1 Basic Serpent Contract Example

Before moving into more difficult examples, let's take a quick look at an Easy Bank example from KenK's first tutorial. A contract like this allows for a fully transparent bank to function with an open ledger that can be audited by any node on the network (an deal feature for ensuring banks aren't laundering money or lending to enemies of the state?)

Before looking at the code for the contract, let's consider what we want our simple bank to be able to do.

1. We need to setup at least one account with an initial balance
2. We want any account to be able to send the currency

```
def init():
        self.storage[msg.sender] = 10000
def code():
        to = msg.data[0]
        from = msg.sender
        value = msg.data[1]
        if self.storage[from] >= value:
                self.storage[from] = self.storage[from] - value
                self.storage[to] = self.storage[to] + value
```

So what's going on in this contract? Our contract is divided into two methods, let's take a look at the first method:

```
def init():
        self.storage[msg.sender] = 10000
```

Init in serpent is very similar to init in python, which is very similar to common constructors in Java. The init function runs once and only once at when the contract is created the block runs, instantiates the objects, and is never run again.

Our init method, from a general perspective, intializes one account with a balance of 10,000. In our Ethereum contract storage is handled with key =¿ value pairs. Every contract has their own storage, which is accessed by calling self.storage[key], So in our example, the easy bank's contract storage now has a value of 10,000 at key msg.sender (we'll identify what this is in a moment).

Awesome. So who is msg.sender? msg.sender is the person who is sending the specific message - which in this case is us. msg.sender is unique and assigned and verified by the network. We now have a heightened understanding of init, lets look at our send method.

```
def code():
        to = msg.data[0]
        from = msg.sender
        value = msg.data[1]
        if self.storage[from] >= value:
```

```
self.storage[from] = self.storage[from] - value
self.storage[to] = self.storage[to] + value
```

Let's take a look at this one piece at a time. The first three lines aim are setting up variables that we will use in the last three lines. The first line is establishing who we are sending our funds to - similar to the infamous Java main (String[] args), our contract storage will accept all arguments passed to it when it is called and store them in the array msg.data. Thus, we are sending our funds to the first argument that is passed to our contract, msg.data[0], which is the address of the recipient.

from is being set to the address that the funds are from, which is us - msg.sender. Finally, the value variable is set to the second argument passed to our contract, which is the value to be sent. Recall that the arguments passed to the contract are stored in the msg.data array, thus the second argument is at msg.data[1].

Okay, now that we understand what variables we are working with let's dive int the last portion of our contract. We want to check that the balance for the bank account in the contract's storage at from (us = from = msg.sender) is greater than or equal to the amount we are attempting to send - obviously we do not want our contract sending money that the sender does not have.

If the account balance passes our check we subtract the amount being sent from the sender balance:self.storage[from] = self.storage[from] - value. We then add to the balance of the account receiving the tokens:self.storage[to] = self.storage[to] + value.

2 Moderate Serpent Contract Example

So we've made it through the first serpent example, which we now have realized wasn't as daunting as it first seemed. We understand that every contract has its own contractual storage that is accessed through self.storage[key] = value. We understand that arguments passed to a contract are stored in the array msg.data[arg #], which is similar to Java's String[] args. Lastly we understand that msg.sender gives us the unique identifier of whoever sent the message, and that all participants involved with a contract have their own unique identifier that can be used in whatever creative way you would like.

Let's look at a more moderate contract that keeps with our bank theme. So, just like with our first contract, we need to classify what we are making and what characteristics the contract will need to leverage our desired features. We are going to be implementing what is known as a mutual credit system. A generalized idea of a mutual credit system is the intersection of a barter system and a non-regulated currency model. So, let?s define a community that implements a mutual credit system and every participant gets a 1000 Unit credit (in this case 1UC = 1USD). In the beginning there is no money at all. It only comes into circulation when one of the participants uses his credit to pay another participant. If he uses his 1000 Units his balance is ? 1000 U. His supplier?s balance is now +1000U. The total amount in circulation is now also 1000 U. This means there is always exactly as much in circulation as there is outstanding credit: a zero sum game.

One can clearly notice that this system creates money at the time of the transaction. At time 0, before any of the community's participants completed a transaction, the currency in circulation was zero. It is also clear that, unlike fiat currencies, this model does not require any centralized money supply management, which when discussing decentralized apps running on blockchain technology is an attractive idea to implement. Regardless of your opinion on such a system, let's automate a contract to initiate these transactions and act as a public ledger to keep track of the community's participants and their account balances.

```
def init():
        contract.storage[((msg.sender * 0x10) + 0x1)] = 0x1
        contract.storage[((msg.sender * 0x10) + 0x2)] = 0x1

def code():
        toAsset = (msg.data[0] * 0x10) + 0x1
        toDebt = (msg.data[0] * 0x10) + 0x2
        fromAsset = (msg.sender * 0x10) + 0x1
        fromDebt = (msg.sender * 0x10) + 0x2
        value = msg.data[1]

        if contract.storage[fromAsset] >= value:
                contract.storage[fromAsset] = contract.storage[fromAsset] - v
        else:
                contract.storage[fromDebt] = value - contract.storage[fromAss
                contract.storage[fromAsset] = 0

        if contract.storage[toDebt] >= value:
                contract.storage[toDebt] = contract.storage[toDebt] - value
        else:
                value = value - contract.storage[toDebt]
                contract.storage[toAsset] = contract.storage[toAsset] + value
                contract.storage[toDebt] = 0
```