7 Adding Inline Actions

This section provides instructions for creating inline embedded calls (inline) for `actions`. Actions may be called by another action of the same contract. Examples of these instructions are shown on the addressbook address book contract, the creation of which is described in Section 5 of this manual.

7.1 Add cyber.code for permission to action

In order for the action in the address book contract to trigger another action, the contract account must be given permission to do so. To do this, add `cyber.code` by executing:

```
cleos set account permission addressbook active --add-code
```

The `cyber.code` authority allows the contract to perform embedded calls to actions.

7.2 Send notification

Open the `addressbook.cpp` contract and create an additional action that will behave like a transaction that receives a notification. To do this, you can create a helper function in the address book class.

```
[[eosio::action]]
void notify(name user, std::string msg) {}
```

This function will only accept the name and string.

7.3 Create a copy of the transaction and send it to the sender as a notification

Create a copy of the transaction in which the action was loaded and send this copy to the user to be accepted. To do this, you can use the `require_recipient` method. Calling `require_recipient` adds an account to the `require_recipient` set and ensures that these accounts receive notification of the action being performed. The notification is similar to sending accounts to an "exact copy" of the action.

To exclude the possibility of the unauthorized calling of this function by a third-party user, it is necessary to require authorization to perform the action. At the same time authorization must be provided by the contract itself. To do this, add the `get_self()` method to authorize yourself.

```
[[eosio::action]]
  void notify(name user, std::string msg) {
    require_auth(get_self());
    require_recipient(user);
  }
```

If an attempt is made to call this function, someone other than the contract itself will generate an exception.

7.4 Notify helper function about sending called action

Since an operation can be called more than once using the built-in invocation, it is necessary to create an auxiliary function to reuse the code in order to reduce duplication.

```
...
  private:
    void send_summary(name user, std::string message){}
```

Inside this helper you need to create an action and send it.

7.5 Create a constructor actionp

7.5.1 Modify an `addressbook` contract to send receipts to the user each time the contract is invoked. It is necessary to take into account a case when a record is absent from the table.

Save this object as `notification` for action.

```
...
  private:
    void send_summary(name user, std::string message){
      action(
        //permission_level,
        //code,
        //action,
        //data
      );
    }
```

The following parameters are required in the action:

- permission_level `permission_level` structure;
- contract to call (initialized using the type `eosio::name`);
- action (initialized using the type `eosio::name`);
- data passed to the action.

7.5.2 Initialize the permission structure.
In the contract, the permission must be authorized by the account of the `active` contract with `get_self()` . To use the built-in «active» credentials, it is necessary that the contract provides active credentials to the `cyber.code` code.

```
...
  private:
    void send_summary(name user, std::string message){
      action(
        permission_level{get_self(),"active"_n},
      );
    }
```

7.5.3 Enable `get_self()` .

Since the called action is placed in the contract directly, you must use `get_self()` . It will also work in the address book `\"addressbook\" _n` . Using `get_self()` is the most acceptable option, since the contract doesn't work if it's deployed under a different account name.

```
...
  private:
    void send_summary(name user, std::string message){
      action(
        permission_level{get_self(),"active"_n},
        get_self(),
        //action
        //data
      );
    }
```

7.5.4 Enable the `_n` operator.

The `notify` action was previously defined to invoke this inline action. Therefore, it is necessary to use the `_n` operator.

```
...
  private:
    void send_summary(name user, std::string message){
      action(
        permission_level{get_self(),"active"_n},
        get_self(),
        "notify"_n,
        //data
      );
    }
```

7.5.5 Define the data to be transferred to the action.

The `notify` function takes two parameters, a name and a string. The constructor action expects data as a `bytes` type, so you must use the `make_tuple` method, available through the `std C++` library. The sent data are determined by the order of parameters received by the called action.

Pass the `user` variable, specified as the `upsert()` action parameter. Combine the string containing the user name and include `message` to send the `notify` notification to the action.

```
...
  private:
    void send_summary(name user, std::string message){
      action(
        permission_level{get_self(),"active"_n},
        get_self(),
        "notify"_n,
        std::make_tuple(user, name{user}.to_string() + message)
      );
    }
```

7.5.6 Send an action using the `send` method.

```
...
  private:
    void send_summary(name user, std::string message){
      action(
        permission_level{get_self(),"active"_n},
        get_self(),
        "notify"_n,
        std::make_tuple(user, name{user}.to_string() + message)
      ).send();
    }
```

7.6 Call the auxiliary function and enter the appropriate message

At this point, the helper function is defined and can be called. There are three events when a new `notify` helper can be triggered:

- After the contract added a new entry:
  `send_summary (user, «the address book entry was added successfully»)` .
- After the contract changed an existing entry:
  `send_summary (user, «the entry in the address book has been changed successfully»)`
  .

- After the contract deleted an existing entry:
  `send_summary (user, «the entry from the address book was successfully deleted»)`
  .

7.7 Make a change to the macro EOSIO_DISPATCH

As the `notify` action was added to the contract, you need to update the EOSIO_DISPATCH macro to enable this action to be available for external execution (for example, an external transaction, a call to an operation (method, action). This ensures that the `notify` action is not excluded by the `eosio.cdt` optimizer.

```
EOSIO_DISPATCH( addressbook, (upsert)(erase)(notify) )
```

The `addressbook` contract will be:

```
#include <eosiolib/eosio.hpp>
#include <eosiolib/print.hpp>

using namespace eosio;

class [[eosio::contract]] addressbook : public eosio::contract {

public:
  using contract::contract;

  addressbook(name receiver, name code,  datastream<const char*> ds): contra

  [[eosio::action]]
  void upsert(name user, std::string first_name, std::string last_name, uint
    require_auth(user);
    address_index addresses(_code, _code.value);
    auto iterator = addresses.find(user.value);
    if( iterator == addresses.end() )
    {
      addresses.emplace(user, [&]( auto& row ) {
       row.key = user;
       row.first_name = first_name;
       row.last_name = last_name;
       row.age = age;
       row.street = street;
```

```
        row.city = city;
        row.state = state;
      });
      send_summary(user, " successfully emplaced record to addressbook");
    }
    else {
      std::string changes;
      addresses.modify(iterator, user, [&]( auto& row ) {
        row.key = user;
        row.first_name = first_name;
        row.last_name = last_name;
        row.street = street;
        row.city = city;
        row.state = state;
      });
      send_summary(user, " successfully modified record to addressbook");
    }
  }

  [[eosio::action]]
  void erase(name user) {
    require_auth(user);

    address_index addresses(_self, _code.value);

    auto iterator = addresses.find(user.value);
    eosio_assert(iterator != addresses.end(), "Record does not exist");
    addresses.erase(iterator);
    send_summary(user, " successfully erased record from addressbook");
  }

  [[eosio::action]]
  void notify(name user, std::string msg) {
    require_auth(get_self());
    require_recipient(user);
  }

private:
  struct [[eosio::table]] person {
    name key;
    std::string first_name;
    std::string last_name;
    uint64_t age;
    std::string street;
    std::string city;
    std::string state;

  uint64_t get_secondary_1() const { return age;}

  };

  void send_summary(name user, std::string message) {
    action(
      permission_level{get_self(),"active"_n},
      get_self(),
```

```
81        "notify"_n,
82        std::make_tuple(user, name{user}.to_string() + message)
83      ).send();
84    };
85
86    typedef eosio::multi_index<"people"_n, person,
87      indexed_by<"byage"_n, member<person, uint64_t, &person::age>>
88    > address_index;
89
90  };
91
92  EOSIO_DISPATCH( addressbook, (upsert)(notify)(erase))
```

7.8 Recompile and edit the ABI file

Recompile the contract with the `--abigen` flag, since the contract has been modified to affect ABI.

```
1  cd CONTRACTS_DIR/addressbook
2  eosio-cpp -o addressbook.wasm addressbook.cpp --abigen
```

The contracts for EOSIO are available for the update so this contract 'addressbook' can be transferred with the changes.

```
cleos set contract addressbook CONTRACTS_DIR/addressbook
```

As a result of the command execution, the following information should appear:

```
1  Publishing contract...
2  executed transaction: ...
3  #         eosio <= eosio::setcode               {"account":"addressbook","vr
4  #         eosio <= eosio::setabi                {"account":"addressbook","ab
```

7.9 Testing

In the previous section, the `alice` user's address book entry was deleted during the testing phase, therefore, when calling `upsert`, the built-in action is run.

Execute:

```
cleos push action addressbook upsert '["alice", "alice", "liddell", 21, "123
```

As a result of the command execution, the following information should appear:

```
executed transaction: ...
#   addressbook <= addressbook::upsert          {"user":"alice","first_name"
#   addressbook <= addressbook::notify          {"user":"alice","msg":"alic
#         alice <= addressbook::notify          {"user":"alice","msg":"alic
```

The text (at the bottom of the output) reports that the address book notifies (`addressbook::notify`) the user `alice` about the transaction.

The `cleos get actions` command can be used to view the actions related to the `alice` user.

```
cleos get actions alice
```

The result of the command will be the following information:

```
#  seq  when     contract::action => receiver      trx id...   args
=================================================================
#   ...       addressbook::notify => alice         685ecc09... {"user":"ali
```