Bluetooth-module
- Temperature sensor
- USB-module (for supply)
- Flash drive
- Battery (*will be added soon*).

After activation, the detector starts automatic data collection and writes the received data to the built-in flash drive.

The detector sends the received data to the mobile app via Bluetooth. Then the app **automatically writes down it to the blockchain**.

Data protection systems *(all below will be added soon)*

The data is protected by the electronic signature, which using the Sha256 encryption algorithm. With that protect the data goes to the app. This feature allows to protect the data from forgery and pinpoint the sender.

- The data is encrypted by the RSA algorithm, which allows to protect the data from dubbing and downloading directly from the flash drive by opening the detector device.
- Protection against copying is realized by setting fuse-bits.

Thus, the device puts down the data into the blockchain, signs a transaction and gives it to the app.

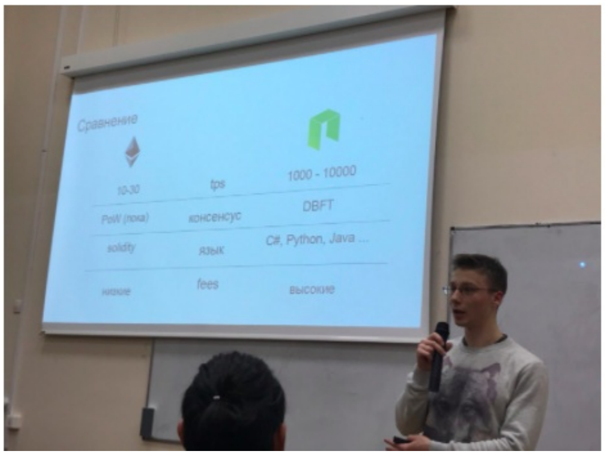
Characteristics

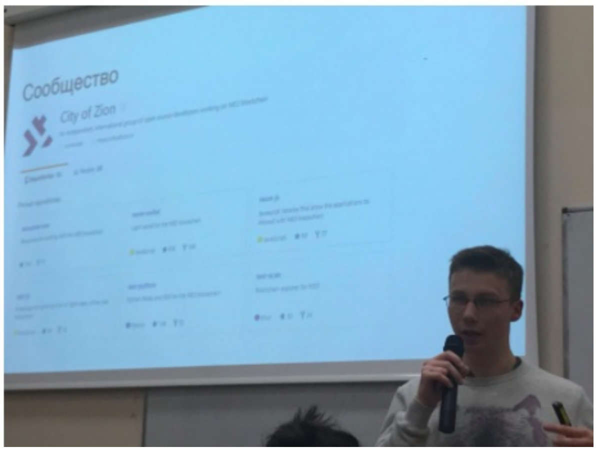
- Operating time with a full battery is 30 days
- Bluetooth range is 20 m.
- Overall dimensions: 70x50x9 mm.
- Weight - 200 grams
- Temperature range: -50 to 80 ºC

Our achievements

Current

1. Our team have already participated in City of Zion #1 with Smart Promise project. We've took a **"Honorable mentions"** nomination.
2. Elight project have reached the final stage of St. Petersburg State University Startups competition**.**
3. We've prepared **a presentation about Neo platform**. With that presentation Sergey Ankarenko has made a speech at St. Petersburg Blockchain Community Meetup and got **a lot of interest to the Neo platform** from the audience.





4. We have got support from **Anton Nazarov**, ***Russian regional director*** of Mediterranean Shipping Company, the world's second-largest shipping line in terms of container vessel capacity, and **Michael Lazarev**, ***CEO & Founder*** of St. Petersburg Blockchain Community **and** Tokenstarter**,** a blockchain-startup, in which we also participate.

5. Recently we've successfully presented our project to Association of Young Entrepreneurs of St. Petersburg, which is actively promoting the development of entrepreneurship among Russian and Chinese youth. We have also **told about Neo Smart Economy and City of Zion community**.

Future

We will

- participate in the 1st Neo Dev Competition;
- create a fully working MVP and test it in a real market conditions;
- create a detailed business model and business plan;
- participate at the final stage of St. Petersburg State University Startups competition;
- **try to expand the interest of Russian blockchain developers to the Neo platform by participating at meetups with our speech.**