

Проблема индексации NFT айтемов (Логическая ошибка)

Тут, в файле `stbl_staking.fc`, происходит минт айтема, если стейка ещё не было

```
278
279     ;; if hasn't staked before
280     if (staker::staked_a == 0) {
281         cell item_message = begin_cell()
282             .store_slice(staker_address)           ;; item owner
283             .store_ref(self::nft_content)         ;; item content
284             .store_slice(self::initializer)       ;; editor
285         .end_cell();
286         cell nft_body = pack_nft_mint_message(msg::query_id, CONF::NFT_ITEM_AMOUNT, item_message);
287         cell nft_mint_msg = delib::int_msg(self::nft_collection_address, false, CONF::NFT_MINT_AMOUNT, nft_body, null());
288         send_new_message(nft_mint_msg, MSG::SENDER_PAYS_FEE);
289     }
```

Проблема в том, что в файле `helpers.fc`, в функции, которая формирует сообщение, отправляемое на контракт коллекции при минте айтема, индекс представлен константой - 0 (95 строка)

```
90
91     cell pack_nft_mint_message(int query_id, int item_value, cell nft_message) inline {
92         return begin_cell()
93             .store_uint(op::nft::mint, 32)
94             .store_uint(query_id, 64)
95             .store_int(0, 64) ;; item_index
96             .store_coins(item_value) ;; ton to send to the item
97             .store_ref(nft_message)
98             .end_cell();
99     }
```

Поэтому айтем с помощью этой функции можно заминтить только один, под индексом 0, при попытке заминтить повторно - будет ошибка 402, ниже код из файла `nft_collection.fc`

```
90
91     cell pack_nft_mint_message(int query_id, int item_value, cell nft_message) inline {
92         return begin_cell()
93             .store_uint(op::nft::mint, 32)
94             .store_uint(query_id, 64)
95             .store_int(0, 64) ;; item_index
96             .store_coins(item_value) ;; ton to send to the item
97             .store_ref(nft_message)
98             .end_cell();
99     }
```



Поэтому нужно запоминать индекс последнего айтема, который мы заминтили и инкрементировать этот индекс для следующих минтов.

```
280
281 let nftData = await nftCollection.getCollectionData();
282 expect(nftData.nextItemId).toEqual(1);
283
284 const userStakeSecond = await userWallet.sendTransfer(
285   user.getSender(),
286   toNano(0.5),
287   userJettons / 2n,
288   stblStaking.address,
289   user.address,
290   beginCell().endCell(),
291   toNano(0.4),
292   beginCell().endCell()
293 );
294
295 expect(userStakeSecond.transactions).toHaveTransaction({
296   from: stblStaking.address,
297   to: nftCollection.address,
298   op: 1, // deploy new nft
299 });
300
301 expect(userStakeSecond.transactions).toHaveTransaction({
302   from: nftCollection.address,
303 });
304
305 nftData = await nftCollection.getCollectionData();
306 expect(nftData.nextItemId).toEqual(2);
307
308 });
309
310 it("should not accept small stake", async () => {
311   //console.log("Staking Address: ", stblStaking.address)
312   const userStake = await userWallet.sendTransfer(
```

OUTPUT DEBUG CONSOLE PORTS PROBLEMS

TERMINAL

```
293 | )
294 |
> 295 | expect(userStakeSecond.transactions).toHaveTransaction({
296 |   from: stblStaking.address,
297 |   to: nftCollection.address,
298 |   op: 1, // deploy new nft
    |   ^
    |   at Object.<anonymous> (tests/StblStaking.spec.ts:295:50)
Test Suites: 1 failed, 1 total
Tests:      1 failed, 7 passed, 8 total
Snapshots:  0 total
Time:       7.542 s, estimated 9 s
Ran all test
```



При втором минте - индекс токена должен быть 2, но, как мы видим, вылетает ошибка

```
276
277     expect(userStake.transactions).toHaveTransaction({
278       from: nftCollection.address,
279     })
280
281     let nftData = await nftCollection.getCollectionData();
282     expect(nftData.nextItemId).toEqual(1);
283
284     // const userStakeSecond = await userWallet.sendTransfer(
285     //   user.getSender(),
286     //   toNano(0.5),
287     //   userJettons / 2n,
288     //   stblStaking.address,
289     //   user.address,
290     //   beginCell().endCell(),
291     //   toNano(0.4),
292     //   beginCell().endCell()
293     // );
294
295     expect(userStake.transactions).toHaveTransaction({
296       from: stblStaking.address,
297       to: nftCollection.address,
298       op: 1, // deploy new nft
299     })
300
301     expect(userStake.transactions).toHaveTransaction({
302       from: nftCollection.address,
303     })
304
305     nftData = await nftCollection.getCollectionData();
306     expect(nftData.nextItemId).toEqual(1);
307   });
308
309   it("should not accept small stake" as any, () => {
310
311     ✓ should accept proper stake (712 ms)
312     ✓ should accept proper stake and mint nft and mint only once (641 ms)
313     ✓ should not accept small stake (667 ms)
314     ✓ should accept jettons for reward (589 ms)
315     ✓ should stake successfully after the period and only after (861 ms)
316     ✓ should unstake user stake successfully and only once (701 ms)
317     ✓ should change owner (652 ms)
318
319     Test Suites: 1 passed, 1 total
320     Tests: 8 passed, 8 total
321     Snapshots: 0 total
322     Time: 7.452 s, estimated 8 s
323     Ran all test suites.
```

Ошибок нет, если после второй транзакции ожидать значение 1, так как nextItemId не изменился, а следовательно, айтем не задеплоился



Предпочтительнее использовать константы вместо ассемблерных примитивов (Использование лишнего газа)

Пример:

```
int op::transfer_ownership() asm "0x2da38aaf PUSHINT";  
const int op::transfer_ownership = 0x2da38aaf;
```

Стоит определять ошибки/оп-коды и прочие "константы" другим способом, не с помощью ассемблерных примитивов, а с помощью констант, так как второе помогает компилятору оптимизировать код самостоятельно, не стоит это делать за него лишний раз.

Компилятор FunC внутрь ассемблерной функции не смотрит. Например так, "256 PUSHINT" можно заменять на "8 PUSHPOW2", компилятор может воспользоваться другим кодом, создать значение в другой момент (если так будет удобнее по стеку).

Более того, нынешний подход расходует больше газа (<https://docs.ton.org/learn/tvm-instructions/instructions>).



Или же вот:

```
cell delib::EC() asm "<b>PUSHREF ;", // creates an empty cell
cell delib::stc(slice s) asm "NEWC SWAP STSLICER ENDC"; ;; converts a slice into cell
slice delib::addr_none() asm "b{00} PUSHSLICE"; // creates an addr_none$00 slice
```

```
begin_cell()
    .store_slice(s)
.end_cell()
```

вместо delib::stc(s) почти наверняка будет не хуже.

Особенно интересно, зачем SWAP STSLICER(26 ед.газа),
когда есть вполне нормальная операция STSLICE(18 ед. газа).

CE	STSLICE	s b - b'	Stores Slice s into Builder b.	18
CF16	STSLICER	b s - b'	Equivalent to SWAP STSLICE.	26

Это не все примеры, в коде полно функций, оформленных
в виде ассемблерных примитивов, которые бездумно расходуют газ.



inline/inline_ref (Совет по оптимизации)

```
104 cell jetton_transfer(int amount, slice to_address, slice response_address,  
105                      slice jetton_wall, int ton_amount) inline {  
106  
107     cell body = pack_simple_jetton_transfer(msg::query_id, amount, to_address,  
108                                             response_address, CONF::JETTON_FWD_AMOUNT);  
109  
110     return delib::int_msg(jetton_wall, false, ton_amount, body, null());  
111 }  
112  
113 () throw_jettons_128(slice to, slice jetton_wall, int amount) impure inline {  
114     cell msg = jetton_transfer(amount, to, to, jetton_wall, 0);  
115     send_raw_message(msg, MSG::ALL_NOT_RESERVED + MSG::IGNORE_ERRORS);  
116  
117     commit();  
118     throw(ERR::NOT_ENOUGH);  
119 }  
120  
121 (slice, cell) calc_escrow_contract(slice staker_address) inline {  
122     cell contract_data = pack_escrow_data(my_address(), staker_address);  
123     cell contract_init = delib::basic_state_init(self::escrow_contract_code, contract_data);  
124     return (delib::addrstd_by_state(CONF::WC, contract_init), contract_init);  
125 }  
126  
127 () auth_by_escrow_contract(slice staker_address) impure inline {  
128     (slice escrow_address, _) = calc_escrow_contract(staker_address);  
129     throw_unless(ERR::ACCESS, equal_slice_bits(msg::sender, escrow_address));  
130 }  
131  
132 ;; contract constructor (can be called only once)  
133 () constructor(slice msg body) impure inline ref {
```

Inline_ref specifier

The code of a function with the inline_ref specifier is put into a separate cell, and every time when the function is called, a CALLREF command is executed by TVM.

So it's similar to inline, but because a cell can be reused in several places without duplicating it, it is almost always more efficient in terms of code size to use inline_ref specifier instead of inline unless the function is called exactly once.

Recursive calls of inline_ref'ed functions are still impossible because there are no cyclic references in the TVM cells.



Источник: (<https://docs.ton.org/develop/func/functions>)

Как сказано в документации: "с точки зрения размера кода почти всегда более эффективно использовать `inline_ref` спецификатор вместо `inline`, если функция не вызывается ровно один раз.", поэтому непонятно почему именно тут спецификатор `inline`, хоть функции вызываются много раз (>> 1).



Источник: (<https://docs.ton.org/develop/func/functions>)

Как сказано в документации: "с точки зрения размера кода почти всегда более эффективно использовать `inline_ref` спецификатор вместо `inline`, если функция не вызывается ровно один раз.", поэтому непонятно почему именно тут спецификатор `inline`, хоть функции вызываются много раз (>> 1).

