```
/-------------------------------------------------------------------\
| Created for: Telegram (Open Network) Blockchain Contest           |
|     Task -2: Documentation for the contest solution entries and more   |
>-------------------------------------------------------------------<
| Author: Oleksandr Murzin (tg: @skydev / em: alexhacker64@gmail.com)    |
|         October 2019                                              |
\-------------------------------------------------------------------/
```

TON Smart-contract implementation contest

User manual for the participation entry

Oleksandr Murzin

October 15, 2019

Introduction

This document contains a description of the TON contest participation package, including, but not limited to, user manual for usage of Fift scripts, automators manual for usage of robot-mode of scripts, programmer's manual with implementation details and data structure of different parts of the package, and some thoughts about arising problems while planning and implementing the smart contracts and user interface scripts.

The sections of the document are sorted in order of tasks given in the contest tasks definition list, with exception to special sections concerning special folders not related to some specific task.

Contents

0. Definition and status of the tasks

The contest is about implementing some of the outlined in text files in the ***0-Definition*** folder smart contracts while adhering to the requirements and following smart-contract guidelines as closely as possible.

Out of all the tasks, here is the status of their implementation:

#	Task name	Proto -type	FunC Code	User Intrf	Local tests	Online tests
1	Multi-signature k/n wallet	half	-	-	-	-
2	*Automatic DNS resolver*	okay	675 L	4 scr	44 ok	17 ok
3	*Manual DNS resolver*	okay	391 L	6 scr	43 ok	23 ok
4	Synchr. payment channel	miss	-	-	-	-
5	Asynch. payment channel	miss	-	-	-	-

It should be noted that root folder contains some of the library .fif and .fc files for safety and easier usability. **It is also expected that *fift* and *func* executables reside in the system binary path, otherwise you need to specify full path all the times, when starting of those commands is required.**

Sadly, I did not manage to complete all tasks in time because of lack of time (I had only about a week to do all the stuff), so it is very unfortunate that there is no time to implement very interseting contraptions such as payment channels. Entire last day (October 15) was spent making this documentation.

Therefore, sections 1, 4 and 5 have been removed from the documentation.

Sections 8 and 9 have such numbers to keep in sync with folder names.

2. Automatic DNS resolver

2.1 Introduction

This smart contract provides an automatic TON DNS resolver service, that allows registering (sub)domains in it upon receiving of specially partitioned message, of course containing the domain name and data. Moreover, the contract allows subsequent modification of that data by domain's owner, if defined (category -2), and prolongation of the domain (which also requires sending from "owner" address, if defined at all).

2.2 Smart contract code

The contract code itself is contained in ***adns-logic.fc*** file, which, if required, can be re-compiled to ***adns-logic.fif*** file by using the compiler:

```
func adns-logic.fc -oadns-logic.fif
```

The fif file is packed in the package, so the recompilation, overall, **should not** be really necessary.

2.3 Creating and deploying the contract

To create the smart-contract, first, you need to create a bag-of-cells (BoC) file containing required special message that will deploy the contract to the TON blockchain. To do that, you need to use ***adns-create.fif*** script:

```
fift -I.. -s adns-create.fif <label> <wch-id> <period>
             <pay-per-reg> <pp-cell> <pp-bit> [-r or -R]
```

If you execute the script without any script parameters specified, it will display a helpful usage information concerning itself. In general, you **have to** specify the following parameters during creation:

label – is used as a base name for saving the **label**.adr (contract address) and **label**-init.boc (initialization message), also used for address diversification (so as to diversify the address hash of the contract)

wch-id – identifies workchain, where the contract is to be deployed, you can use **0** for the **basechain (recommended)** or **-1** for the **masterchain (which is much more expensive,** but required in some special cases**)**

period – defines for how long domain will stay before expiration, can be defined in seconds (like **12345**), or using special period suffix (**y** for years

(365 **d**), m for months (31 **d**), w for weeks (7 **d**), and finally, **d** for days (86400 seconds)). You can use decimals with suffix (like **1.5d** = **129600**)

pay-per-reg – declares how much Gram has to be additionaly paid in order to register a domain (or to prolong it). *Please note, that for all prices they are specified in Grams (10^9 nanograms), but you can explicitly set the price in nanograms by suffixing it with **n** for ex. **100n**.*

pp-cell – price per each cell of user data for registration or data change.

pp-bit – price per each bit of user data for registration or data change.

Optional* -r or -R** – activates special ***robot mode, which turns output into much more easily parseable mode. **-R** flag prevents writing to any files, while **-r** just changes output format and does not prohibit file saves.

For example, to create a resolver labelled **myres** inside basechain, which registers domain for **one year** and requires GR$**0.5** for registration **365000** nGR per cell and **365** nGR per bit, one could issue following:

```
fift -I.. -s adns-create.fif myres 0 1y 0.5 365000n 365n
```

Robot mode outputs the following variables after creation:
raw_address, init_address, send_address – different forms of created contract's address (raw, unbouncable and bouncable).
workchain_id – shows where contract will be deployed
label – the user-entered label for the contract
period – period of domain registration/prolong, in seconds
ppr, ppc, ppb – various prices (per each registration, cell, bit)
init_query – contains serialized BoC-type query in hex form
init_query_len – shows the length of **init_query**
boc_file – contains file name where BoC was saved (if not **-R**)

Afterwards, to finish the contract creation, some amount of grams has to be sent to the contract, and the contract has to be sent to blockchain using the generated boc file, for ex. using ***lite-client*** with ***sendfile*** functionality.

You can check status of the contract using ***last*** and ***getaccount*** methods.

After deploying the contract, you (and anyone else) can use user-interface scripts (***adns-register.fif, adns-prolong.fif*** and ***adns-modify.fif***) to prepare messages for the smart contract. Usage of those scripts will be outlined in the next subsections on next pages.

2.4 Registering a domain

Having obtained the address of automatic DNS registrator contract (by deploying it yourself or by finding existing contract) you can send some messages to it to perform some actions. The first thing that you should want to do is to register a domain by sending a special message to contract. To obtain this message in a BoC format you can use ***adns-register.fif***:

```
fift -I.. -s adns-register.fif <query-id> <domain> <-r/-R/->
                                    ..<.. <category> <value> ..>..
```

If you execute the script without any script parameters specified, it will display a helpful usage information concerning itself. In general, you **have to** specify the following parameters during creation:

query-id – query-id parameter of the message, defines which query-id will be in the response, may be useful if sending from complex wallet smart contract which can handle responses, otherwise you can put 0 or 1

domain – domain name which you want to register, ***may be prefixed***

-r or -R or - - required flag that indicates output mode (**-** means standard output, **-r** – robot mode, **-R** – full robot mode (without file writes))

Next variables come in pair and can be repeated as many times as needed:

> **category** – numeric category to which will be assigned following value, can be any number from **-32768 to 32767 except 0**, note that some category values are ***special***:
>
> > **-1** category holds address of "next resolver contract": if subdomain of this domain, that does not exist, will be looked for, this address will be returned instead of original requested category
> >
> > **-2** defines **owner address** of the domain, **make sure** to specify it and that it equals to **address of smart-contract from which you can send messages** to the resolver, otherwise you **would not be able to modify** domain information or even **prolong** it!
>
> **value** – contains the value that will be assigned to the aforementioned category. It may contain simple text, binary data (***using prefixes***) or contract address in any form (**wch:hex or base64 or base64url**).

It has been noted that some variables may be ***prefixed*** or ***use prefixes*** to do special processing of input text. These prefixes allow to decode binary

data in hexadecimal form or to load input contents from file. They are F:, X: and H:, and are described as following:

F: prefix means that (possibly binary) contents will be loaded from the specified file and used instead of prefixed string. Particularly useful in this case to, for example, define -2 category entry as *-2 "F:new-wallet.addr"* to do not write the address by hand and load it from the file.

X: or **H:** prefix allows to enter arbitrary binary data by writing it as hexadecimal string, for example *"X:1234567890ABCDEF"*.

Robot mode outputs the following variables after creation:
type – type of executing script, always **regd** for this script
domain_hex – hexadecimal form of entered domain
domain – text form (if possible) or hex form of entered domain
body – contents of BoC message file with length **body_len** in hex form
dict_cell_hash – hash of representation of dictionary (table) built from provided categories and values, for debugging runmethod in lite-client
dict_size – number of entires in domain category table, which follows:
- **cat_*i*** – category number of table entry
- **value_*i*** – decoded addr or text form or hex form of entry value
- **value_hex_*i*** – hexadecimal form of entry value

boc_file – contains file name where BoC was saved (if not **-R**)

For example, if you want to register a domain TestDom, with query id 1, and the domain will contain following entries (with standard output):

-2 (*owner*)	"F:new-wallet.addr"	*Load data from file new-wallet.addr*
1	TextValue	*Specified text-value as it is*
2	H:0102AABB	*Binary value 01 02 AA BB*

The following command is to be used:

```
fift -I.. -s adns-register.fif 1 TestDom -
        -2 "F:new-wallet.addr" 1 TextValue 2 "H:0102AABB"
```

The resulting **body-adns-reg-*TestDom*.boc** file has to be embedded into an external message, to be sent to the registrator smart contract from another smart contract. Typical use case is to embed it into signed wallet external message by using some *wallet.fif*-like script and providing **-B** option to set the contents of the message to the created file. **Make sure to provide enough Grams** with the message, at least ***calculated price* plus 1 Gram** to pay for message forwarding and processing. The unused

extra above the price will be returned. Refer to *explanation of getter methods* concerning determining the ***calculated price*** for registration.

2.5 Prolonging domain registration

After registration the domain is active for a predefined period, afterwards it expires and would not be resolved (if time is provided while resolving) and will be soon garbage collected and deleted with other user's operations. Expired domain is not considered existing at all, so can be registered as a free one, and cannot be prolonged or modified, so **make sure to prolong domain in advance** – there is no limit on how many times you can prolong the domain. In order to prolong the domain and prevent it from expiring you can form prolong message BoC with ***adns-prolong.fif***:

```
fift -I.. -s adns-prolong.fif <query-id> <domain> [-r or -R]
```

If you execute the script without any script parameters specified, it will display a helpful usage information concerning itself. In general, you **have to** specify the following parameters during creation:

query-id – query-id parameter of the message, defines which query-id will be in the response, may be useful if sending from complex wallet smart contract which can handle responses, otherwise you can put 0 or 1

domain – domain name which you want to prolong, ***may be prefixed***

Optional* -r or -R** – activates special ***robot mode, which turns output into much more easily parseable mode. **-R** flag prevents writing to any files, while **-r** just changes output format and does not prohibit file saves.

Concerning usage of ***prefixed data*** refer to *2.4 Registering a domain*.

Robot mode outputs the following variables after creation:
type – type of executing script, always **prol** for this script
domain_hex – hexadecimal form of entered domain
domain – text form (if possible) or hex form of entered domain
body – contents of BoC message file with length **body_len** in hex form
boc_file – contains file name where BoC was saved (if not **-R**)

For example, to prolong domain **TestDom** with query id 1 one can use:

```
fift -I.. -s adns-prolong.fif 1 TestDom
```

Concerning usage of resulting file **body-adns-prol-*TestDom*.boc** please refer to the ending of *section 2.4*. Price is the **same** as for registration.

Please be aware that both prolonging a domain or modifying it's data requires sending the message from a specific smart-contract address that is **stored in category -2 of domain data**. If there is no access to that specific address, or information is not in correct format, or there is no category -2 entry at all **it is impossible** to prolong or modify it.

2.6 Modifying domain information

If it is necessary to modify data stored in domain, it is possible to send a special **modification message** to the contract, that is *very* similar to registration message, containing new value table for domain. In order to create such message BoC you can use ***adns-modify.fif*** script:

```
fift -I.. -s adns-modify.fif <query-id> <domain> <-r/-R/->
                              ..<.. <category> <value> ..>..
```

If you execute the script without any script parameters specified, it will display a helpful usage information concerning itself. Parameters and behaviour of script is **exactly the same** as the *for registering a domain*, so please consult the *section 2.4* of the document.

The only changes are that the resulting BoC file name is now **body-adns-mod-*DomainName*.boc** instead of **-reg-** infix, and that the **type** result in robot mode is now **modd** instead of **regd**.

So, to modify the domain provided in domain registration script example to that data, you would issue following command (notice script name):

```
fift -I.. -s adns-modify.fif 1 TestDom -
        -2 "F:new-wallet.addr" 1 TextValue 2 "H:0102AABB"
```

Pay very great attention when modifying domain data, because the domain table will be replaced in it's entirety, including the **-2 (owner)** value! If you do not provide the correct **-2 (owner)** value or fail to provide it at all you will not be able to prolong or modify domain anymore!

Therefore, the script will warn you if owner entry is missing from the table.

The ***calculated price*** for modification includes only the cost of data storage, but does not include ***fixed registration cost***, therefore it is cheaper than registering a new domain, but still, requires payment for data. Also, do not forget to put **1 Gram** above the calculated price to pay for message forwarding and processing fees.

2.7 Explanation of getter methods

As per TON DNS resolver and contest specification, the contract of course supports **dnsresolve (slice, int → (int, cell)): 123660** method, that resolves provided **domain name (slice)** with **category number (int)** into an **domain category value or dictionary (cell)** and **number of bits matched in domain name (int)**. The method allows for both full and partial matches of domain name prefix if provided multiple subdomain segments in input. For example, provided the name "org∅telegram∅" where ∅ is zero-code character, the resolver may return (104, *requested category val*) if the exact match is found, or (32, *-1 category value of org∅*) if the org∅telegram∅ does not exist, but org∅ does, or (0, Null) if neither are found.

Note, that due to special implementation of name storage in prefix tree both org∅ and org∅telegram∅ may coexist in same resolver at the same time, that means that, for example, org∅telegram∅ will resolve exactly, but org∅somenotexistingdomain∅ will resolve to -1 of org∅. Another thing to note that it is not necessary to put ∅ at end of **input domain name**, it will be appended automatically if missing.

The **dnsresolve** method attempts to extract current time from **c7** block, if it is available in environment, however, sometimes it is not available, for example when using **runmethod** of **lite-client**. In that case, zero time is assumed, so all stored domains are considered not expired (because it is impossible to get current time). In that case, you can call extended method **dnsresolvex(slice, int, int → (int, cell)): 81618** which's extra **int** parameter (compared to **dnsresolve** method) allows specifying current time in unixtime format. While resolving shorter and shorter subdomain names, expired domains are considered as completely non-existing both for **dnsresolve(x)** and **getexpiration(x)** methods.

In order to obtain current expiration time in unixtime format for a specific domain you can use **getexpiration(slice → int): 110574** method, which looks up provided **domain name (slice)** and returns expiration time for found domain, or 0 if one does not exist. With the same consideration of **c7** not available, you can use **getexpirationx(slice, int → int): 87450** method which accepts extra **int** as the current time, just like **dnsresolvex**.

As the registration costs Grams, some additional get-method are implemented to be able to estimate ***calculated price*** for domain registration, modification or prolonging. The methods are as following:

getstdperiod (→ int): 67418 – returns the period, for which domain is registered or prolonged, in seconds (has no input arguments)

getppr (→ int): 109522 – returns the price (nGR) per registration

getppc (→ int): 108994 – returns the price (nGR) per each cell of data

getppb (→ int): 113123 – returns the price (nGR) per each bit of data

calcprice (slice → int): 113304 – calculates and returns the amount of nGR (nanograms) that is to be paid for **storing** specified domain data (a **slice** that is the root of category value table of the domain), is effectively **ppb * number of bits in data + ppc * number of cells in data**. This function can calculate price for **modifying** domain information.

calcregprice (slice → int): 85700 – calculates and returns the price for **registration** or **prolongation** of the domain. Is effectively a **calcprice** result plus the **ppr (price per registration)**.

2.8 Implementation details

The contract is controlled with internal messages from other smart contracts, carrying some amount of Grams, that is enough to pay the required due for performing the operation plus 1 Gram for possible message forwarding and processing (gas) fees. The contract always sends a response for valid commands (having at least 32 (op) + 64 (query_id) bits in body), for invalid responses all received amount except the forwarding and processing fees is returned, in case of operation success, operation price is deducted in addition to aforementioned fees. There are several predefined operations, which can be carried out by messages, having op set to values:

`regd`	`0x72656764`	register a new domain (name and value are refs)
`prol`	`0x70726f6c`	prolong a specified domain (name is ref)
`updd`	`0x75706464`	update existing domain (name and value are refs)
`gogc`	`0x676f6763`	trigger a garbage collection iteration manually

In response to those messages, contract will respond with a message, containing in it's body: **response code (tag) [32b], query id [64b], request operation id (op) [32b]** and remainder of message value.

Possible response message codes are following (incl. ascii | 2^{31}):

`okay`	`0xef6b6179`	**Operation completed successfully**
`----`	`0xffffffff`	Unknown query operation code
`no\0`	`0xee6f5c30`	Name must end with ∅ (for messages)
`notf`	`0xee6f7466`	Specified domain is not found or expired
`no-2`	`0xee6f2d32`	Specified domain is missing owner (-2) at all
`bad2`	`0xe2616432`	Specified domain owner record is malformed
`nown`	`0xee6f776e`	Sender is not owner of specified domain
`serr`	`0xf3657272`	Failed to save data to the internal prefix tree
`alre`	`0xe16c7265`	Domain already exists, impossible to register
`gr<p`	`0xe7723c70`	Not enough grams provided to perform operation
`ov\0`	`0xef765c30`	Too much ∅s in message (*this check is disabled*)

Each change of data stores updated information in two trees: first is primary prefix tree, that contains domain information itself, second is map of sortable numeric key with embedded expiration time to domain value.

The second dictionary with UInt64 keys allows efficient garbage colleciton of expired domains. On every request if at least 60 seconds have passed since the last garbage collection ("housekeeping"), the entry with least value is found in the dictionary, and if it has already expired, it is deleted from both dictionary and prefix tree. It may be possible that stale reference is obtained (if it was re-registered between expiration and garbage collection), then just entry from dictionary is deleted. Each 64-bit key in such dictionary consists of 32-bit expiration time as the higher dword, and part of SHA256 hash of the name as the lower dword. This approach will prevent collisions upto several million registered domain names per second, while it can be scaled by increasing the size of second part of the key at the cost of using more storage for the key. Deleting only one entry with high frequency of garbage collection (60 seconds) allows not to stress the process on one random transaction, but rather balance the garbage collection process among many different user transactions.

The primary prefix tree itself has a very interesting structure. Usually, prefix tree does not allow prefix collisions in the tree, for example, if you have org∅ in tree, it would not be possible to put org∅telegram∅ in and vice versa. This approach allows very fast lookups without loops but prevents from storing both domain and subdomain in the tree and allowing to return more specific domain if it exists, or less specific with next resolver

response if the more specific one is not found. In this contract, this problem is resolved by special key stucture: the key in prefix tree is prefixed with 7-bit value, that contains amount of ∅s in the domain name. For example, domain org∅ would be stored as [1]org∅, where [x] is stored 7-bit value x. Meanwhile, org∅telegram∅ would be stored as [2]org∅telegram∅, and org∅telegram∅ton∅ would be stored as [3]org∅telegram∅ton∅, etcetra. As the maximal length of domain name is 127 characters, 7 bits are right enough to store amount of possible ∅s in the domain name.

When performing lookup, the searches on the prefix tree are done with decreasing prefix. For example, when searching for org∅telegram∅ton∅, first lookup is attempted for value [3]org∅telegram∅ton∅, if it is not found, [2]org∅telegram∅ton∅ lookup is attempted (which may return, for example, org∅telegram∅ name), and finally [1]org∅telegram∅ton∅ is looked up (which may return org∅), if it does not work the domain is not found.

As from the security perspective, if required for the specific task, registration of such domain names with multiple ∅s can easily be **forbidden in code (enable zeros != 1 check)**, or restricted to the owner of parent domain (parent domain can be looked up and, for example, **-2** can be checked, or may be some another special category, such as -3, which will allow to some another entity to register subdomains for the specific domain or something else). Nevertheless, for the current task, domains are registered on first-come, first-serve basis, and allowing registering both org∅ and org∅telegram∅ provides more flexibility. And once more reminding that the ∅-amount check can be easily re-enabled in code by uncommenting two lines.

3. Manual DNS resolver

3.1 Introduction

This smart contract provides a manual TON DNS resolver service, that allows managing it's data (adding, deleting, replacing (sub)domain category values; adding, deleting, replacing the (sub)domains altogether, and changing the owner of contract) only by sending a special external message to the contract signed by the current owner of contract. Because gas and storage fees are deducted from the contract itself, owner should keep an eye on contract balance and refill it, when necessary, to prevent contract freezing and destruction.

3.2 Smart contract code

The contract code itself is contained in ***mdns-logic.fc*** file, which, if required, can be re-compiled to ***mdns-logic.fif*** file by using the compiler:

```
func mdns-logic.fc -omdns-logic.fif
```

The fif file is packed in the package, so the recompilation, overall, **should not** be really necessary.

3.3 Creating and deploying the contract

To create the smart-contract, first, you need to create a bag-of-cells (BoC) file containing required special message that will deploy the contract to the TON blockchain. To do that, you need to use ***mdns-create.fif*** script:

```
fift -I.. -s mdns-create.fif <wcid> <filename-base> [-r or -R]
```

If you execute the script without any script parameters specified, it will display a helpful usage information concerning itself. In general, you **have to** specify the following parameters during creation:

wcid (workchain-id) – identifies workchain, where the contract is to be deployed, you can use **0** for the **basechain (recommended)** or **-1** for the **masterchain (which is much more costly,** but sometimes required)

filename-base – specifies a filename base for following created files: private key file **◊.tpriv**, where ◊ is filename-base value, public key file **◊.tpub**, contract address ◊.addr and init query BoC file **◊-init.boc**. If any of the files exist, but **tpriv** file does not, no action will be done. If **tpriv** file exists, any of the missing 3 files will be re-generated from the tpriv file.

Optional* -r or -R** – activates special ***robot mode, which turns output into much more easily parseable mode. **-R** flag prevents writing to any files, while **-r** just changes output format and does not prohibit file saves.

For example, to create a resolver in **basechain** and save resulting files to **myres** [.tpriv, .tpub, .addr, -init.boc] one could issue the following:

```
fift -I.. -s mdns-create.fif 0 myres
```

Robot mode outputs the following variables after creation:

raw_address, init_address, send_address – different forms of created contract's address (raw, unbouncable and bouncable).
workchain_id – shows where contract will be deployed
public_key – outputs public key in hexadecimal format
private_key – outputs private key in hexadecimal format
init_query – contains serialized BoC-type query in hex form
init_query_len – shows the length of **init_query**
boc_file – contains file name where BoC was saved (if not **-R**)

Afterwards, to finish the contract creation, some amount of grams has to be put into the contract, and afterwards the contract has to be sent to blockchain using the generated .boc file with, for example, using ***lite-client*** and it's ***sendfile*** functionality.

You can check status of the contract using ***last*** and ***getaccount*** methods.

After deploying the contract you can use user-interface scripts (***mdns-add.fif, mdns-del.fif, mdns-setdom.fif, mdns-domdel.fif, mdns-chown.fif***) to prepare external messages for the smart contract. Usage of those scripts will be outlined in the next subsections of the document. Please note that **private key file .tpriv is required** to create **any** controlling message to the contract, moreover it is **strongly recommended** to keep the .addr file, because despite the fact that scripts support regenerating contract address from the private key file if it is lost, such regeneration requires presence of **mdns-logic.fif** file, and may be suddenly unreliable if new version of contract arrives, and code changes in the future. So, it is better to **keep an eye on all 4 files** after contract generation so as not to have any problems with it's later usage.

3.4 Adding or replacing domain category entry

Having deployed the smart contract you should now have **private key .tpriv** file and **address .addr** file. Using interface scripts and those two files you can perform various actions. One of such actions is adding a new entry to a domain category table. To do that, or any other action, you need to first obtain a message in BoC format, and then send it to the network. To obtain a message for adding entry use ***mdns-add.fif*** file:

```
fift -I.. -s mdns-add.fif <pkfile> <address> <seqno> <domain>
                                  <category> <value> [-r or -R]
```

If you execute the script without any script parameters specified, it will display a helpful usage information concerning itself. In general, you **have to** specify the following parameters during creation:

pkfile – a filename of **private key file**, with optional **.tpriv** extension

address – must contain target contract address, or filename of **.addr** file (with optional **.addr** extension), or **-** to try to find ***pkfile*.addr** file, or numeric value corresponding to workchain id in order to calculate address out of **.tpriv** file (requires correct version of **mdns-logic.fif**)

seqno – has to match target contract's sequence number (you can get it using getter-methods, described at *3.9 Explanation of getter methods*)

domain – contains domain name to be manipulated (***can be prefixed***)

category – has to contain numeric category of which value will be set (you can read more information about domain categories in *section 2.4*).

value – has to contain value that will be assigned to the provided category of the specified domain (more info format in *section 2.4*, ***can be pref.***).

Optional* -r or -R** – activates special ***robot mode, which turns output into much more easily parseable mode. **-R** flag prevents writing to any files, while **-r** just changes output format and does not prohibit file saves.

Concerning usage of ***prefixed data*** refer to *section 2.4*. In general, it allows to input hexadecimal binary data or load data from some file.

Robot mode outputs the following variables after creation:
type – type of executing script, always **vset** for this script
pk_file – name of the private key file used
seqno – sequence number that was provided

domain_hex – hexadecimal form of entered domain
category – provided category number to be set
value_hex – hexadecimal form of provided value that will be set
domain – text form (if possible) or hex form of entered domain
value – decoded addr or text form or hex form of entered value
query – contents of BoC query file with length **query_len** in hex form
value_cell_hash – hash of representation of value cell built from provided value, for debugging runmethod in lite-client
boc_file – contains file name where BoC was saved (if not **-R**)

For example, if you have **myres.tpriv** and **myres.addr** files that correspond to the resolver that you want to work on, and you want to add the value **TestVal** to category **1** of domain **TestDom**, and considering that current seqno of contract is **13** you should issue:

```
fift -I.. -s mdns-add.fif myres - 13 TestDom 1 TestVal
```

The resulting ***myres-13*-vset.boc** file should be directly sent to the network just like initialization message using, for example, ***lite-client*** and it's ***sendfile*** functionality. If the execution was succesful, you can observe the change of **seqno** number in the contract.

3.5 Deleting a domain category entry

Sometimes, you need to do something reverse to the thing done in previous section: delete an entry from the category table of the domain. To do that, you should use ***mdns-del.fif*** user interface script as the following:

```
fift -I.. -s mdns-del.fif <pkfile> <address> <seqno> <domain>
                                  <category> [-r or -R]
```

The parameters of this script are absolutely same with respect to previous script (see *section 3.4*) with exception that there is no **value** parameter – because the entry is being deleted, not added.

Consequently, the *robots ouput variables* are the same, with exception that **type** is now **vdel** instead of **vset**, and **value*** variables are missing.

For example, to delete domain entry that was added in previous section, you can issue the following command (lets consider **seqno** = **14**):

```
fift -I.. -s mdns-del.fif myres - 14 TestDom 1
```

The resulting ***myres-14*-vdel.boc** file should be handled just like BoC query files in two previous sections (sent to network directly).

3.6 Adding or replacing domain category table

If you need to define a domain with many category entries, or you know the category table of domain in advance, it may not make sense to add category table entries one-by-one because it is slow, and consumes much gas. Therefore, if you need to set all entries of specific domain's category table, this can be done in one query using ***mdns-setdom.fif*** script:

```
fift -I.. -s mdns-setdom.fif <pkfile> <address> <seqno>
            <domain> <-r/-R/-> ..<.. <category> <value> ..>..
```

pkfile, **address**, **seqno**, **domain** – described in *section 3.4*.

-r or -R or - - required flag that indicates output mode (**-** means standard output, **-r** – robot mode, **-R** – full robot mode (without file writes))

Next variables come in pair and can be repeated as many times as needed:

category – numeric category to which will be assigned following value, can be any number from **-32768 to 32767 except 0**, note that **-1 category is *special:*** it holds address of "next resolver contract": if subdomain of this domain, that does not exist, will be looked for, this address will be returned instead of original category.

value – contains the value that will be assigned to the aforementioned category. It may contain simple text, binary data (***using prefixes***) or contract address in any form (**wch:hex or base64 or base64url**).

Concerning usage of ***prefixed data*** refer to *2.4 Registering a domain*.

Robot mode outputs the following variables after creation:
type – type of executing script, always **dset** for this script
pk_file – name of the private key file used
seqno – sequence number that was provided
domain_hex – hexadecimal form of entered domain
domain – text form (if possible) or hex form of entered domain
query – contents of BoC query file with length **query_len** in hex form
dict_cell_hash – hash of representation of dictionary (table) built from provided categories and values, for debugging runmethod in lite-client
dict_size – number of entires in domain category table, which follows:

cat_*i* – category number of table entry
value_*i* – decoded addr or text form or hex form of entry value
value_hex_*i* – hexadecimal form of entry value

boc_file – contains file name where BoC was saved (if not **-R**)

For example, if you have **myres.tpriv** and **myres.addr** files that correspond to the resolver that you want to work on, and you want to replace category table of domain **TestDom** with the following table, and considering that current seqno of contract is **15**:

-2 (*owner*)	"F:new-wallet.addr"	*Load data from file new-wallet.addr*
1	TextValue	*Specified text-value as it is*
2	H:0102AABB	*Binary value 01 02 AA BB*

The following command is to be used:

```
fift -I.. -s mdns-setdom.fif myres - 15 TestDom -
        -2 "F:new-wallet.addr" 1 TextValue 2 "H:0102AABB"
```

The resulting ***myres-15*-dset.boc** file should be handled just like BoC query files in three previous sections (sent to network directly).

3.7 Deleting a domain

Sometimes it may be necessary to delete the entire domain (not just a single category entry), if that is the case you can use ***mdns-domdel.fif***:

```
fift -I.. -s mdns-domdel.fif <pkfile> <address> <seqno>
                                    <domain> [-r or -R]
```

The parameters of this script are absolutely same with respect to entry addition script (*3.4 Adding or replacing domain category entry*) with exception that there is no **value** and **category** parameters – because the entire domain is being deleted.

Consequently, the *robots ouput variables* are the same, with exception that **type** is now **ddel** instead of **vset**, and **value*** and **category** are missing.

For example, to delete domain **TestDom** that was exploited in previous examples you can issue following command (lets consider **seqno** = **16**):

```
fift -I.. -s mdns-domdel.fif myres - 16 TestDom
```

The resulting ***myres-16*-ddel.boc** file should be handled just like BoC query files in other sections (sent to network directly).

3.8 Changing ownership of the contract

If you need, **REALLY** need to change ownership to contract, that is, change the public key, which verifies signatures of all incoming messages, and which's counterpart private key is required to do any manipulations, you can use ***mdns-chown.fif*** script. Just make sure that you are really

sure that you want to change ownership, because once you do that there is no way back. If you are absolutely sure that you need to transfer ownership over the contract to another public key, issue following:

```
fift -I.. -s mdns-chown.fif <pkfile> <address> <seqno>
                            <new-key-file> [-r or -R]
```

pkfile, **address**, **seqno, [-r or -R]** – described in *section 3.4*.

new-key-file – must be a **full** file name (has to contain extension) of a **.tpriv** or **.tpub** key file, to which ownership will be transferred. It is not possible to specify the key itself here or use unsigned public key or non-hash-appended private key to ensure safety of the contract owner state.

Robots output variables are a very short subset of robots variables encountered in previous sections, having only **type=oset, pk_file, seqno, query, query_len** variables. Additional variable **new_key** contains hexadecimal representation of new public key.

For example, to transfer ownership from your key (of which you have at least private key file **myres.tpriv** and address file **myres.addr**) to another key, of which you have, for example, signed public key file **newowner.tpub**, assuming seqno is **666** you shall issue following comnd:

```
fift -I.. -s mdns-chown.fif myres - 666 newowner.tpub
```

The resulting ***myres-666*-oset.boc** file should be handled just like BoC query files in other sections (sent to network directly). **But think twice before doing that**, you would not be able to control the contract using **myres.tpriv** private key anymore!

3.9 Explanation of getter methods

This smart contract contains only two getter methods, that can be invoked:

seqno (→ int): 85143 – retrieves and returns current sequence number of the contract (implemented as an external message anti-replay attack as per smart-contract guidelines)

dnsresolve (slice, int → (int, cell)): 123660 – resolves provided (sub)domain and category, and returns corresponding value. For much more information about the resolving behaviour, logic, and input formats please refer to *section 2.7* for the automatic DNS resolver smart contract, with the only exception and difference is that domains have no expiration,

so any part about expiration time or current time requirements and c7 checks do not apply to this contract.

3.10 Implementation details

This contract is controlled with external messages which originate from outside the network. To actually be accepted and do something, those messages have to be signed by the contract owner's public key (which is stored into the contract at the moment of contract creation), and contain a valid data structure, instructing the contract of what to do and with what. All messages begin with **512-bit Signature** that signs repr hash of remainder of the message, followed by **32-bit Sequence number** for anti-replay protection, **6-bit Operation identifier (op)** determining which action should be taken and **1-bit Name reference flag (nameref)** which determines whether domain name is put inside the message (**inlined, value 0)** or as a separate reference to another cell (**externed, value 1**). All values afterwards are operation-dependent. Possible operations include:

op	desc	Description
11	vset	Adds (replaces) domain's category value
12	vdel	Deletes domain's category value
21	dset	Sets (replaces) domains' category table (all categories values)
22	ddel	Deletes a domain (removes it's category table and reference)
31	tset	*Replaces entire data prefix tree (all data of all domains)*
32	tdel	*Nullifies entire data prefix tree (deletes all domains)*
51	oset	Changes owner public key of the contract

It should be noted, that interface scripts for operations *31 tset* and *32 tdel* were intentionally left out and not created because of high risk of destructivity of such operations, and immense difficulty providing ability of user to adequately use them. Moreover, the usability of those operations is highly questioned, however they are implemented on the smart contract side, so if required, they can be carried out without deploying new contract.

First of all, operations 11 and 12 require category that is stored as **Int<16>** value. Afterwards, for operations 11, 12, 21 and 22 name is read from the message. In case of **inlined** name, *6-bit **byte** **n**ame **l**ength* is read from the message, afterwards name is read as a ***8bnl*** slice from input. Otherwise, referenced cell is loaded that is considered a name in its entirety. Following deserialization is different for each operation: *11 vset* requires reference to value cell, *21 dset* requires reference to category table dict, *31 tset* wants reference to new root of data prefix tree, and, finally, *51 oset* requires 256-bit new public key right in the message.

Internally, contract stores **32-bit sequence number**, **256-bit owner public key** and **reference to root of data prefix tree** containing data of all domains.

Names in data prefix tree are prefixed with 7-bit amount of zero-code characters in the name, please refer to 2.8 Implementation details of the automatic DNS resolver smart contract for a very detailed and throught explanation of implementation details of such keying and what they allow to do (details about the prefix trees). Obviously, you should skip details about internal messages, their codes and garbage collection dictionary, because they are irrelevant for this contract.

Nevertheless, there are also some obvious things to consider, comparing to automatic resolver, because this version of resolver is controlled by a specific owner with a specific key, therefore ability to register a concrete subdomain of a more generic domain may never be considered a security vulnerability. Lets consider that we have org∅ domain in the resolver, then owner may freely register the org∅telegram∅ domain. If owner wanted to influence the resolution result for org∅telegram∅ domain, he could simply change the -1 (next resolver) category value for org∅ domain. Considering that, ability to concretize subdomains and contain them simultaneously with the higher-level domain is actually very useful for manually-controlled DNS resolver smart contracts.

Another thing to consider and do not forget is that manual DNS resolver smart contract pays by itself for all occuring fees for processing incoming external messages from owner (gas fees) and for storing the data in the blockchain (storage fees). Therefore, owner should be careful and keep an eye on the balance of smart contract – if it goes negative, smart contract will be frozen, and later, destroyed.

8. Bonus contents

The ***8-Bonus*** directory contains some useful goodies, that can help reading, writing and understanding Fift and FunC code with help of UDL syntax highlighting in Notepad++. Of course if you don't have some IDE.

Provided UDL .xml files can be directly imported into Notepad++'s "Define your language" plugin to noticeably beautify aforementioned script types. Example of usage of those UDLs are as following:

```
def? $3 { @' $3 } { "" } cond =: Flags

variable RMode     false RMode  !
variable FRMode    false FRMode !
Flags "-r" $= { true RMode ! } if
Flags "-R" $= { true RMode ! true FRMode ! } if

RMode  @ =: RM
FRMode @ =: FRM

file-base dup +".tpub"  file-exists? =: FPubExists
          dup +".tpriv" file-exists? =: FPrivExists
          dup +".taddr" file-exists? =: FAddrExists
              +".boc"   file-exists? =: FBocExists

FPrivExists not FRM not and { // If tpriv does not exist we need some additional checks
    FPubExists  { ."Error! Public key file exists, but private key file does not!"   cr 1 }
    FAddrExists { ."Error! Address file exists, but private key file does not!"      cr 2 }
    FBocExists  { ."Error! Bag of cells file exists, but private key file does not!" cr 3 }
} if

variable bPrK   variable bPuK
variable uPrK   variable uPuK

RM { cr } ifnot
FPrivExists { // if private file exists, load the key
    RM { ."Loading *EXISTING* private key from " file-base +".tpriv" type ."..." cr } ifno
    file-base +".tpriv" file>B dup Blen dup 64 = {
        ."Error: File " file-base +".tpriv" type ." contains corrupted private key materia
        ."of length " . ."(64 expected)" cr 4 halt } ifnot     drop
    32 B| over -rot 256 B>u@ swap <b swap B, b> hash = {
```

```
        [Name<?>:subdomain]
    31 TSet: replace ENTIRE DOMAIN TABLE with the provided tree root cell (x=-)
        [Cell<1r>:new_domains_table]
    32 TDel: nullify ENTIRE DOMAIN TABLE (x=-)
    51 OSet: replace owner public key with a new one (x=-)
        [UInt<256b>:new_public_key]
-}

() recv_external(slice in_msg) impure {
    slice signature = in_msg~load_bits(512);
    slice data = get_data().begin_parse();
    int stored_seqno = data~load_uint(32);
    int public_key = data~load_uint(256);
    int shash = slice_hash(in_msg);
    int provided_seqno = in_msg~load_uint(32);
    throw_unless(33, provided_seqno == stored_seqno);
    throw_unless(34, check_signature(shash, signature, public_key));
    accept_message(); ;; message is signed by owner, sanity not guaranteed yet
    cell root = data~load_opt_ref(); ;; tree root (dictionary with domains)
    int op = in_msg~load_uint(6);
    int is_name_ref = (in_msg~load_uint(1) == 1);
    ;; 00 Contract initialization message (only if seqno = 0)
    if ((op == 0) & (stored_seqno == 0)) {
        ;; reinitialize data, set seqno = 1
        ;; possible rollover after 2^32 is not critical because it does not
        ;;   change any data, and requires owner signature anyway
        return store_data(stored_seqno + 1, public_key, root);
    }
    ;; lets assume at this point that special operations 00..09 are handled
    throw_if(44, op < 10);
    slice name = data; ;; anything! better do not begin or it costs much gas
```

9. Deprecated or not strictly related items

This folder (***9-Rubbish***) contains some items that may not be strictly a solution or a user script but may, nevertheless, pose some interest as for the competition itself or for the development / developers.

Sky-1-1-Multisig contains first experimental version or multisignature contract which involved usage of custom-built 4-search-trees to store the public keys in the contract data. It proved to be somehow not extremly versatile because custom implementation of 4ST is hard to manage, not mutable and requires very large amount of storage (because no native functions) and gas (because many opers are requires instead of just one).

That way, the implementation, at the current time, is not very practical, nevertheless, the monstrosity, that takes all public keys and builds a 4ST out of them in Fift language may be of some interest.

Unfortunately, I lacked time to recreate Multisig contract from scratch and make my own perfected smart contract... But this half-done prototype version is surely not tested enough and does not have interface scripts to participate with it.

TestingData contains some of auxilliary files and data (private and public keys, bags of cells, addresses, etc.) that were used in development and testing of smart contracts. May be somehow a little useful, but most likely not. Nevertheless, included for completity reason.

Various Fift and/or FunC scripts in this folder may be archival copies before some large refactoring processes to display some different approaches in time, while the development was done.

Conclusion

The competition and learning of absolutely new languages such as Fift and FunC was very interesting, the only sad thing is that almost 2/3 of the time I was unable to do the tasks themselves, and it took me a week to perfect those languages and create my versions of smart contracts for participation.

I hope this is not the last competition, where I would be able to make something useful and show off my computer engineering skills.

Anyway, for this competition, hoping for the best!