

# #14 Reward Distribution Contest: Telegram/TON Contests Winners

## Contest entry submission

**Shiny Giraffe** (GitHub: [Skydev0h](#))

December 17, 2020

### Abstract

This is the description of a contest entry submission to FreeTON Contest #14: Reward Distribution Contest: Telegram/TON Contests Winners. Moreover, the contest entry submission can be considered a withdrawal request as per mentioning of such entity in Dates section of the contest specification.

### Introduction

This document consists of several parts, each of which outlines an important aspect constituting to the contest entry. For your convenience, navigable **table of contents** follows the introduction after a page break.

The first chapter, **Origin**, provides reasonable information about original contest entries and underlying smart-contracts without deep technical details.

Second chapter, **Integration**, reveals what was done to integrate contest entries into FreeTON, and what is planned to be done in future as FreeTON evolves.

In the third chapter, **Implementation**, you will be presented with some technical implementation details of the integrated solutions.

The fourth chapter, **Demonstration**, presents example use of integrated solutions and informs how you can try it yourself, if needed.

Fifth chapter, **Deployment**, shortly outlines how you currently can deploy the contract to the network and use it as needed.

Finally, in the sixth chapter, **Authorship**, you will get to know an author of this document and proofs of ownership over different accounts along with their relations.

*I also should mention some additional necessary information in the introduction for the sake of possible increase of integrity and unambiguity:*

*My FreeTON wallet and TON Surf messaging address is the following:*

[0:954688c088881241c5c6b248da398aefa2752bd5a97202f41454fdf3671e671c](#)

*As per provided table I should receive 215 000 Tons in total for this contest.*

## Table of Contents

1	Origin .....	3
1.1	Rationale.....	3
1.2	Conditional transfer smart contract.....	4
1.3	Example CTSC use cases .....	5
1.3.1	Escrow.....	5
1.3.2	Custody .....	5
1.3.3	Automatic crowdfunding.....	6
1.3.4	Controlled crowdfunding .....	6
1.3.5	Predictable splitting.....	7
1.3.6	Creative ideas.....	7
2	Integration .....	8
2.1	Integration plan .....	8
2.1.1	Rewriting the contract .....	8
2.1.2	Creating a user interface.....	8
2.1.3	Building a web explorer.....	9
2.1.4	Optimizing gas costs.....	9
2.1.5	Analyzing and reviewing the code.....	9
2.1.6	Adding new features .....	9
2.2	Current state and future plans.....	9
3	Implementation.....	10
4	Demonstration.....	15
4.1	Automatic crowdfunding (simplified).....	15
4.2	Custody.....	16
4.2.1	Successful funds release .....	16
4.2.2	Returning funds to sender.....	17
4.3	Distribution rules .....	18
5	Deployment.....	20
6	Authorship.....	23
6.1	Definition of an author of this document .....	23
6.2	Proof of authorship and ownership.....	23
6.3	The proof relation graph.....	24

# 1 Origin

## 1.1 Rationale

For the Telegram Contests I have created several smart contracts:

- Manual DNS
- Automatic DNS
- Conditional transfer
- Data storage proxy

Out of those contracts, the most suitable, and actually the only plausible smart contract to be integrated and be useful for FreeTON is **CTSC** – Conditional transfer smart contract.

Rationale behind this is that DNS smart contracts are very low level and tied to origin Telegram network high-level protocols. Moreover, those contracts were enhanced, fixed, and integrated into TON node core by Telegram developers. As I understand for now there is no required high-level infrastructure and protocols for DNS smart contracts in FreeTON system without which integrating this contract is not possible. Judging from the current state of ecosystem, I'd expect either that FreeTON developers have already rewritten my contract or made their own, or will make a contest for DNS smart-contract or even protocol later in time.

As for the Data storage proxy smart contract, it is also very low level, and uses very intricate features of TVM, that yet have not been implemented in FreeTON compilers. It is worth mentioning, that due to specifics of message handling in FreeTON Solidity, the **DSPSC** may be even not needed in such ecosystem at all, because rich inter-contract interaction features of Solidity actually are clearer, performant, and feature-rich than basic templating system of this contract. It may be worth reviewing and integrating this during the ongoing integration phase, but as of now this contract is out of scope.

Finally, the remaining contract that is to be integrated into FreeTON is Conditional transfer smart contract, that would be referred as to **CTSC**. Such contract may be useful to FreeTON network because of its versatility, flexibility and usefulness. It allows to reign some aspect of DeFi automatically, predictably and decentralizedly.

Rules of such contract are "set in stone" and are enforced by the great machinery of the entire FreeTON network. Moreover, they are transparent and can be viewed and verified by any user of the network. This makes the **CTSC** a safe and reliable mechanism to transfer FreeTON Crystals (Tons) when some condition is met.

On the other hand, the **CTSC** can be very flexibly configured to fill the gap for many possible applications. Some application examples include escrow, automatic and controlled crowdfunding, custody, predictable splitting, and it does not end with that.

## 1.2 Conditional transfer smart contract

So now, I am going to describe the **CTSC** a little more in depth.

If desired, you may watch a video presentation describing mechanics of the contract at the following link: <https://www.youtube.com/watch?v=bUYS8PS8Vqk>. You may also visit a web page dedicated to this smart contract if you desire to read some extra information right away: <https://skydev0h.github.io/ton-freestyle-2/ctsc.html>.

In a nutshell, **CTSC** accepts some TON crystals from one or more sources and routes them to one or more destinations, but only if a specific condition is met.

If the condition fails to succeed, deposited TON crystals can be very simply returned just by sending an ordinary message to the contract from the depositing wallet.

The following list is concise outline of the most important contract features:

It is possible to set many different conditions for the fund transfer:

- Number of collected TON crystals can be set as a condition:
  - Minimum required collected Tons that are needed to release the funds
  - Maximum permitted collected Tons that can be collected by contract overall
  - Minimum accepted amount per each transaction to prevent various attacks
- Important time constraints may (and should) be set:
  - Collection deadline by which minimum required Tons must be collected
  - Release locktime by which it is not possible to act upon collected Tons
  - Release deadline after which Tons are automatically released or refunded
- Various modes that provide flexibility to the contract operation:
  - Automatic release or refund when release deadline occurs
  - Continuous collection mode that allows to collect more tons after deadline
- Different levels of control over the conditions:
  - Automatic mode works without any control and relies only on preset conditions
  - Controlled mode depends upon actions of an external user or smart contract
- Splitting between destinations can be prescribed in the contract:
  - Percentage of total collected sum can be sent (of all sum or remaining one)
  - Some fixed Ton amount can be distributed
  - Ultimate beneficiary will receive all remaining Ton

Contract operation is very easy for all parties:

- Controller can use simple commands to generate and send required messages
- Moreover, internal controller can send simple text commands from Surf
- Investors do not need to use any scripts at all even to return funds
- The contract responds in Surf with English text that makes it very convenient

These features can be combined for interesting use cases that follow this page.

## 1.3 Example CTSC use cases

Here I will briefly describe the most prominent and common use cases of the contract, and how such use case can be attained.

### 1.3.1 Escrow

For this case it is possible to temporarily safely lock funds into the contract and release them to destination if some preset agreements are met.



The trusted external entity, that can be either external user or a smart contract, would either release funds to receiver, or return them to sender.

### 1.3.2 Custody

This case differs from the previous in that the receiver themselves control releasing.



This way, TON Crystals can be safely transferred to destination only when they are ready to accept them, or return the funds back otherwise after some timeout.

### 1.3.3 Automatic crowdfunding

The contract may collect Tons from investors and automatically send them to destination if the collection target is met by a set time.



Depending on the configuration, the contract may either automatically transfer all collected funds to the beneficiary as soon as required sum is collected in the contract or obey some deadlines that may require for the sum to be obtained before some set time and for it to be released no sooner than set point in time.

### 1.3.4 Controlled crowdfunding

This is a more realistic use case, that would involve an external entity to confirm whether to release funds to beneficiaries or return them to senders. Such controller can be either external user or another smart contract.

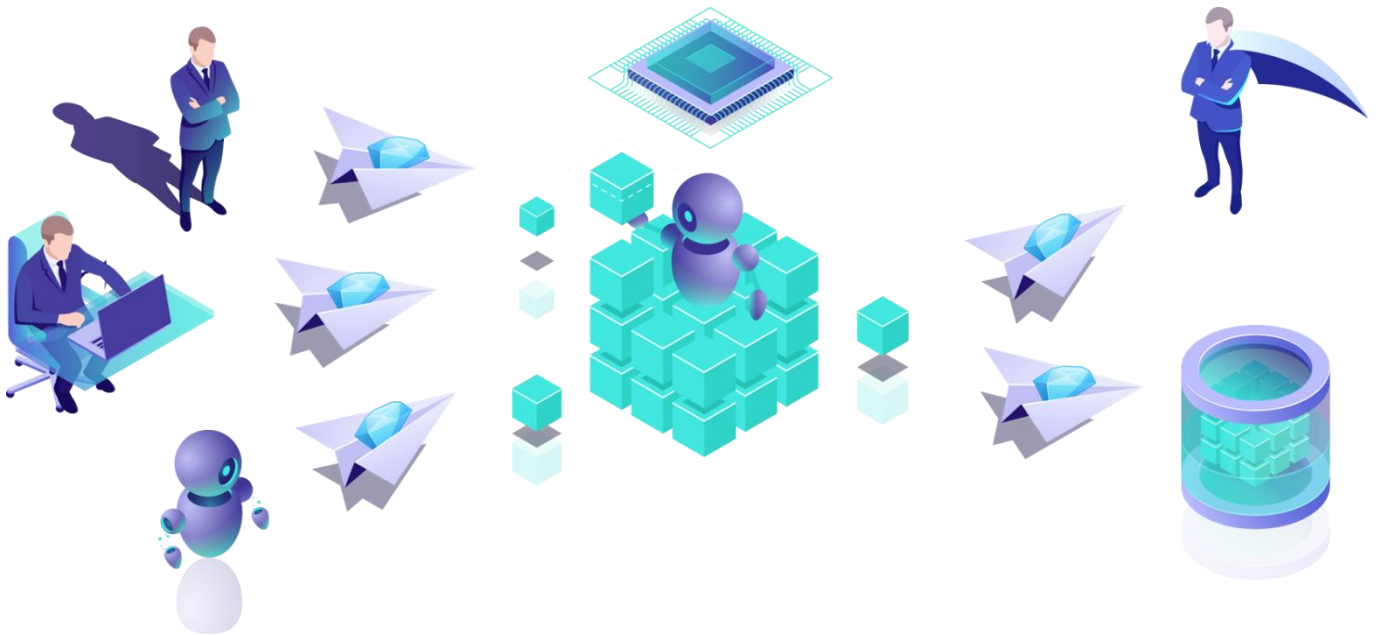


Therefore, funds would be transferred only after confirmation by controller. For more sophisticated controlling, such as vesting, special destination contract can be used.



### 1.3.5 Predictable splitting

By using beneficiaries table in required fashion, it is possible to split all coming funds to specified destinations in defined proportions or amounts.



Such usage involves simply proxying crystals through this contract, but outgoing funds are predictably and transparently calculated and sent to beneficiaries.

### 1.3.6 Creative ideas

The list does not just end there. Using features and settings of the contract it is possible to construct countless different combinations and invent many more interesting use cases for this contract.



By using other smart contracts for different purposes, either as a controller, sender or beneficiary of the funds, it is certainly possible to bridge the gap for any possible required use case, and this contract would be a reliable cog in the complex system.

## 2 Integration

### 2.1 Integration plan

As I expect this smart contract to be reliable, user friendly, economic and useful, the integration process is rather complex. Counting the fact that I have never written a Solidity smart contract before, and that TON Labs Solidity dialect is rather unusual, and is constantly undergoing evolution over time, writing code and keeping up with all the new and very useful features is no easy task. Therefore, it is not possible to do all my plans at once, and I have outlined a specific integration plan.

#### 2.1.1 Rewriting the contract

**The first**, but a very large and crucial, **step is rewriting the contract in Solidity.** Actually, in many parts some logic was completely left out (because it is now handled automatically), on the other hand in some parts it was required to write more logic due to some language specifics.

Although, the language worked out to be so easy and convenient to use that I have even added several very useful and convenient features to the contract.

**Although the following steps are numerated, I can execute them in any order. Therefore consider them to be a list of objectives rather than a strict plan.**

#### 2.1.2 Creating a user interface

**The second step** that I have considered is **making a convenient user interface.** For the perfect native way to do that I am planning to **write a DeBot** that would handle deployment and interaction with the contract, and make all those related tasks very user friendly and easy.

***Actually speaking, waiting for this very anticipated feature is the very reason why I am submitting this contest entry so late. But I can't afford to wait more.***

Nevertheless, despite that as of now DeBots and DEngine are released, they are still not ready to the extent necessary to implement this step as I **require** to **thoroughly test** the result of my work, especially because it automatically handles real currency. It is still not possible to *talk* to a DeBot using the official Surf clients, and *tonos-cli* DeBot interaction experience seems to still be very far from the one that I expect to receive in the Surf.

Therefore, this step is now impossible to complete, although it is planned to be done in the ongoing integration phase (the one that is at least three months after the contest, but I expect to support my creation beyond that period if it turns out to be actually useful in the ecosystem).



### 2.1.3 Building a web explorer

The third step that I consider is to **build a web explorer UI** to be able to easily find and inspect **CTSC** contracts in the blockchain. I expect that it would be easy to inspect the desired contract, see its entire configuration and state without blowing out your mind and being 100% sure that the contract is original, was not modified, and that it would behave exactly as it is configured to. I may even provide interface to deploy and interact with those contracts if I would not do that earlier in a DeBot.

### 2.1.4 Optimizing gas costs

The major hurdle for me when rewriting code in Solidity was loss of the entire control over the TVM code that was generated for me by compiler. The FunC language is so low-level that I can easily see what instructions are generated, but writing anything complicated is extremely complex and bug-prone. Solidity, on the other hand, encapsulates all the fuss away (such as storing data, replay protection, message routing and ABI) but at a cost of breaking the link between high-level and low-level code. Therefore, as **a fourth step** I would like to carefully and thoughtfully **analyze the code, low-level instructions and try to optimize gas usage**, because it is obviously higher due to all the conveniences that Solidity brings.

### 2.1.5 Analyzing and reviewing the code

Due to large size and complexity of the **CTSC** I think that as **a fifth step** I should **review, analyze code and fix found bugs**. It is not possible to test 100% everything, therefore I think that this process should be continuing all the time.

### 2.1.6 Adding new features

And **as a sixth step** I shall not forget to **invent and add new features to the CTSC**. Even while simply rewriting this code to Solidity I thought of several interesting features that I have implemented, so I hope that I would not have problems here.

## 2.2 Current state and future plans

As of now, I have completed **the first step**, have partially done **fifth and sixth steps** (although they are ongoing ones), and am eager to do **the second step** as soon as DeBots shine in their glory and be usable in Surf.

Until then I will **work on the contract (steps 4, 5, 6)** and consider strategy for **web explorer (step 3)**. It may be also worth mentioning that as I go on with those plans, I also relatively inadvertently work on coinciding stuff. For example, during the development of **CTSC** Solidity version I have **enhanced IntelliJ IDE Solidity plugin** and [published the changes in my GitHub repository fork](#). Moreover, during development and testing I have written simple Bash shell automation for deploying, testing and locally testing the smart contract, and those scripts are published too in the code repository. Therefore, **I would like to ask you to take that into account**.

### 3 Implementation

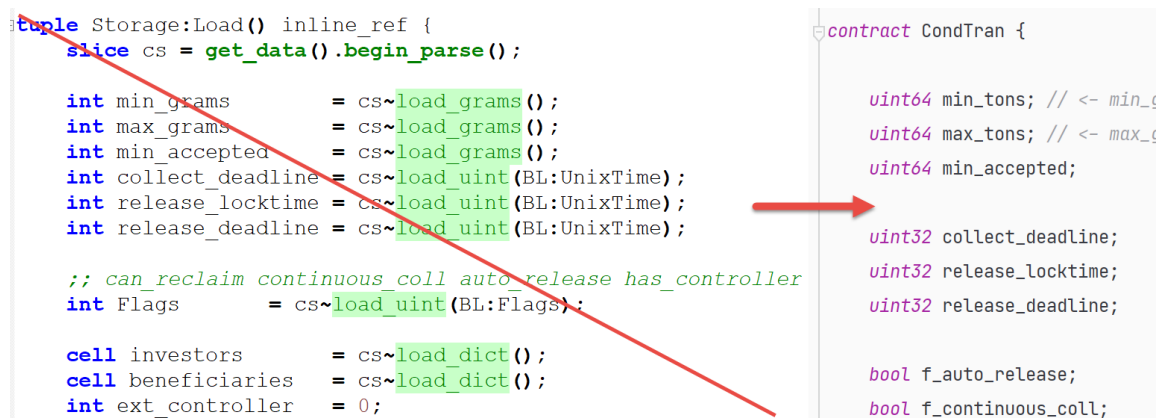
The actual implementation is an interesting story, during which I would like to allow myself to make comparisons of the old Fift + FunC smart contract with the shiny new Solidity + TON OS one. Alright, let's get started.

First and foremost, I want to start by providing the link to a repository that contains all the resulting code of the integration implementation and other useful goodies. It is: <https://github.com/Skydev0h/tg-contest-freeton>, you may check it out.

I have rewritten this smart contract going from the easiest parts to hardest.

Objectively, easiest functions to consider as first candidates are controller helper functions (each being up to ten lines of code) and getter functions (spanning around the same). Message handlers and funds release logic is very complex and large, therefore it was done a little later. And that even not counting substantial changes and tricks discovered during migrating to Solidity.

The most noticeable thing is that the TON Labs Solidity takes care control over that aspect and automagically loads contract information and to the storage. Therefore, the Storage:Load and Save functions became useless and not used anymore. The only thing that had to be set is storage mutability of the function (read/write, read only, none) but that is reasonably simple to consider.



```


stuple Storage:Load() inline_ref {
    slice cs = get_data().begin_parse();

    int min_grams      = cs~load_grams();
    int max_grams      = cs~load_grams();
    int min_accepted   = cs~load_grams();
    int collect_deadline = cs~load_uint(BL:UnixTime);
    int release_locktime = cs~load_uint(BL:UnixTime);
    int release_deadline = cs~load_uint(BL:UnixTime);

    ;; can_reclaim continuous_coll auto_release has_controller
    int Flags          = cs~load_uint(BL:Flags);

    cell investors      = cs~load_dict();
    cell beneficiaries  = cs~load_dict();
    int ext_controller  = 0;
}


```

```

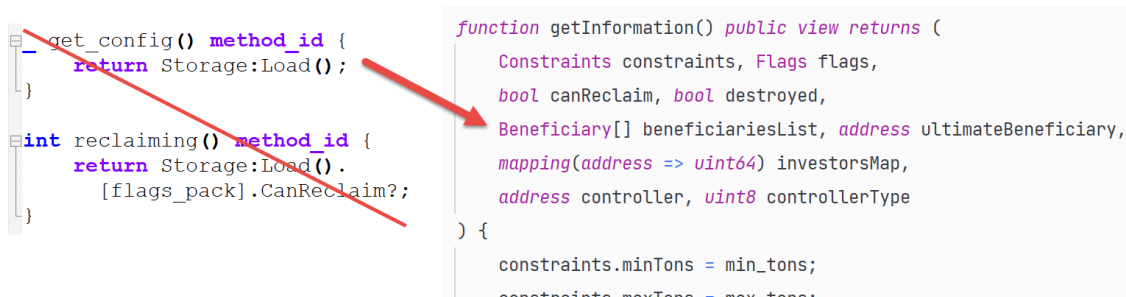
contract CondTran {
    uint64 min_tons; // <- min_tons;
    uint64 max_tons; // <- max_tons;
    uint64 min_accepted;

    uint32 collect_deadline;
    uint32 release_locktime;
    uint32 release_deadline;

    bool f_auto_release;
    bool f_continuous_coll;
}

```

Another almost immediately noticed thing was that I couldn't anymore just lazily return all the contract data as config, and that's it. In Solidity I had to actually walk through what I want to return and prepare the output data. I consider that good.



```


get_config() method_id {
    return Storage:Load();
}

int reclaiming() method_id {
    return Storage:Load().
        [flags_pack].CanReclaim?;
}


```

```

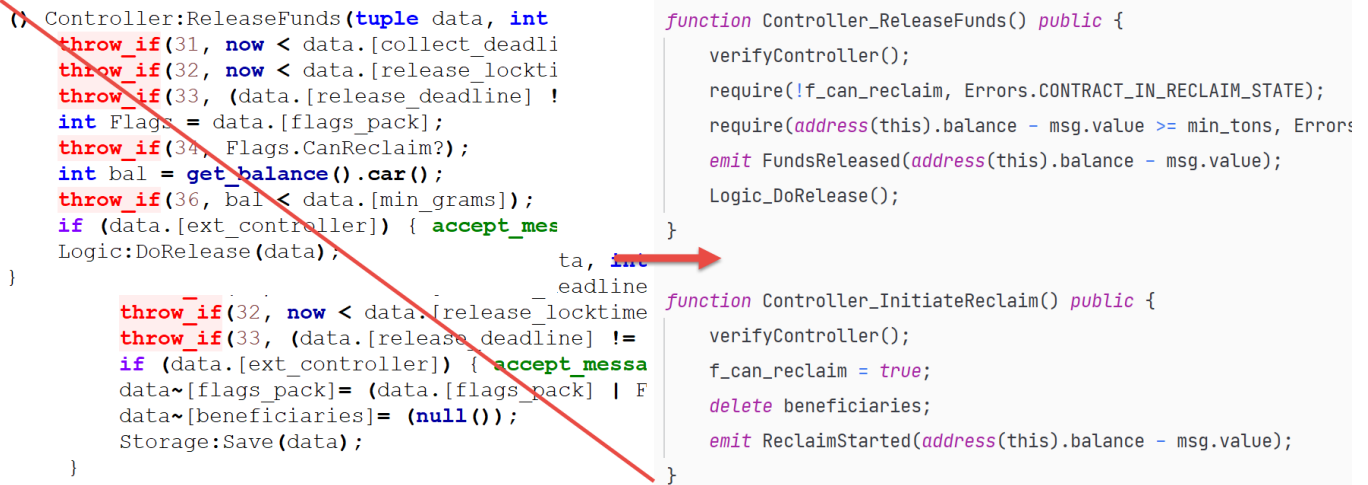
function getInformation() public view returns (
    Constraints constraints, Flags flags,
    bool canReclaim, bool destroyed,
    Beneficiary[] beneficiariesList, address ultimateBeneficiary,
    mapping(address => uint64) investorsMap,
    address controller, uint8 controllerType
) {
    constraints.minTons = min_tons;
    constraints.maxTons = max_tons;
}

```

Having to explicitly walk through all that stuff may be a good discipline and making sure that everything that needs to be presented is here, and useless stuff is not.



As the function switching and routing is now done by the compiler, now it is possible to just make functions public and they can be called from anywhere instead of manually routing them from internal or external message handler:



The diagram illustrates the simplification of function calls. On the left, a complex function `Controller:ReleaseFunds` is shown with multiple `throw_if` statements and a `Logic:DoRelease` call. A red arrow points from this function to a simplified version on the right. The simplified version consists of two public functions: `Controller_ReleaseFunds` and `Controller_InitiateReclaim`, both of which are straightforward and do not contain the complex routing logic.

```

() Controller:ReleaseFunds(tuple data, int
throw_if(31, now < data.[collect_deadli
throw_if(32, now < data.[release_lockti
throw_if(33, (data.[release_deadline] !=
int Flags = data.[flags_pack];
throw_if(34, Flags.CanReclaim?);
int bal = get_balance().car();
throw_if(36, bal < data.[min_grams]);
if (data.[ext_controller]) { accept_mes
Logic:DoRelease(data);
ta, int
deadline
throw_if(32, now < data.[release_locktime
throw_if(33, (data.[release_deadline] !=
if (data.[ext_controller]) { accept_messa
data~[flags_pack] = (data.[flags_pack] | F
data~[beneficiaries] = (null());
Storage:Save(data);
}

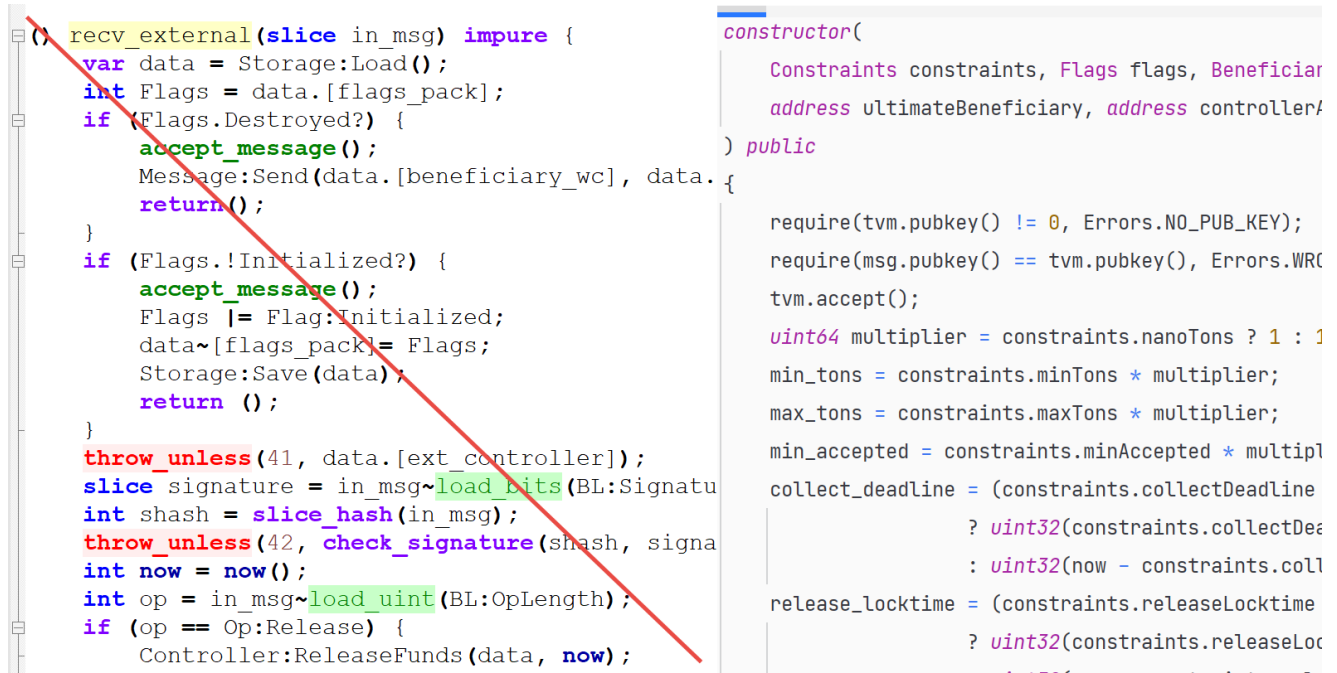
function Controller_ReleaseFunds() public {
    verifyController();
    require(!f_can_reclaim, Errors.CONTRACT_IN_RECLAIM_STATE);
    require(address(this).balance - msg.value >= min_tons, Errors.
    emit FundsReleased(address(this).balance - msg.value);
    Logic_DoRelease();
}

function Controller_InitiateReclaim() public {
    verifyController();
    f_can_reclaim = true;
    delete beneficiaries;
    emit ReclaimStarted(address(this).balance - msg.value);
}

```

It is worth mentioning that such unity allows to call those functions both internally or externally, therefore additional verification logic had to be put in place to handle those situations differently, but once again, it was surprisingly comfortable to think about the code logic rather than worrying how FunC would interpret my input. It may be also noticeable on the last picture that *event emitting* was added – there was no kind of thing like that in FunC and it facilitates more great possibilities in future.

As for the external message handler and constructor, here it is actually a huge tradeoff. On the one hand, in FunC + Fift I did not have to write a heavy constructor and the contract was deployed preinitialized to blockchain. However, still, I had to correctly handle deploy external message and do replay protection by myself. On the other hand, in Solidity I had to manually install all variables from constructor parameters into contract data, but at least that was a little less difficult than doing replay protection for external messages, verifying signatures and routing methods.



The diagram illustrates the simplification of the external message handler and constructor. On the left, a complex `recv_external` function is shown with multiple `if` statements, `accept_message` calls, and `throw_unless` statements. A red arrow points from this function to a simplified version on the right. The simplified version consists of a single `constructor` function that initializes variables and sets constraints.

```

() recv_external(slice in_msg) impure {
    var data = Storage:Load();
    int Flags = data.[flags_pack];
    if (Flags.Destroyed?) {
        accept_message();
        Message:Send(data.[beneficiary_wc], data.
        return();
    }
    if (Flags.!Initialized?) {
        accept_message();
        Flags |= Flag:Initialized;
        data~[flags_pack] = Flags;
        Storage:Save(data);
        return();
    }
    throw_unless(41, data.[ext_controller]);
    slice signature = in_msg~load_bits(BL:Signatu
    int shash = slice_hash(in_msg);
    throw_unless(42, check_signature(shash, signa
    int now = now();
    int op = in_msg~load_uint(BL:OpLength);
    if (op == Op:Release) {
        Controller:ReleaseFunds(data, now);
    }

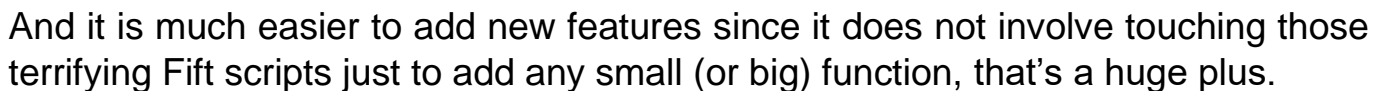
constructor(
    Constraints constraints, Flags flags, Beneficiar
    address ultimateBeneficiary, address controller/
) public

    require(tvm.pubkey() != 0, Errors.NO_PUB_KEY);
    require(msg.pubkey() == tvn.pubkey(), Errors.WRC
    tvn.accept();

    uint64 multiplier = constraints.nanoTons ? 1 : 1
    min_tons = constraints.minTons * multiplier;
    max_tons = constraints.maxTons * multiplier;
    min_accepted = constraints.minAccepted * multipl
    collect_deadline = (constraints.collectDeadline
        ? uint32(constraints.collectDee
        : uint32(now - constraints.coll
    release_locktime = (constraints.releaseLocktime
        ? uint32(constraints.releaseLoc
        : uint32(now - constraints.rele

```



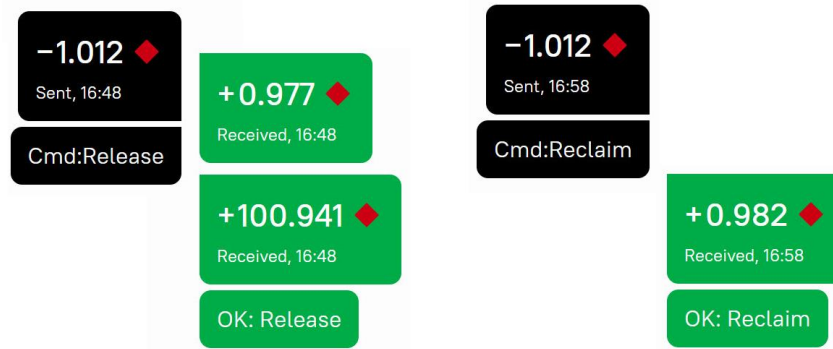


Finally, rewriting the internal message logic (that is even bigger than funds release logic) proven to be more or less the same as original one (no screenshot here because the function is *really* tall).

However, the addition of *strings* to Solidity proved to be so useful that I have made it so that the bot replies to incoming messages with meaningful English text responses! That makes it much easier for users to understand if everything went OK, or that some problem occurred, and even immediately understand the problem without diving into code and explorer stuff.



Even the much more interesting addition, that I will showcase at Demonstration in the next chapter, is that the Controller, if it is defined as a smart contract address, can send a simple message with specific text right in Surf to the smart contract, and it will actually trigger the corresponding action! Of course, this may be a temporary measure until DeBots arrive, but, nevertheless, it looks very cool and convenient. A small preview of how it looks like in Surf:



I think that I have surfaced all the points of interest that were worth mentioning relating to the integration effort, FunC, Fift and Solidity code and the important supporting environment.

You can get acquainted with the code yourself, if you wish, the link to the code repository is at [the top of this chapter](#) (beginning of page 10).

An example of how the smart contract actually worked on the Dev Network using it mostly from the Surf and some instructions of how you could do it yourself are showcased in the next chapter.

## 4 Demonstration

In this chapter I first outline several exemplary use cases and configurations of this contract and show how that worked out from Surf perspective. Then I provide a short manual of how you can configure and deploy such contract yourself.

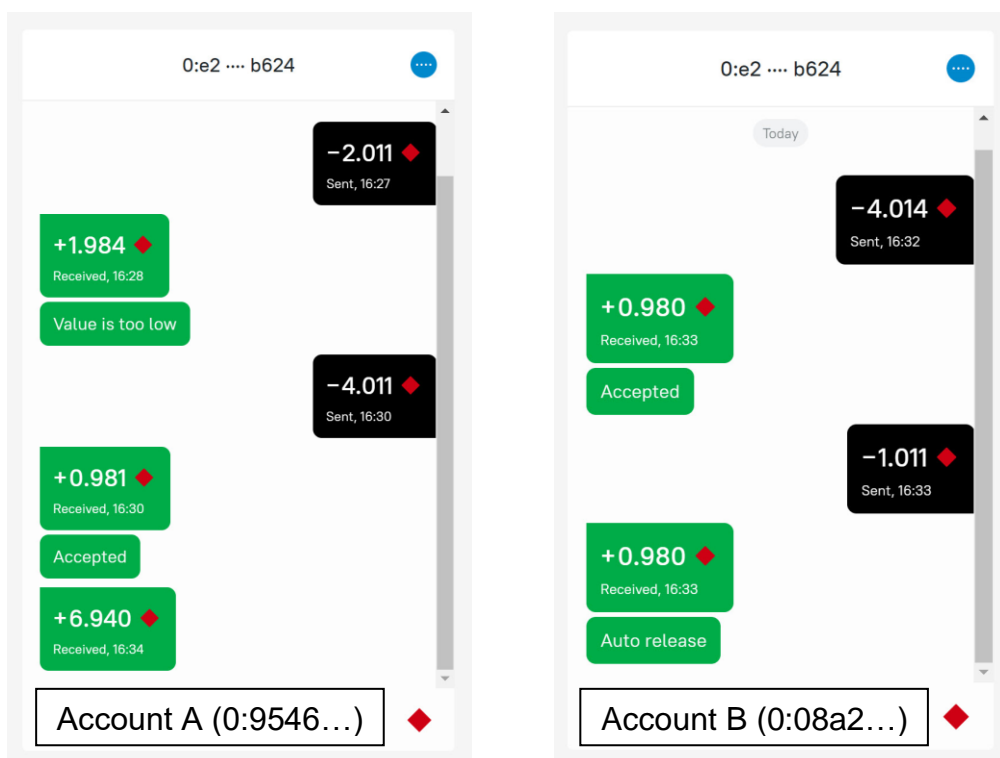
### 4.1 Automatic crowdfunding (simplified)

This is a simplified configuration, a proof of concept of automatic crowdfunding algorithm: that is, once the target is reached, funds are automatically released.

```
{ "constraints": {
  "minTons": 5,      "maxTons": 10,      "minAccepted": 3,      "nanoTons": false,
  "collectDeadline": 0,      "releaseLocktime": 0,      "releaseDeadline": 1
},
"flags": {      "autoRelease": true,      "continuousColl": false  },
"beneficiariesList": [],
"ultimateBeneficiary": "0:954688c088881241...",
"controllerAddr": "" }
```

That is, the contract would send funds to ultimate beneficiary (0:9546...) as soon as minimum amount of Tons is collected (5) but no more than maximum amount (10, excess would be returned to sender), each transfer shall be no less than 3 Ton. The "as soon as" is provided by setting release deadline to 1 (far in the past) and enabling auto release – therefore reaching the objective immediately sends funds.

On the following screenshots you can see Account A trying to deposit 1 Tons, but that amount is too low (1 is returned and used for gas fees). Afterwards, 3 Tons are successfully deposited. Then, Account B deposits 3 more Tons and then sends a simple 1 Ton message to the contract that initiates funds release, and then we can observe ultimate beneficiary (Account A) receiving all contract's collected funds.





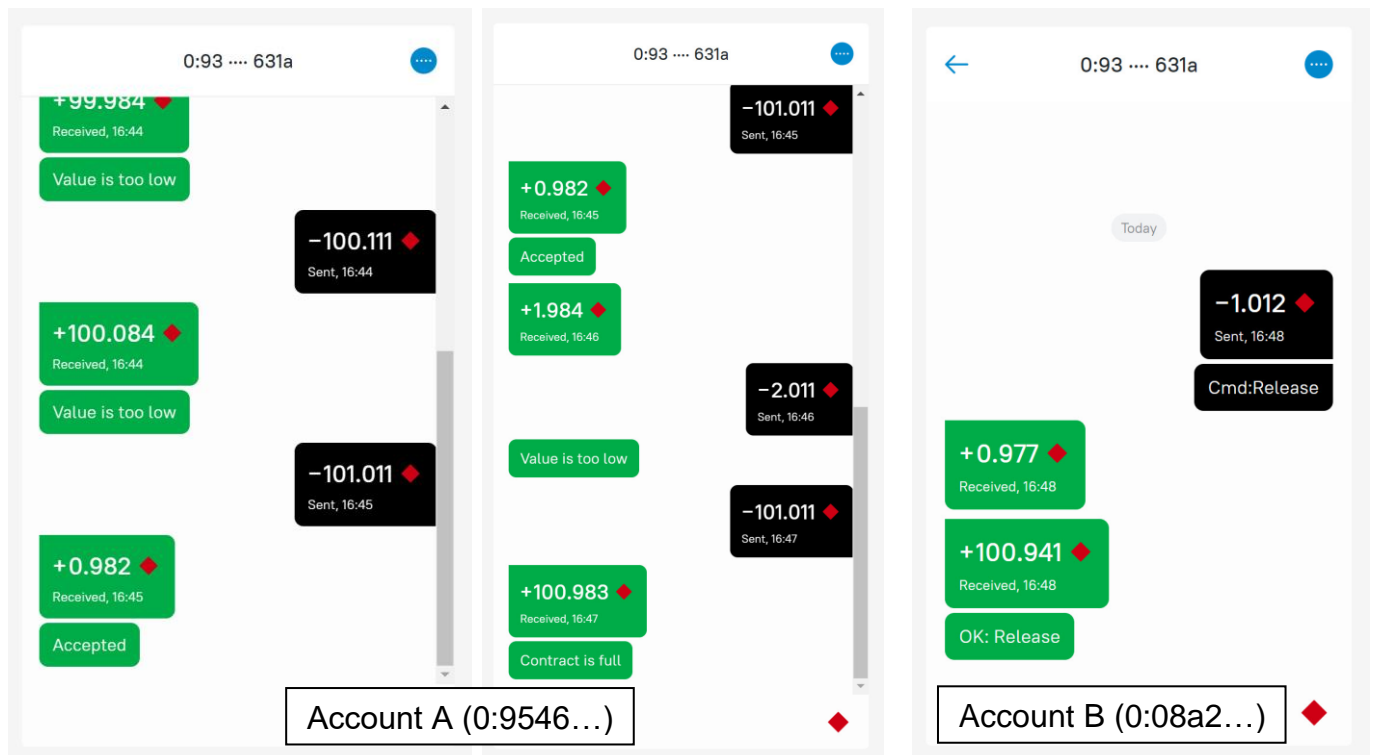
## 4.2 Custody

### 4.2.1 Successful funds release

In this simplified example, funds are held in custody on this contract, and are released to destination as soon as they confirm it. That is, the destination is the controller in this situation. This can be easily turned into escrow by setting controller to a third-party address. Of course, for real-life use you shall set release deadline to a reasonable time after which sender will be able to reclaim their funds.

```
{ "constraints": {
  "minTons": 100,      "maxTons": 101,      "minAccepted": 100,      "nanoTons": false,
  "collectDeadline": 0, "releaseLocktime": 0, "releaseDeadline": 0
},
"flags": {
  "autoRelease": false, "continuousColl": false
},
"beneficiariesList": [],
"ultimateBeneficiary": "0:08a2993aa5c9d938...",
"controllerAddr": "0:08a2993aa5c9d938..."
}
```

The major point here is that amount of collected tons is almost fixed (minimum is 100 and maximum is 101 with accepting no less than 100) and that controller address is the same as ultimate beneficiary. That is, the beneficiary have to ask the contract by sending a special message or a simple message with text Cmd:Release to obtain the funds. And they (Account B) do it in following conversation:



At these screenshots we can see how the contract rejects too small transactions from Account A, then accepts the 100 Ton message and later (at the second one) rejects another message since it would overflow max ton limit of contract.

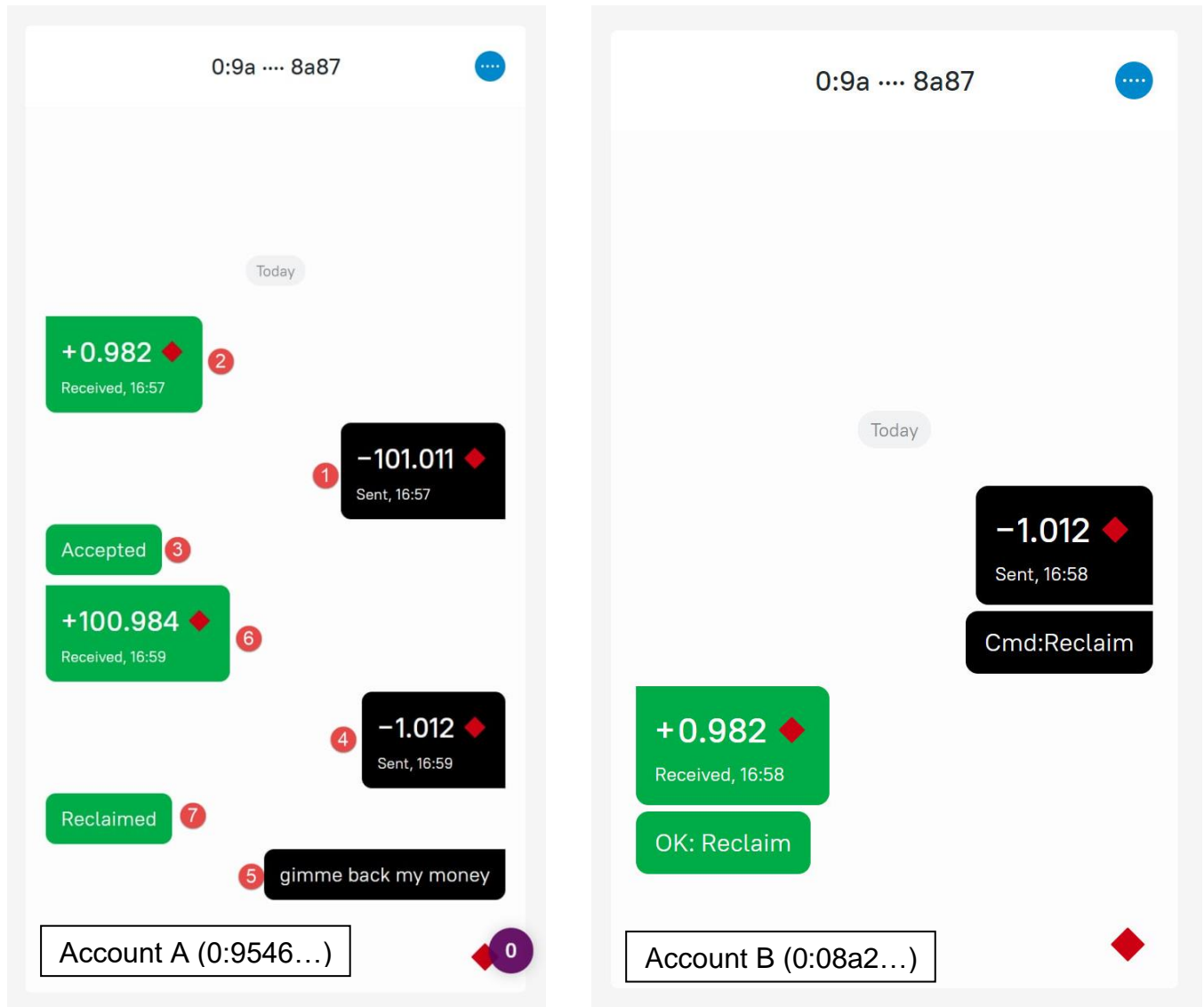
After that, Account B sends a simple small 1 Ton message with test Cmd:Release to the contract that results in the contract releasing to them the held sum. If release

deadline is set if they do not do it in time, Account A would be able to send empty 1 Ton message to the contract and get their funds back.

#### 4.2.2 Returning funds to sender

The following conversation demonstrates possibility of Account A reclaiming their deposit under some conditions. In this example, Account B willingly asks contract to allow the reclaim, but in real-life, such situation might occur if a preset release deadline is set and it already happened.

*The configuration for this example is identical to the previous one (4.2.1).*



I would like to notice that on the first screenshot the Surf actually bugged the order of messages in the conversation for some reason, but I marked with numbers the correct order. That is, Account A sent 100 Tons deposit to the smart contract, then Account B initiated reclaim mode on the contract, and afterwards, Account A sent a small message to the smart contract, that would immediately return them their funds. The behavior here is similar to the elector contract – you shall check the contract state and reclaim your funds, if possible. By using DeBots and events it may be theoretically possible to notify the user of such event in future.

### 4.3 Distribution rules

This last demonstration showcases the new percentage fund distribution rule along with the old one (remaining balance versus entire collected balance). The config:

```
{
  "constraints": {
    "minTons": 10,      "maxTons": 20,      "minAccepted": 5,      "nanoTons": false,
    "collectDeadline": 0,  "releaseLocktime": 0,  "releaseDeadline": 1
  },
  "flags": {
    "autoRelease": true,    "continuousColl": false
  },
  "beneficiariesList": [
    {
      "addr": "0:08a2...",  "value": -25000000
    },
    {
      "addr": "0:9546...",  "value": -20000000
    },
    {
      "addr": "0:08a2...",  "value": -125000000
    }
  ],
  "ultimateBeneficiary": "0:08a2...",
  "controllerAddr": ""
}
```

The config is somewhat similar to the 4.1 one, but this time beneficiaries list is filled. That is, positive value means that exactly specified amount of nanotons shall be sent to corresponding beneficiary, while negative value processing is interesting.

If the negative value is less than -100,000,000 then a specific portion of currently remaining funds would be distributed to this beneficiary. That is, for example, a value of -123,456,789 would give 23.456789% of currently remaining funds.

If the negative value is between -100,000,000 and 0, then it is classic distribution of a portion of total collected funds. That is, a value of -12,345,678 would give 12.345678% of total collected funds regardless of beneficiary position, if that amount still remains when beneficiary is credited.

So, in this example, the following actions would be carried out:

- 0:08a2... would receive 25% of totally collected funds
- 0:9546... would receive 20% of totally collected funds
- 0:08a2... would receive 25% of *remaining unsent funds*
- 0:08a2... would receive all remaining funds as ultimate beneficiary

During my demonstration, screenshots of which I will showcase on next page, upon funds distribution my contract had approximately 15.934,139,876 tons.

Now, let's calculate how it should have worked out:

- 0:08a2... should receive 25% of total, that is 3.983,534,969 tons
- 0:9546... should receive 20% of total, that is 3.186,827,975 tons
- Remain:  $15.934,139,876 - 3.983,534,969 - 3.186,827,975 = 8.763,776,932$  tons
- 0:08a2... would receive 25% of remaining, that is 2.190,944,233 tons
- 0:08a2... would receive all remaining funds: 6.573,482,699 tons

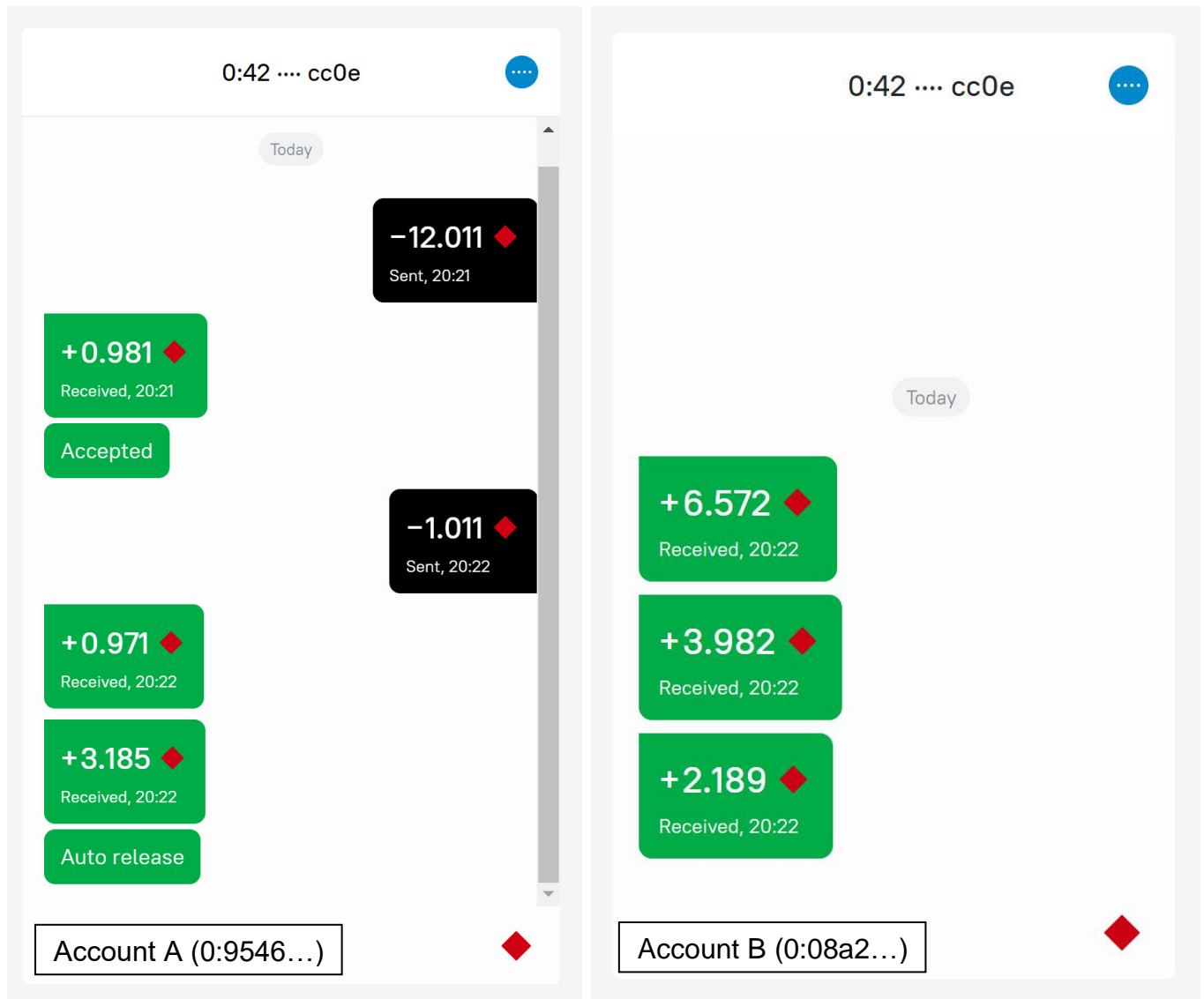
Now, lets see how it really worked out during the testing.

Actually, if not account for the fact that ordering in TON Live is a bit messed up, the digits seem to actually match the theoretical calculations pretty precisely. In order to check the actual amounts, I have exported the list to csv file and inspected it. I observed:

id	value	dst
057eb825a	3.983534969	0:08a29
9d7722a30	3.186627975	0:95468
d2a4edca6	2.190494233	0:08a29
ee580a59d	6.573482699	0:08a29

ee580a59d ... 551243a59	6.573 ♦
8:22:23 PM · Dec 15, 2020	→ to 0:08a
d2a4edca6 ... 5e286e36d	2.190 ♦
8:22:23 PM · Dec 15, 2020	→ to 0:08a
057eb825a ... 182643982	3.984 ♦
8:22:23 PM · Dec 15, 2020	→ to 0:08a
9d7722a30 ... 4bc305c66	3.187 ♦
8:22:23 PM · Dec 15, 2020	→ to 0:954

The calculations match the actual results very precisely, so everything works great. And finally, how it looked in the Surf (it mixed up transactions order like TON Live):



Everything looks great except that transactions are a little out of order... but the most important thing is that sums match, and that is great!

You can view configuration and full images used in the demo [here at the repo](#).

## 5 Deployment

Currently, to deploy the **CTSC** you have to use **tonos-cli** tool, but that will absolutely sure change in the future either after being able to deploy and use **CTSC DeBot** when the ecosystem gets ready for them, or, as a fallback solution, deploy mechanics may be integrated into **CTSC** web explorer UI.

First, you would need to install the **tonos-cli** tool (you may already have it installed, since it is needed for many tasks in the network, including multisig and depools), you can find out more at <https://github.com/tonlabs/tonos-cli>. **As of now the binaries are only provided for Linux**, so if you use Windows or MacOS you would need to build the tool from sources (or use WSL if you use Windows 10).

Having installed the tool, you may now deploy the contract. In order to do that, you should download files from the <https://github.com/Skydev0h/tg-contest-freeton> repository (at the very least you would need **args.json**, **CondTran.tvc** and **CondTran.abi.json** files). You may use git tools or just click the green Code button and select Download ZIP. Then you need to extract the archive (if you downloaded it), open terminal (console) and navigate to the downloaded or extracted files dir.

First, you should open the **args.json** file and modify it to suit your requirements. The template file would look like this (I have added description of entries):

```
{
  "constraints": {
    "minTons": 0, // Minimum required amount of Tons to be able to release them
    "maxTons": 0, // Maximum possible amount of Tons to be collected
    "minAccepted": 0, // Minimum required amount of Tons in one transaction
    "nanoTons": false, // true to specify values above in nanotons, false for tons
    "collectDeadline": 0, // minTons shall be collected by this time
    "releaseLocktime": 0, // Controller cannot release or reclaim before this time
    "releaseDeadline": 0 // A decision would be carried out automatically after this time
  }, // the numbers above shall be either unixtime or be minus offset in seconds from now
  "flags": {
    "autoRelease": false, // false to reclaim after release deadline or true to release
    "continuousColl": false // allow collecting funds after collect deadline or not
  },
  "beneficiariesList": [
    { // this element can occur multiple times
      "addr": "0:0000000000000000000000000000000000000000000000000000000000000000",
      "value": 0 // positive value for fixed amount of nanograms, negative for percentages
    } // -1 to -100000000 is 1/100000000th of total sum, -100000001 to -200000000 for partial
  ], // ultimate beneficiary would receive all remaining funds after beneficiaries above
  "ultimateBeneficiary": "0:0000000000000000000000000000000000000000000000000000000000000000",
  "controllerAddr": "" // address of controller who can decide to reclaim or release funds if
} // conditions are met or empty for no controller
```

Then after editing and saving **args.json** file you can start deploying the contract. But before that you need to choose on which network you would like to operate. You can select **net.ton.dev** for testing with rubies (Rubynet), or **main.ton.dev** to work with real TON Crystals. To select the network execute one of the commands:

```
tonos-cli config --url https://main.ton.dev
tonos-cli config --url https://net.ton.dev
```

Then you need to generate a keypair for contract deployment using the command:

```
tonos-cli genaddr CondTran.tvc CondTran.abi.json --genkey CondTran.keys.json
```

Afterwards you need to send some tons to the address, that you can observe in the output of the command. For simple deployments 1 tons may be enough, but if many beneficiaries are set you may need 5 tons or more.

You can recall the address by issuing the following command afterwards:

```
tonos-cli genaddr CondTran.tvc CondTran.abi.json --setkey CondTran.keys.json
```

Then you should use the following command to verify that tons have arrived:

```
tonos-cli account 0:...
```

(You have to enter the address called "Raw address:" in command output)

The account should be found (no error) and have positive balance.

Then, finally, you can deploy the contract. The issue here is that the provided command substitution was tested only on Linux and WSL. It may work on MacOS, and most certainly won't yet work on Windows. The command is (type on one line!):

```
tonos-cli deploy --abi CondTran.abi.json --sign CondTran.keys.json  
CondTran.tvc "$(cat args.json)"
```

If everything works out correctly you will see message that contract was deployed.

To make sure everything worked out correctly and to later inspect state of the contract you can use `getInformation` method in the following manner:

```
tonos-cli run --abi CondTran.abi.json "0:..." getInformation {}
```

(Obviously, you need to replace `0:...` with address of your contract)

At any time, you can call `getReleaseable` method to emulate ton distribution as if the contract would be released with current balance:

```
tonos-cli run --abi CondTran.abi.json "0:..." getReleaseable {}
```

It would return how many tons would be distributed to beneficiaries (`toBen`), ultimate beneficiary (`toUltBen`) and which beneficiaries won't receive their part because their configuration is invalid (`invalid`).

If needed to emulate distribution with another balance value, you can call `getReleaseableEmulated` method and supply to it balance parameter specifying how many nanotons you intend to be distributed:

```
tonos-cli run --abi CondTran.abi.json "0:..." getReleaseableEmulated '{"balance": "..."}'
```

As of now to interact with the contract you can use Surf. Some special interaction might be required by Controller, but they can also use Surf to send control messages. If possible, in order to release funds, they should write a message with comment **Cmd:Release** to the contract. In order to initiate reclaim, they should use the comment **Cmd:Reclaim** and attach it to the message. Messages should be no lower than 1 ton to pay for some initial gas, almost all of this sum would return.

This concludes the short deployment manual. If you want, you can go to [documentation section of the repository](#) to read more on the contract and different deployment parameters and scenarios.

The **Overview** section provides information about **CTSC** in general and some example use cases without much technical details.

The **Deployment** section explains deployment far more detailed than this short section of this document. It also contains **Examples** page that provides exemplary args.json that can be used as a base for your own **CTSC** configurations.

Finally, **Usage and interaction** section explains verifying state of contract and running informational methods on it in far more detail.

I hope that you would find my contract useful and versatile, as I really enjoyed not only making it initially for Telegram Blockchain contest but rewriting it for FreeTON on Solidity, with much space to expand upon and add more and more stuff. Great!



## 6 Authorship

### 6.1 Definition of an author of this document

The sole author of this contest entry submission (withdrawal request), original Telegram contest submissions, all corresponding smart contract code, documentation, promotional website and video (for Telegram bonus contest) is Oleksandr Murzin, who has the following identity on the necessary platforms:

GitHub: **Skydev0h** (<https://github.com/Skydev0h>)

FreeTON forum: **Skydev** (<https://forum.freeton.org/u/Skydev>)

Telegram Contests platform:  **Shiny Giraffe** ([Stage 1](#), [Stage 2](#), [Bonus stage](#))

Telegram account link: **@skydev** (<https://t.me/skydev>)

E-mail address: alexhacker64 (at symbol) gmail (dot) com

### 6.2 Proof of authorship and ownership

As a proof that I have actually written this submission I would write an accompanying message on the forum, that is linked as *Discussion / Discuss on forum* in this submission page at FreeTON Governance website. The message contains basic information about the contest entry, direct link to gov submission page would be added shortly after submission is sent. Such ceremony links the identity of the author of this contest entry and use Skydev on the FreeTON forum.

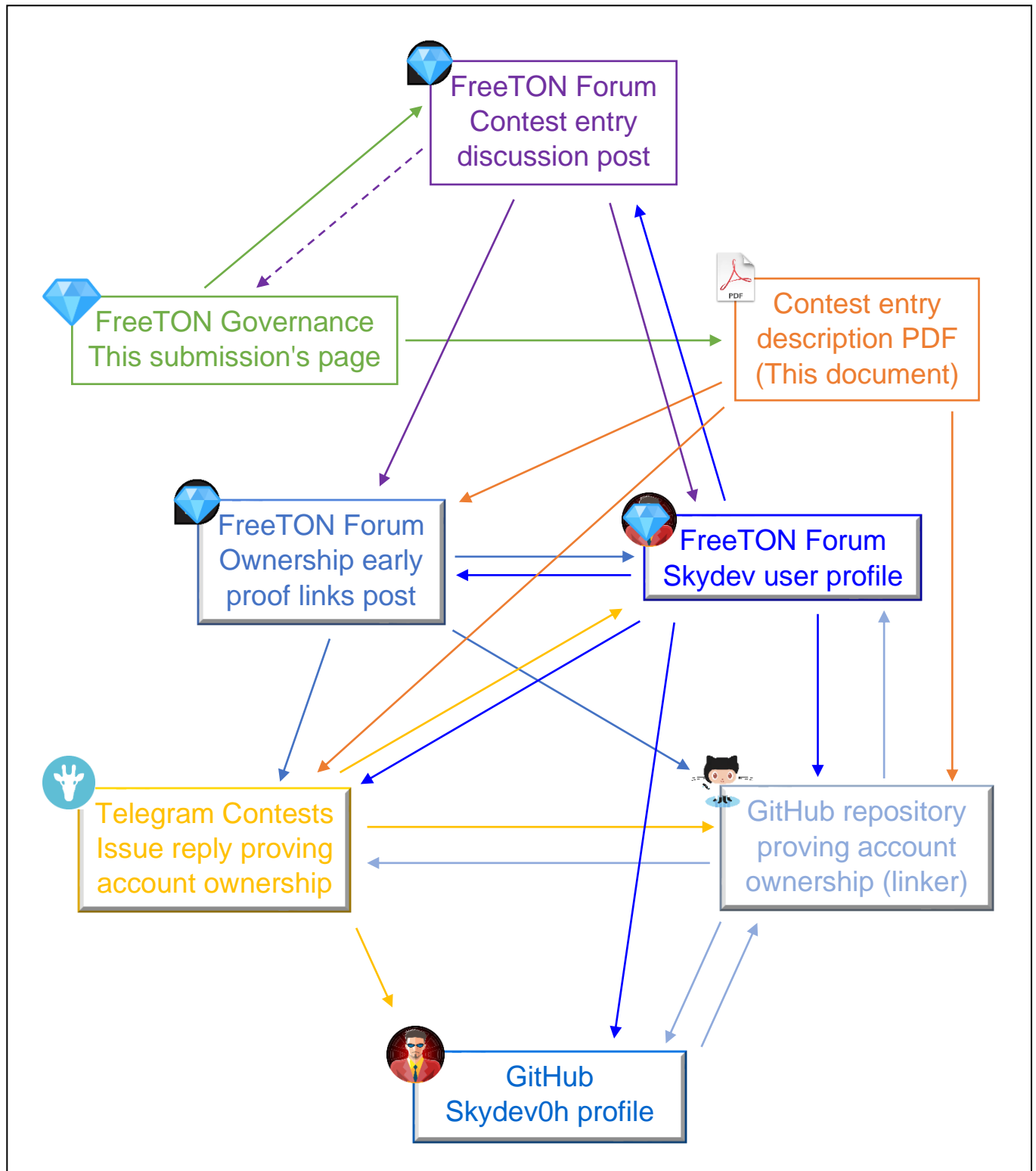
A message that I published beforehand, contains links that prove ownership of GitHub and Telegram Contests accounts: <https://forum.freeton.org/t/1455/28>

Reciting the message, the following link proves ownership over my GitHub account: <https://github.com/Skydev0h/FreeTON-Contest-Proof>. It should be noted, that after sending the submission I will commit a direct link to gov submission page to that repository, that would additionally directly link identity of the GitHub repository owner to this contest entry, Skydev0h at GitHub, to be precise.

To confirm ownership of Telegram Contests account, I have added a reply to an issue opened by judge in a contest entry because adding comments is not possible: <https://contest.com/blockchain-2-bonus/entry1357#issue10998>, scroll down a bit.

### 6.3 The proof relation graph

To make understanding of the authorship proofs relations easier, I have made a graph, that clearly indicates how all those elements relate to each other:



The table provided above not only outlines links between proofs and different profiles but also lets you easily access them by clicking button-like boxes (all boxes except for top three are clickable). This illustration should make it easier to observe and be sure in actual ownership of me over these profiles and pages, most importantly, Telegram Contests and GitHub one, as required per contest rules.