LESSON SC 15 – Last Changes For Convenience

University of West Attica
Department of Electrical and Electronics Engineering
Ioannis Christidis
Christoforos Kachris

Support by Ethereum Foundation ESP

What will we accomplish!

In this lesson we will do some small adjustments to our SC to get it ready for testing in Sepolia testnet.

The changes we are going to make are the following:

1. We are going to add the `address` of the renter inside the `Car struct` so that we know who is able to unlock the car.
2. We are going to change the `constant values` of our SC for the testing, so that we do not run out of funds or wait a lot of time.
3. We are going to protect our SC from situations where the microcontroller does not have enough ETH to make a transaction. Because if the microcontroller tries to change the status of its car and it has no funds the transaction will `revert`.
4. We are going to make some changes to the `ICRFactory` to know which microcontroller `address` is used on which `ICR` SC.

Adding renter inside the car struct

We can start by adding an `address` named `currentRenter` in the `Car struct`.

When the car is registered, we need to add the `currentRenter`.

We will add the `owner`.

So, if someone has rented the car the `currentRenter` will be their `address` else it will be the `owner`. The car will only be unlocked to the `currentRenter`.

```
//...
struct Car {
    address mc;
    uint256 price;
    Status status;
    uint256 timeOfLastRent;
    address currentRenter;
}
//...
function registerCar(address _mc, uint256 _price) public nonReentrant {
    //...
    Car memory car = Car(_mc, _price, Status.UNAVAILABLE, 0, i_owner);
    //...
}
//...
```

Next we will change the functions that change the status of cars.

Adding renter inside the `car struct`

At the `rentCar` function we can change the `currentRenter` to the one that pays the car's price to rent it, meaning the `msg.sender`.

```
    //...
    function rentCar(uint _carId) public payable carIsValid(_carId) isPaused() nonReentrant {
        //...
        Car storage car = s_cars[_carId];
        car.status = Status.OCCUPIED;
        car.timeOfLastRent = block.timestamp;
        car.currentRenter = msg.sender; //here we declare the msg.sender as current renter
        emit CarRented(_carId, car.mc, msg.sender);
    }
```

Next at the `changeCarStatusMc` that microcontrollers use, we change the `currentRenter` back to the `owner` of the cars.

```
    function changeCarStatusMc(uint256 _carId) public carIsValid(_carId) nonReentrant {
        //...
        Car storage car = s_cars[_carId];
        car.currentRenter = i_owner;
        //...
    }
```

And with that, our microcontrollers can always find who is the current renter.

Changing `constant values`

Now, we are going to change the `constant values` of our SC to make testing easier with the microcontrollers.

For example, if we leave renting time to 1 day we will have to wait that long to see if our DApp works.

Since our SepoliaETH also have a limit, we should reduce the cars' price limits to a value we can provide during testing.

```
    uint256 private constant LOWER_PRICE_LIMIT = 0.0001 ether;
    uint256 private constant UPPER_PRICE_LIMIT = 0.0002 ether;

    uint256 internal constant RENT_TIME = 6 minutes;
```

Ensure microcontrollers have funds

In our DApp, we are going to be sending transactions from our microcontrollers. To do that we need to ensure that microcontrollers have enough ETH to make a transaction.

What we will do is:

Declare a *minimum limit* of ETH that a microcontroller should possess.

Each time we try to change a car's status to `AVAILABLE`, we will check if the microcontroller has enough funds to make a transaction and if not, we will make the car `UNAVAILABLE`.

To make the car available again the owner will have to fund the microcontroller externally.

It is possible to calculate the amount of ETH required to cover the gas price at the time of a transaction (since gas prices are not constant). However, this requires additional logic.

To keep things simple, we will use a flat ETH value that the microcontroller must hold. This value will be set slightly higher to ensure the SC's functionality remains unaffected even if gas prices increase significantly.

We will add a `custom error` and the `constant` variable.

```
error ICR__MCBalanceIsBelowMinimum(address _mc, uint256 _mcBalance);

uint256 internal constant MINIMUM_MC_BALANCE = 0.001 ether;
```

Ensure microcontrollers have funds (2)

In the `ICRManager` we will change the logic of `changeCarStatusOwner` to `revert` when the `owner` tries to make the car `AVAILABLE` if the microcontroller does not have enough funds.

```
    function changeCarStatusOwner(uint256 _carId) public carIsValid(_carId) nonReentrant {
        if (msg.sender != i_owner) {
            revert ICR__IsNotOwner(msg.sender);
        }
        if (s_cars[_carId].status == Status.OCCUPIED) {
            revert ICR__CarIsNotInTheCorrectStatus(_carId, s_cars[_carId].status);
        }
        Car storage car = s_cars[_carId];
        if(car.status == Status.AVAILABLE) {
            car.status = Status.UNAVAILABLE;
            emit CarIsUnavailable(_carId, car.mc);
        } else {
            if(car.mc.balance <= MINIMUM_MC_BALANCE) { //check if mc does not have enough balance
                revert ICR__MCBalanceIsBelowMinimum(car.mc, car.mc.balance); // revert if it does not
            }
            car.status = Status.AVAILABLE;
            emit CarIsAvailable(_carId, car.mc);
        }
    }
```

Ensure microcontrollers have funds (3)

Similarly, we change the `changeCarStatusMc` but instead of reverting we will make the car `UNAVAILABLE`. This way it will not stay `OCCUPIED` by the renter and only the `owner` can change it back to `AVAILABLE`, if first provides the funds needed.

```
    function changeCarStatusMc(uint256 _carId) public carIsValid(_carId) nonReentrant {
        if (msg.sender != s_cars[_carId].mc) {
            revert ICR__IsNotMc(_carId, msg.sender);
        }
        if (s_cars[_carId].status != Status.OCCUPIED) {
            revert ICR__CarIsNotInTheCorrectStatus(_carId, s_cars[_carId].status);
        }
        if (s_cars[_carId].timeOfLastRent + RENT_TIME > block.timestamp) {
            revert ICR__RentingTimeIsNotOverYet(s_cars[_carId].timeOfLastRent + RENT_TIME, _carId);
        }
        Car storage car = s_cars[_carId];
        car.currentRenter = i_owner;
        if(car.mc.balance <= MINIMUM_MC_BALANCE) { // check if mc has enough funds
            car.status = Status.UNAVAILABLE; // changes car to unavailable
            emit CarIsUnavailable(_carId, car.mc);
        } else {
            car.status = Status.AVAILABLE; // else make it available
            emit CarIsAvailable(_carId, car.mc);
        }
    }
```

If we want, we can check during the `rentCar` function to make sure that the user will not spend ETH, if the microcontroller does not have enough funds to operate.

Find Microcontrollers on ICRFactory

To find in which ICR SC a microcontroller is registered we can alter the s_isMc mapping that returns a true/false boolean to return the address of the ICR SC .

```
mapping(address _mc => address _icr) private s_mcToIcr; //change s_isMc to s_mcToIcr
```

Next in the registerCarToICR function we can check if the microcontroller is used, by checking if s_mcToIcr mapping with key the _mc address will return the address(0), meaning it is not used in any ICR SC.

```
function registerCarToICR(uint _icrId, uint _price, address _mc) external onlyOwner nonReentrant {
    if(s_mcToIcr[_mc] != address(0)) { // check if the mapping returns address(0) else revert
        revert ICRFactory__AddressIsMC(_mc);
    }
    address icrAddress = s_icrs[_icrId];
    s_mcToIcr[_mc] = icrAddress; // update the mapping
}
```

Remember to add a getter function for this because we are going to use it from our microcontrollers.

```
function getMcToIcr(address _mc) public view returns (address) {
    return(s_mcToIcr[_mc]);
}
```

Outro

Everything is set up to go to test our DApp with microcontrollers.

Be sure to test the DApp locally first because with the microcontrollers we will have to test in Sepolia testnet.

DISCLAIMER:

Normally during and after the development of a SC we would write some unit and staging tests to ensure that it works properly localy and on testnets. There are testing frameworks like **hardhat** or **foundry** we could use for that, but since this is not the scope of this course, we will not go into that for now.