

Audit

By OCamlPro

August 2, 2021

Contents

1	Introduction	8
2	Overview	9
3	General Remarks	10
3.1	Readability	10
3.1.1	Typography of Static Variables	10
3.1.2	Typography of Global Variables	10
3.1.3	Typography of Internal Functions	10
3.1.4	Naming of Errors	11
3.1.5	Better Units for Big Numbers	11
3.1.6	Use Method Calls instead of <code>tvm.encodeBody</code>	11
3.2	Security	11
3.2.1	Accept Methods without Checks	11
3.2.2	<code>require</code> after <code>tvm.accept</code>	11
4	Contract DEXClient	13
4.1	Contract Inheritance	13
4.2	Type Definitions	13
4.2.1	Struct Connector	13
4.2.2	Struct Callback	13
4.2.3	Struct Pair	14
4.3	Constant Definitions	14
4.4	Static Variable Definitions	14
4.5	Variable Definitions	14
4.6	Modifier Definitions	15
4.6.1	Modifier <code>alwaysAccept</code>	15
4.6.2	Modifier <code>checkOwnerAndAccept</code>	15
4.7	Constructor Definitions	15
4.7.1	Constructor	15
4.8	Public Method Definitions	15
4.8.1	Receive function	15
4.8.2	Function <code>connectCallback</code>	15
4.8.3	Function <code>connectPair</code>	16

4.8.4	Function connectRoot	16
4.8.5	Function createNewPair	16
4.8.6	Function getAllDataPreparation	17
4.8.7	Function getBalance	17
4.8.8	Function getCallback	17
4.8.9	Function getConnectorAddress	18
4.8.10	Function getPairData	18
4.8.11	Function processLiquidity	18
4.8.12	Function processSwapA	19
4.8.13	Function processSwapB	19
4.8.14	Function returnLiquidity	20
4.8.15	Function sendTokens	20
4.8.16	Function setPair	20
4.8.17	Function tokensReceivedCallback	21
4.9	Internal Method Definitions	21
4.9.1	Function computeConnectorAddress	21
4.9.2	Function getFirstCallback	21
4.9.3	Function getQuotient	22
4.9.4	Function getRemainder	22
4.9.5	Function isReady	22
4.9.6	Function isReadyToProvide	22
4.9.7	Function thisBalance	22
5	Contract DEXConnector	23
5.1	Contract Inheritance	23
5.2	Constant Definitions	23
5.3	Static Variable Definitions	23
5.4	Variable Definitions	24
5.5	Modifier Definitions	24
5.5.1	Modifier alwaysAccept	24
5.5.2	Modifier checkOwnerAndAccept	24
5.6	Constructor Definitions	24
5.6.1	Constructor	24
5.7	Public Method Definitions	25
5.7.1	Receive function	25
5.7.2	Function burn	25
5.7.3	Function deployEmptyWallet	25
5.7.4	Function expectedWalletAddressCallback	26
5.7.5	Function getBalance	26
5.7.6	Function setBouncedCallback	26
5.7.7	Function setTransferCallback	27
5.7.8	Function transfer	27
5.8	Internal Method Definitions	27
5.8.1	Function getQuotient	27
5.8.2	Function getRemainder	28

6	Contract DEXPair	29
6.1	Contract Inheritance	29
6.2	Type Definitions	29
6.2.1	Struct Connector	29
6.2.2	Struct Callback	29
6.3	Constant Definitions	30
6.4	Static Variable Definitions	30
6.5	Variable Definitions	30
6.6	Modifier Definitions	31
6.6.1	Modifier alwaysAccept	31
6.6.2	Modifier checkOwnerAndAccept	31
6.6.3	Modifier checkPubKeyAndAccept	31
6.7	Constructor Definitions	31
6.7.1	Constructor	31
6.8	Public Method Definitions	32
6.8.1	Receive function	32
6.8.2	Function burnCallback	32
6.8.3	Function connect	33
6.8.4	Function connectCallback	33
6.8.5	Function getBalance	34
6.8.6	Function getCallback	34
6.8.7	Function getConnectorAddress	35
6.8.8	Function tokensReceivedCallback	35
6.9	Internal Method Definitions	41
6.9.1	Function acceptForProvide	41
6.9.2	Function cleanProcessing	42
6.9.3	Function computeConnectorAddress	42
6.9.4	Function connectRoot	42
6.9.5	Function getAmountOut	43
6.9.6	Function getFirstCallback	43
6.9.7	Function getQuotient	43
6.9.8	Function getRemainder	44
6.9.9	Function liquidityA	44
6.9.10	Function liquidityB	44
6.9.11	Function qtyAforB	44
6.9.12	Function qtyBforA	45
6.9.13	Function thisBalance	45
7	Contract DEXroot	46
7.1	Contract Inheritance	46
7.2	Type Definitions	46
7.2.1	Struct Pair	46
7.3	Constant Definitions	46
7.4	Static Variable Definitions	46
7.5	Variable Definitions	46
7.6	Modifier Definitions	47

7.6.1	Modifier alwaysAccept	47
7.6.2	Modifier checkOwnerAndAccept	47
7.6.3	Modifier checkCreatorAndAccept	47
7.7	Constructor Definitions	48
7.7.1	Constructor	48
7.8	Public Method Definitions	48
7.8.1	Receive function	48
7.8.2	Function checkPubKey	48
7.8.3	Function createDEXclient	48
7.8.4	Function createDEXpair	49
7.8.5	Function getBalanceTONgrams	51
7.8.6	Function getClientAddress	51
7.8.7	Function getConnectorAddress	51
7.8.8	Function getPairAddress	51
7.8.9	Function getPairByRoots01	52
7.8.10	Function getPairByRoots10	52
7.8.11	Function getRootTokenAddress	52
7.8.12	Function getRootsByPair	53
7.8.13	Function sendTransfer	53
7.8.14	Function setCreator	53
7.8.15	Function setDEXclientCode	53
7.8.16	Function setDEXconnectorCode	54
7.8.17	Function setDEXpairCode	54
7.8.18	Function setRootTokenCode	54
7.8.19	Function setTONTokenWalletCode	54
7.9	Internal Method Definitions	55
7.9.1	Function computeClientAddress	55
7.9.2	Function computeConnectorAddress	55
7.9.3	Function computePairAddress	55
7.9.4	Function computeRootTokenAddress	56
8	Contract RootTokenContract	57
8.1	Contract Inheritance	57
8.2	Static Variable Definitions	57
8.3	Variable Definitions	57
8.4	Modifier Definitions	58
8.4.1	Modifier onlyOwner	58
8.4.2	Modifier onlyInternalOwner	58
8.5	Constructor Definitions	58
8.5.1	Constructor	58
8.6	Public Method Definitions	59
8.6.1	Fallback function	59
8.6.2	OnBounce function	59
8.6.3	Function deployEmptyWallet	59
8.6.4	Function deployWallet	60
8.6.5	Function getDetails	61

8.6.6	Function getTotalSupply	62
8.6.7	Function getVersion	62
8.6.8	Function getWalletAddress	62
8.6.9	Function getWalletCode	63
8.6.10	Function mint	63
8.6.11	Function proxyBurn	63
8.6.12	Function sendExpectedWalletAddress	64
8.6.13	Function sendPausedCallbackTo	65
8.6.14	Function sendSurplusGas	65
8.6.15	Function setPaused	65
8.6.16	Function tokensBurned	66
8.6.17	Function transferOwner	66
8.7	Internal Method Definitions	67
8.7.1	Function getExpectedWalletAddress	67
8.7.2	Function isExternalOwner	68
8.7.3	Function isInternalOwner	68
8.7.4	Function isOwner	68
9	Contract TONTokenWallet	69
9.1	Contract Inheritance	69
9.2	Static Variable Definitions	69
9.3	Variable Definitions	69
9.4	Modifier Definitions	70
9.4.1	Modifier onlyRoot	70
9.4.2	Modifier onlyOwner	70
9.4.3	Modifier onlyInternalOwner	70
9.5	Constructor Definitions	70
9.5.1	Constructor	70
9.6	Public Method Definitions	71
9.6.1	Fallback function	71
9.6.2	OnBounce function	71
9.6.3	Function accept	72
9.6.4	Function allowance	72
9.6.5	Function approve	72
9.6.6	Function balance	73
9.6.7	Function burnByOwner	73
9.6.8	Function burnByRoot	74
9.6.9	Function destroy	75
9.6.10	Function disapprove	75
9.6.11	Function getDetails	76
9.6.12	Function getVersion	76
9.6.13	Function getWalletCode	76
9.6.14	Function internalTransfer	76
9.6.15	Function internalTransferFrom	78
9.6.16	Function setBouncedCallback	79
9.6.17	Function setReceiveCallback	79

9.6.18	Function transfer	79
9.6.19	Function transferFrom	81
9.6.20	Function transferToRecipient	82
9.7	Internal Method Definitions	84
9.7.1	Function getExpectedAddress	84

Table of Issues

Minor issue: Typography of static variables	10
Minor issue: Typography of global variables	10
Minor issue: Typography of internal function	10
Minor issue: Naming of Errors	11
Minor issue: Use Better Units	11
Minor issue: Use Method Call instead of <code>tvm.encodeBody</code> . . .	11
Critical issue: No Accept All Methods	11
Critical issue: No Require after Accept	11

Chapter 1

Introduction

The present document is an official submission to the 13th contest of the ForMet sub-governance: *13 Radiance-DEX Phase 0 Formal Verification* <https://formet.gov.freeton.org/proposal?isGlobal=false&proposalAddress=0%3A07783c48e8789fa1163699e9e3071a47>

The specification was: *The contestants shall provide the informal audit of the central Radiance-DEX smart contracts (DEXClient, DEXConnector, DEXPair, DEXRoot RootTokenContract, TONTokenWallet), hereinafter referred to as “smart contracts”. where the detailed description of the “informal audit” is described below. All debot contracts are excluded from the present contest.*

and *All the source codes must be provided by the authors via <https://t.me/joinchat/-3zDgM62gQ020GUy> Telegram group. The code to be audited has a hash 7d65f0d3b85e504ac33f01395b6ba0ffef9d5fe5 (branch main, link - <https://github.com/radianceteam/dex2/commit/7d65f0d3b85e504ac33f01395b6ba0ffef9d5fe5>)*

and finally *Contestants shall submit a document in PDF format that covers:*

- *All the errors found*
- *All the warnings found*
- *All the “bad code” (long functions, violation of abstraction levels, poor readability etc.)*

Errors and warnings should be submitted to the developers as early as possible, during the contest, so that the code can be fixed immediately.

Chapter 2

Overview

The infrastructure is composed of a set of DEX specific contracts, associated with tokens contracts (developed by Broxus, to the best of our knowledge).

The DEX contracts are :

DEXRoot: The “root” contract, used to perform global operations, such as creating “client” contracts;

DEXClient: The contract with which a user may interact with the system.

DEXPair: The contract associated with a given pair of tokens (root token contracts)

DEXConnector: A simplified interface to interact with token contracts. The goal is probably to be able to interact with different implementations/interfaces of token contracts.

The token contracts are :

RootTokenContract: The root token contract, shared by all the wallet contracts for a given token;

TONTokenWallet: The wallet contract, containing the balance associated either with a public key or (exclusive) a contract address;

Compared to <https://github.com/broxus/broxus/ton-eth-bridge-token-contracts/>, the two token contracts have only been modified to change the `ton-solidity` pragma version.

All the DEX contracts use a static `soUINT` field to be able to instantiate several ones for a given public key or other static field.

Chapter 3

General Remarks

In this chapter, we introduce some general remarks about the code, that are not specific to this project, but whose occurrences have been found in the project in several locations.

3.1 Readability

3.1.1 Typography of Static Variables

Minor issue: Typography of static variables

A good coding convention is to use typography to visually discriminate static variables from other variables, for example using a prefix such as `s_`.

This issue was found everywhere in the code of DEX and token contracts.

3.1.2 Typography of Global Variables

Minor issue: Typography of global variables

A good coding convention is to use typography to visually discriminate global variables from local variables, for example using a prefix such as `m_` or `g_`.

This issue was found everywhere in the code of DEX and token contracts.

3.1.3 Typography of Internal Functions

Minor issue: Typography of internal function

A good coding convention is to use typography to visually discriminate public functions and internal functions, for example using a prefix such as `_`.

This issue was found everywhere in the code of DEX and token contracts.

3.1.4 Naming of Errors

Minor issue: Naming of Errors

A good coding convention is to define constants instead of using direct numbers for errors in `require()`.

This issue was found everywhere in the code of DEX contracts, but not for token contracts.

3.1.5 Better Units for Big Numbers

Minor issue: Use Better Units

Use decimals of `ton` instead of default nanotons to decrease the size of integer constants.

This issue was found in all constant definitions for gas cost. Numbers like 500000000 are difficult to read, whereas the equivalent `0.5 ton` is much easier.

3.1.6 Use Method Calls instead of `tvm.encodeBody`

Minor issue: Use Method Call instead of `tvm.encodeBody`

Using `tvm.encodeBody` makes code harder to read. Method calls are easier to read and interpret. The issue is minor as checks are still correctly performed on argument types.

This issue was found in all DEX contracts except `DEXRoot`.

3.2 Security

3.2.1 Accept Methods without Checks

Critical issue: No Accept All Methods

Public methods using `tvm.accept()` without any prior check should not exist. Indeed, such methods could be used by attackers to drain the balance of the contracts, even with minor amounts but unlimited number of messages.

This issue was found in the code the DEX contracts, especially with the `alwaysAccept()` modifier. Methods using this modifier should check the origin of the message and limit `tvm.accept()` to either the user or known contracts.

3.2.2 `require` after `tvm.accept`

Critical issue: No Require after Accept

Methods using `tvm.accept()` should never use `require()` after the accept. Indeed, a `require()` failing after `tvm.accept()` will cost a huge amount of gas, as all shards will execute the failing method.

This issue was found in the code of the DEX contracts. Methods should always keep calls to `require()` before `tvm.accept()`, and if it is not possible, should not fail but should *return* an error code instead.

Chapter 4

Contract DEXClient

In file DEXClient.sol

4.1 Contract Inheritance

ITokensReceivedCallback	
IDEXClient	
IDEXConnect	

4.2 Type Definitions

4.2.1 Struct Connector

```
29 struct Connector {
30     address root_address;
31     uint256 souint;
32     bool status;
33 }
```

4.2.2 Struct Callback

```
43 struct Callback {
44     address token_wallet;
45     address token_root;
46     uint128 amount;
47     uint256 sender_public_key;
48     address sender_address;
49     address sender_wallet;
50     address original_gas_to;
51     uint128 updated_balance;
52     uint8 payload_arg0;
53     address payload_arg1;
54     address payload_arg2;
55 }
```

4.2.3 Struct Pair

```
60 struct Pair {
61     bool status;
62     address rootA;
63     address walletA;
64     address rootB;
65     address walletB;
66     address rootAB;
67 }
```

4.3 Constant Definitions

```
23 uint128 constant GRAMS_CONNECT_PAIR = 500000000;
24 uint128 constant GRAMS_SET_CALLBACK_ADDR = 100000000;
25 uint128 constant GRAMS_SWAP = 500000000;
26 uint128 constant GRAMS_PROCESS_LIQUIDITY = 500000000;
27 uint128 constant GRAMS_RETURN_LIQUIDITY = 500000000;
```

4.4 Static Variable Definitions

```
18 address static public rootDEX;
19 uint256 static public soUINT;
20 TvmCell static public codeDEXConnector;
```

4.5 Variable Definitions

```
35 address[] public rootKeys;
36 mapping (address => address) public rootWallet;
37 mapping (address => address) public rootConnector;
38 mapping (address => Connector) connectors;
40 uint public counterCallback;
57 mapping (uint => Callback) callbacks;
69 mapping(address => Pair) public pairs;
70 address[] public pairKeys;
```

4.6 Modifier Definitions

4.6.1 Modifier alwaysAccept

```
73  modifier alwaysAccept {
74      tvm.accept();
75      _;
76  }
```

4.6.2 Modifier checkOwnerAndAccept

```
79  modifier checkOwnerAndAccept {
80      require(msg.pubkey() == tvm.pubkey(), 102);
81      tvm.accept();
82      _;
83  }
```

4.7 Constructor Definitions

4.7.1 Constructor

```
85  constructor() public {
86      require(msg.sender == rootDEX, 103);
87      tvm.accept();
88      counterCallback = 0;
89  }
```

4.8 Public Method Definitions

4.8.1 Receive function

```
413  receive() external {
414  }
```

4.8.2 Function connectCallback

```
181  function connectCallback(address wallet) public override
182      alwaysAccept {
183      address connector = msg.sender;
184      if (connectors.exists(connector)) {
185          Connector cc = connectors[connector];
186          rootKeys.push(cc.root_address);
187          rootWallet[cc.root_address] = wallet;
188          rootConnector[cc.root_address] = connector;
189          TvmCell bodySTC = tvm.encodeBody(IDEXConnector(connector).
190              setTransferCallback);
191          connector.transfer({value: GRAMS_SET_CALLBACK_ADDR, bounce:
192              true, flag: 0, body:bodySTC});
193          TvmCell bodySBC = tvm.encodeBody(IDEXConnector(connector).
194              setBouncedCallback);
```



```

191     connector.transfer({value: GRAMS_SET_CALLBACK_ADDR, bounce:
192         true, flag: 0, body:bodySBC});
193     cc.status = true;
194     connectors[connector] = cc;
195 }

```

4.8.3 Function connectPair

```

92     function connectPair(address pairAddr) public checkOwnerAndAccept
93         returns (bool statusConnection) {
94         statusConnection = false;
95         if (!pairs.exists(pairAddr)){
96             Pair cp = pairs[pairAddr];
97             cp.status = false;
98             pairs[pairAddr] = cp;
99             pairKeys.push(pairAddr);
100             TvmCell body = tvm.encodeBody(IDEXPair(pairAddr).connect);
101             pairAddr.transfer({value:GRAMS_CONNECT_PAIR, body:body});
102             statusConnection = true;
103         }
104     }

```

4.8.4 Function connectRoot

```

158     function connectRoot(address root, uint256 souint, uint128
159         gramsToConnector, uint128 gramsToRoot) public
160         checkOwnerAndAccept returns (bool statusConnected){
161         statusConnected = false;
162         if (!rootWallet.exists(root)) {
163             TvmCell stateInit = tvm.buildStateInit({
164                 contr: DEXConnector,
165                 varInit: { soUINT: souint, dexclient: address(this) },
166                 code: codeDEXConnector,
167                 pubkey: tvm.pubkey()
168             });
169             TvmCell init = tvm.encodeBody(DEXConnector);
170             address connector = tvm.deploy(stateInit, init,
171                 gramsToConnector, address(this).wid);
172             Connector cr = connectors[connector];
173             cr.root_address = root;
174             cr.souint = souint;
175             cr.status = false;
176             connectors[connector] = cr;
177             TvmCell body = tvm.encodeBody(IDEXConnector(connector).
178                 deployEmptyWallet, root);
179             connector.transfer({value:gramsToRoot, bounce:true, body:body
180                 });
181             statusConnected = true;
182         }
183     }

```

4.8.5 Function createNewPair

```

356     function createNewPair(
357         address root0,

```

```

358     address root1,
359     uint256 pairSoArg,
360     uint256 connectorSoArg0,
361     uint256 connectorSoArg1,
362     uint256 rootSoArg,
363     bytes rootName,
364     bytes rootSymbol,
365     uint8 rootDecimals,
366     uint128 gramsForPair,
367     uint128 gramsForRoot,
368     uint128 gramsForConnector,
369     uint128 gramsForWallet,
370     uint128 gramsTotal
371 ) public view checkOwnerAndAccept {
372     require (!(gramsTotal < (gramsForPair+2*gramsForConnector+2*
        gramsForWallet+gramsForRoot)) && !(gramsTotal < 5 ton)
        ,106);
373     require (!(address(this).balance < gramsTotal),105);
374     TvmCell body = tvm.encodeBody(IDEXRoot(rootDEX).createDEXpair,
        root0,root1,pairSoArg,connectorSoArg0,connectorSoArg1,
        rootSoArg,rootName,rootSymbol,rootDecimals,gramsForPair,
        gramsForRoot,gramsForConnector,gramsForWallet);
375     rootDEX.transfer({value:gramsTotal, bounce:false, flag: 1,
        body:body});
376 }

```

4.8.6 Function getAllDataPreparation

```

215     function getAllDataPreparation() public view alwaysAccept returns
        (address[] pairKeysR, address[] rootKeysR){
216         pairKeysR = pairKeys;
217         rootKeysR = rootKeys;
218     }

```

4.8.7 Function getBalance

```

351     function getBalance() public pure responsible returns (uint128) {
352         return { value: 0, bounce: false, flag: 64 } thisBalance();
353     }

```

4.8.8 Function getCallback

```

318     function getCallback(uint id) public view checkOwnerAndAccept
        returns (
319         address token_wallet,
320         address token_root,
321         uint128 amount,
322         uint256 sender_public_key,
323         address sender_address,
324         address sender_wallet,
325         address original_gas_to,
326         uint128 updated_balance,
327         uint8 payload_arg0,
328         address payload_arg1,
329         address payload_arg2
330     ){

```

```

331     Callback cc = callbacks[id];
332     token_wallet = cc.token_wallet;
333     token_root = cc.token_root;
334     amount = cc.amount;
335     sender_public_key = cc.sender_public_key;
336     sender_address = cc.sender_address;
337     sender_wallet = cc.sender_wallet;
338     original_gas_to = cc.original_gas_to;
339     updated_balance = cc.updated_balance;
340     payload_arg0 = cc.payload_arg0;
341     payload_arg1 = cc.payload_arg1;
342     payload_arg2 = cc.payload_arg2;
343 }

```

4.8.9 Function getConnectorAddress

```

153     function getConnectorAddress(uint256 connectorSoArg) public view
154         responsible returns (address) {
155         return { value: 0, bounce: false, flag: 64 }
            computeConnectorAddress( connectorSoArg);
156     }

```

4.8.10 Function getPairData

```

379     function getPairData(address pairAddr) public view alwaysAccept
380         returns (
381             bool pairStatus,
382             address pairRootA,
383             address pairWalletA,
384             address pairRootB,
385             address pairWalletB,
386             address pairRootAB,
387             address curPair
388         ){
389         Pair cp = pairs[pairAddr];
390         pairStatus = cp.status;
391         pairRootA = cp.rootA;
392         pairWalletA = cp.walletA;
393         pairRootB = cp.rootB;
394         pairWalletB = cp.walletB;
395         pairRootAB = cp.rootAB;
396         curPair = pairAddr;
397     }

```

4.8.11 Function processLiquidity

```

251     function processLiquidity(address pairAddr, uint128 qtyA, uint128
252         qtyB) public view checkOwnerAndAccept returns (bool
253         processLiquidityStatus) {
254         processLiquidityStatus = false;
255         if (isReadyToProvide(pairAddr)) {
256             Pair cp = pairs[pairAddr];
257             address connectorA = rootConnector[cp.rootA];
258             address connectorB = rootConnector[cp.rootB];
259             TvmBuilder builderA;

```

```

258     builderA.store(uint8(2), address(this), rootWallet[cp.rootAB
        ]);
259     TvmCell payloadA = builderA.toCell();
260     TvmBuilder builderB;
261     builderB.store(uint8(2), address(this), rootWallet[cp.rootAB
        ]);
262     TvmCell payloadB = builderB.toCell();
263     TvmCell bodyA = tvm.encodeBody(IDEXConnector(connectorA).
        transfer, cp.walletA, qtyA, payloadA);
264     TvmCell bodyB = tvm.encodeBody(IDEXConnector(connectorB).
        transfer, cp.walletB, qtyB, payloadB);
265     connectorA.transfer({value: GRAMS_PROCESS_LIQUIDITY, bounce:
        true, body:bodyA});
266     connectorB.transfer({value: GRAMS_PROCESS_LIQUIDITY, bounce:
        true, body:bodyB});
267     processLiquidityStatus = true;
268 }
269 }

```

4.8.12 Function processSwapA

```

221 function processSwapA(address pairAddr, uint128 qtyA) public view
        checkOwnerAndAccept returns (bool processSwapStatus) {
222     processSwapStatus = false;
223     if (isReady(pairAddr)) {
224         Pair cp = pairs[pairAddr];
225         address connector = rootConnector[cp.rootA];
226         TvmBuilder builder;
227         builder.store(uint8(1), cp.rootB, rootWallet[cp.rootB]);
228         TvmCell payload = builder.toCell();
229         TvmCell body = tvm.encodeBody(IDEXConnector(connector).
            transfer, cp.walletA, qtyA, payload);
230         connector.transfer({value: GRAMS_SWAP, bounce:true, body:body
            });
231         processSwapStatus = true;
232     }
233 }

```

4.8.13 Function processSwapB

```

236 function processSwapB(address pairAddr, uint128 qtyB) public view
        checkOwnerAndAccept returns (bool processSwapStatus) {
237     processSwapStatus = false;
238     if (isReady(pairAddr)) {
239         Pair cp = pairs[pairAddr];
240         address connector = rootConnector[cp.rootB];
241         TvmBuilder builder;
242         builder.store(uint8(1), cp.rootA, rootWallet[cp.rootA]);
243         TvmCell payload = builder.toCell();
244         TvmCell body = tvm.encodeBody(IDEXConnector(connector).
            transfer, cp.walletB, qtyB, payload);
245         connector.transfer({value: GRAMS_SWAP, bounce:true, body:body
            });
246         processSwapStatus = true;
247     }
248 }

```

4.8.14 Function returnLiquidity

```
272 function returnLiquidity(address pairAddr, uint128 tokens) public
    view checkOwnerAndAccept returns (bool returnLiquidityStatus)
    {
273     returnLiquidityStatus = false;
274     if (isReadyToProvide(pairAddr)) {
275         Pair cp = pairs[pairAddr];
276         TvmBuilder builder;
277         builder.store(uint8(3), rootWallet[cp.rootA], rootWallet[cp.
            rootB]);
278         TvmCell callback_payload = builder.toCell();
279         TvmCell body = tvm.encodeBody(IDEXConnector(rootConnector[cp.
            rootAB]).burn, tokens, pairAddr, callback_payload);
280         rootConnector[cp.rootAB].transfer({value:GRAMS_RETURN_LIQUIDITY
            , body:body});
281         returnLiquidityStatus = true;
282     }
283 }
```

4.8.15 Function sendTokens

```
399 function sendTokens(address tokenRoot, address to, uint128 tokens
    , uint128 grams) public checkOwnerAndAccept view returns (
    bool sendTokenStatus){
400     sendTokenStatus = false;
401     if (rootConnector[tokenRoot] != address(0)) {
402         address connector = rootConnector[tokenRoot];
403         TvmBuilder builder;
404         builder.store(uint8(4), address(this), rootWallet[tokenRoot])
            ;
405         TvmCell payload = builder.toCell();
406         TvmCell body = tvm.encodeBody(IDEXConnector(connector).
            transfer, to, tokens, payload);
407         connector.transfer({value: grams, bounce:true, body:body});
408         sendTokenStatus = true;
409     }
410 }
```

4.8.16 Function setPair

```
127 function setPair(address arg0, address arg1, address arg2,
    address arg3, address arg4) public alwaysAccept override {
128     address dexpair = msg.sender;
129     if (pairs.exists(dexpair)){
130         Pair cp = pairs[dexpair];
131         cp.status = true;
132         cp.rootA = arg0;
133         cp.walletA = arg1;
134         cp.rootB = arg2;
135         cp.walletB = arg3;
136         cp.rootAB = arg4;
137         pairs[dexpair] = cp;
138     }
139 }
```

4.8.17 Function tokensReceivedCallback

```
286 function tokensReceivedCallback(  
287     address token_wallet,  
288     address token_root,  
289     uint128 amount,  
290     uint256 sender_public_key,  
291     address sender_address,  
292     address sender_wallet,  
293     address original_gas_to,  
294     uint128 updated_balance,  
295     TvmCell payload  
296 ) public override alwaysAccept {  
297     Callback cc = callbacks[counterCallback];  
298     cc.token_wallet = token_wallet;  
299     cc.token_root = token_root;  
300     cc.amount = amount;  
301     cc.sender_public_key = sender_public_key;  
302     cc.sender_address = sender_address;  
303     cc.sender_wallet = sender_wallet;  
304     cc.original_gas_to = original_gas_to;  
305     cc.updated_balance = updated_balance;  
306     TvmSlice slice = payload.toSlice();  
307     (uint8 arg0, address arg1, address arg2) = slice.decode(uint8,  
308         address, address);  
309     cc.payload_arg0 = arg0;  
310     cc.payload_arg1 = arg1;  
311     cc.payload_arg2 = arg2;  
312     callbacks[counterCallback] = cc;  
313     counterCallback++;  
314     if (counterCallback > 10){delete callbacks[getFirstCallback()  
    ];}
```

4.9 Internal Method Definitions

4.9.1 Function computeConnectorAddress

```
142 function computeConnectorAddress(uint256 souint) private inline  
143     view returns (address) {  
144     TvmCell stateInit = tvm.buildStateInit({  
145         contr: DEXConnector,  
146         varInit: { soUINT: souint, dexclient: address(this) },  
147         code: codeDEXConnector,  
148         pubkey: tvm.pubkey()  
149     });  
150     return address(tvm.hash(stateInit));  
151 }
```

4.9.2 Function getFirstCallback

```
121 function getFirstCallback() private view returns (uint) {  
122     optional(uint, Callback) rc = callbacks.min();  
123     if (rc.hasValue()) {(uint number, ) = rc.get();return number;}  
124     else {return 0;}
```

```
124 }
```

4.9.3 Function getQuotient

```
109 function getQuotient(uint128 arg0, uint128 arg1, uint128 arg2)
    private inline pure returns (uint128) {
110     (uint128 quotient, ) = math.muldivmod(arg0, arg1, arg2);
111     return quotient;
112 }
```

4.9.4 Function getRemainder

```
115 function getRemainder(uint128 arg0, uint128 arg1, uint128 arg2)
    private inline pure returns (uint128) {
116     (, uint128 remainder) = math.muldivmod(arg0, arg1, arg2);
117     return remainder;
118 }
```

4.9.5 Function isReady

```
198 function isReady(address pair) private inline view returns (bool)
    {
199     Pair cp = pairs[pair];
200     Connector ccA = connectors[rootConnector[cp.rootA]];
201     Connector ccB = connectors[rootConnector[cp.rootB]];
202     return cp.status && rootWallet.exists(cp.rootA) && rootWallet.
        exists(cp.rootB) && rootConnector.exists(cp.rootA) &&
        rootConnector.exists(cp.rootB) && ccA.status && ccB.status;
203 }
```

4.9.6 Function isReadyToProvide

```
206 function isReadyToProvide(address pair) private inline view
    returns (bool) {
207     Pair cp = pairs[pair];
208     Connector ccA = connectors[rootConnector[cp.rootA]];
209     Connector ccB = connectors[rootConnector[cp.rootB]];
210     return cp.status && rootWallet.exists(cp.rootA) && rootWallet.
        exists(cp.rootB) && rootWallet.exists(cp.rootAB) &&
        rootConnector.exists(cp.rootA) && rootConnector.exists(cp.
        rootB) && ccA.status && ccB.status;
211 }
```

4.9.7 Function thisBalance

```
346 function thisBalance() private inline pure returns (uint128) {
347     return address(this).balance;
348 }
```

Chapter 5

Contract DEXConnector

In file `DEXConnector.sol`

5.1 Contract Inheritance

IEpectedWalletAddressCallback	
IDEXConnector	

5.2 Constant Definitions

- Minor issue: see Better Units for Big Numbers (3.1.5)

```
19  uint128 constant GRAMS_TO_ROOT = 5000000000;
```

```
20  uint128 constant GRAMS_TO_NEW_WALLET = 2500000000;
```

5.3 Static Variable Definitions

- Minor issue: see Typography of Static Variables (3.1.1)
- Minor issue: why is `drivenRoot` not a static variable ? it looks like there is only one possible `DEXConnector` for a given pair of `DEXClient` and `RootTokenContract`.

```
15  uint256 static public soUINT;
```

```
16  address static public dexclient;
```


5.4 Variable Definitions

- Minor issue: see Typography of Global Variables (3.1.2)

```
22 address public drivenRoot;
```

```
23 address public driven;
```

```
24 bool public statusConnected;
```

5.5 Modifier Definitions

5.5.1 Modifier alwaysAccept

- Critical issue: see Accept Methods without Checks (3.2.1). This modifier should be removed.

```
27 modifier alwaysAccept {  
28     tvn.accept();  
29     -;  
30 }
```

5.5.2 Modifier checkOwnerAndAccept

- Minor issue: see Naming of Errors (3.1.4). Constants should be used instead of direct numbers.

```
32 modifier checkOwnerAndAccept {  
33     // Check that message from contract owner.  
34     require(msg.sender == dexclient, 101);  
35     tvn.accept();  
36     -;  
37 }
```

5.6 Constructor Definitions

5.6.1 Constructor

- OK

```
39 constructor() public checkOwnerAndAccept {  
40     statusConnected = false;  
41 }
```

5.7 Public Method Definitions

5.7.1 Receive function

- OK

```
129 receive() external {  
130 }
```

5.7.2 Function burn

- Minor Issue: this method should check `require(statusConnected,...)`
- Minor Issue: see Naming of Errors (3.1.4)
- Minor Issue: see Method Calls instead of `tvm.encodeBody` (3.1.6)

```
116 function burn(uint128 tokens, address callback_address, TvmCell  
    callback_payload) public override {  
117     require(msg.sender == dexclient, 101);  
118     tvm.rawReserve(address(this).balance - msg.value, 2);  
119     TvmCell body = tvm.encodeBody(IBurnableByOwnerTokenWallet(  
        driven).burnByOwner, tokens, 0, dexclient, callback_address  
        , callback_payload);  
120     driven.transfer({value: 0, bounce:true, flag: 128, body:body});  
121 }
```

5.7.3 Function deployEmptyWallet

- Minor Issue: if this method is called twice with different roots, the second one may still be executed before `statusConnected` is set in `expectedWalletAddressCallback`. It's a minor issue, as only the second call will finally set `driven` with the value associated with the second `drivenRoot`, as `drivenRoot` is correctly checked in `expectedWalletAddressCallback`. The only drawback is a potential small loss in gas. Anyway, this would not be possible if `drivenRoot` was a static global variable of the contract.
- Minor Issue: It would probably be better to use `require(!statusConnected,...)` instead of `if(!statusConnected)..` to fail in case of called twice.
- Minor Issue: see Naming of Errors (3.1.4)
- Minor Issue: see Method Calls instead of `tvm.encodeBody` (3.1.6)

```
60 function deployEmptyWallet(address root) public override {  
61     require(msg.sender == dexclient, 101);  
62     require(!(msg.value < GRAMS_TO_ROOT * 2), 103);  
63     tvm.rawReserve(address(this).balance - msg.value, 2);  
64     if (!statusConnected) {  
65         drivenRoot = root;
```

```

66     TvmCell bodyD = tvm.encodeBody(IRootTokenContract(root).
        deployEmptyWallet, GRAMS_TO_NEW_WALLET, 0, address(this),
        dexclient);
67     root.transfer({value:GRAMS_TO_ROOT, bounce:true, body:bodyD})
        ;
68     TvmCell bodyA = tvm.encodeBody(IRootTokenContract(root).
        sendExpectedWalletAddress, 0, address(this), address(this)
        ));
69     root.transfer({value:GRAMS_TO_ROOT, bounce:true, body:bodyA})
        ;
70     dexclient.transfer({value: 0, bounce:true, flag: 128});
71 } else {
72     dexclient.transfer({value: 0, bounce:true, flag: 128});
73 }
74 }

```

5.7.4 Function expectedWalletAddressCallback

- Minor Issue: this method should check `require(!statusConnected,...)`
- Minor Issue: see Naming of Errors (3.1.4)
- Minor Issue: see Method Calls instead of `tvm.encodeBody` (3.1.6)

```

77 function expectedWalletAddressCallback(address wallet, uint256
    wallet_public_key, address owner_address) public override {
78     require(msg.sender == drivenRoot && wallet_public_key == 0 &&
        owner_address == address(this), 102);
79     tvm.rawReserve(address(this).balance - msg.value, 2);
80     statusConnected = true;
81     driven = wallet;
82     TvmCell body = tvm.encodeBody(IDEXConnect(dexclient).
        connectCallback, wallet);
83     dexclient.transfer({value: 0, bounce:true, flag: 128, body:body
        });
84 }

```

5.7.5 Function getBalance

- Minor issue: is there a good reason to use `checkOwnerAndAccept` to allow the user to spend gas to get the balance when this action can be performed without spending gas (through the GraphQL interface or through get-methods executed locally).

```

124 function getBalance() public pure checkOwnerAndAccept returns (
    uint128 balance){
125     balance = address(this).balance;
126 }

```

5.7.6 Function setBouncedCallback

- Minor Issue: there is no real need for `setBouncedCallback` and `setTransferCallback` to be in two different methods instead of having a single method performing both actions, as they are always called together.
- Minor Issue: this method should check `require(statusConnected,...)`
- Minor Issue: see Naming of Errors (3.1.4)
- Minor Issue: see Method Calls instead of `tvm.encodeBody` (3.1.6)

```
95 function setBouncedCallback() public override {
96     require(msg.sender == dexclient, 101);
97     tvm.rawReserve(address(this).balance - msg.value, 2);
98     TvmCell body = tvm.encodeBody(ITONTTokenWallet(driven).
        setBouncedCallback, dexclient);
99     driven.transfer({value: 0, bounce:true, flag: 128, body:body});
100 }
```

5.7.7 Function setTransferCallback

- Minor Issue: this method should check `require(statusConnected,...)`
- Minor Issue: see Naming of Errors (3.1.4)
- Minor Issue: see Method Calls instead of `tvm.encodeBody` (3.1.6)

```
87 function setTransferCallback() public override {
88     require(msg.sender == dexclient, 101);
89     tvm.rawReserve(address(this).balance - msg.value, 2);
90     TvmCell body = tvm.encodeBody(ITONTTokenWallet(driven).
        setReceiveCallback, dexclient, true);
91     driven.transfer({value: 0, bounce:true, flag: 128, body:body});
92 }
```

5.7.8 Function transfer

- Minor Issue: this method should check `require(statusConnected,...)`
- Minor Issue: see Naming of Errors (3.1.4)
- Minor Issue: see Method Calls instead of `tvm.encodeBody` (3.1.6)

```
108 function transfer(address to, uint128 tokens, TvmCell payload)
        public override {
109     require(msg.sender == dexclient, 101);
110     tvm.rawReserve(address(this).balance - msg.value, 2);
111     TvmCell body = tvm.encodeBody(ITONTTokenWallet(driven).transfer,
        to, tokens, 0, dexclient, true, payload);
112     driven.transfer({value: 0, bounce:true, flag: 128, body:body});
113 }
```

5.8 Internal Method Definitions

5.8.1 Function getQuotient

- OK

```
48  function getQuotient(uint128 arg0, uint128 arg1, uint128 arg2)
    private inline pure returns (uint128) {
49      (uint128 quotient, ) = math.muldivmod(arg0, arg1, arg2);
50      return quotient;
51  }
```

5.8.2 Function getRemainder

- OK

```
54  function getRemainder(uint128 arg0, uint128 arg1, uint128 arg2)
    private inline pure returns (uint128) {
55      (, uint128 remainder) = math.muldivmod(arg0, arg1, arg2);
56      return remainder;
57  }
```

Chapter 6

Contract DEXPair

In file DEXPair.sol

6.1 Contract Inheritance

IDEXPair	
IDEXConnect	
ITokensReceivedCallback	
IBurnTokensCallback	

6.2 Type Definitions

6.2.1 Struct Connector

```
37 struct Connector {
38     address root_address;
39     uint256 souint;
40     bool status;
41 }
```

6.2.2 Struct Callback

```
48 struct Callback {
49     address token_wallet;
50     address token_root;
51     uint128 amount;
52     uint256 sender_public_key;
53     address sender_address;
54     address sender_wallet;
55     address original_gas_to;
56     uint128 updated_balance;
57     uint8 payload_arg0;
58     address payload_arg1;
59     address payload_arg2;
```

```
60 }
```

6.3 Constant Definitions

```
65 uint128 constant GRAMS_SET_CALLBACK_ADDR = 500000000;
```

```
66 uint128 constant GRAMS_SEND_UNUSED_RETURN = 100000000;
```

```
67 uint128 constant GRAMS_MINT = 50000000;
```

```
68 uint128 constant GRAMS_RETURN = 200000000;
```

6.4 Static Variable Definitions

```
19 address static public rootDEX;
```

```
20 uint256 static public soUINT;
```

```
21 address static public creator;
```

```
22 TvmCell static public codeDEXConnector;
```

```
23 address static public rootA;
```

```
24 address static public rootB;
```

```
25 address static public rootAB;
```

6.5 Variable Definitions

```
27 mapping(address => address) public walletReserve;
```

```
28 mapping(address => bool) public syncStatus;
```

```
29 mapping(address => uint128) public balanceReserve;
```

```
31 uint128 public totalSupply;
```

```
33 mapping(address => mapping(address => bool)) public  
    processingStatus;
```

```
34 mapping(address => mapping(address => uint128)) public  
    processingData;
```

```
35 mapping(address => mapping(address => address)) public  
    processingDest;
```

```
43 mapping (address => address) public rootConnector;
```

```
44 mapping (address => Connector) public connectors;
```

```

46  uint public counterCallback;

62  mapping (uint => Callback) callbacks;

```

6.6 Modifier Definitions

6.6.1 Modifier alwaysAccept

```

71  modifier alwaysAccept {
72      tvn.accept();
73      _;
74  }

```

6.6.2 Modifier checkOwnerAndAccept

```

76  modifier checkOwnerAndAccept {
77      require(msg.sender == rootDEX, 102);
78      tvn.accept();
79      _;
80  }

```

6.6.3 Modifier checkPubKeyAndAccept

```

82  modifier checkPubKeyAndAccept {
83      require(msg.pubkey() == tvn.pubkey(), 103);
84      tvn.accept();
85      _;
86  }

```

6.7 Constructor Definitions

6.7.1 Constructor

```

88  constructor(uint256 souintA, uint256 souintB, uint128
      gramsDeployConnector, uint128 gramsDeployWallet) public
      checkOwnerAndAccept {
89      counterCallback = 0;
90      connectRoot(rootA, souintA, gramsDeployConnector,
      gramsDeployWallet);
91      connectRoot(rootB, souintB, gramsDeployConnector,
      gramsDeployWallet);
92  }

```


6.8 Public Method Definitions

6.8.1 Receive function

```
609 receive() external {  
610 }
```

6.8.2 Function burnCallback

```
522 function burnCallback(  
523     uint128 tokens,  
524     TvmCell payload,  
525     uint256 sender_public_key,  
526     address sender_address,  
527     address wallet_address,  
528     address send_gas_to  
529 ) public override alwaysAccept {  
530     if (msg.sender == rootAB) {  
531         tvm.rawReserve(address(this).balance - msg.value, 2);  
532         TvmSlice slice = payload.toSlice();  
533         (uint8 arg0, address arg1, address arg2) = slice.decode(uint8,  
534             address, address);  
535         counterCallback++;  
536         Callback cc = callbacks[counterCallback];  
537         cc.token_wallet = wallet_address;  
538         cc.token_root = msg.sender;  
539         cc.amount = tokens;  
540         cc.sender_public_key = sender_public_key;  
541         cc.sender_address = sender_address;  
542         cc.sender_wallet = wallet_address;  
543         cc.original_gas_to = address(0);  
544         cc.updated_balance = 0;  
545         cc.payload_arg0 = arg0;  
546         cc.payload_arg1 = arg1;  
547         cc.payload_arg2 = arg2;  
548         callbacks[counterCallback] = cc;  
549         if (arg0 == 3 && arg1 != address(0) && arg2 != address(0)) {  
550             uint128 returnA = math.muldiv(balanceReserve[rootA], tokens,  
551                 totalSupply);  
552             uint128 returnB = math.muldiv(balanceReserve[rootB], tokens,  
553                 totalSupply);  
554             if (!(returnA > balanceReserve[rootA]) && !(returnB >  
555                 balanceReserve[rootB])) {
```

```

552     totalSupply -= tokens;
553     balanceReserve[rootA] -= returnA;
554     balanceReserve[rootB] -= returnB;
555     TvmBuilder builder;
556     builder.store(uint8(6), address(0), address(0));
557     TvmCell new_payload = builder.toCell();
558     TvmCell bodyA = tvml.encodeBody(IDEXConnector(rootConnector[
559         rootA]).transfer, arg1, returnA, new_payload);
560     TvmCell bodyB = tvml.encodeBody(IDEXConnector(rootConnector[
561         rootB]).transfer, arg2, returnB, new_payload);
562     rootConnector[rootA].transfer({value: GRAMS_RETURN, bounce:
563         true, body:bodyA});
564     rootConnector[rootB].transfer({value: GRAMS_RETURN, bounce:
565         true, body:bodyB});
566     if (counterCallback > 10){delete callbacks[getFirstCallback
567         ()];}
568     send_gas_to.transfer({value: 0, bounce:true, flag: 128});
569     if (counterCallback > 10){delete callbacks[getFirstCallback()
570         ];}
571 }
572 if (counterCallback > 10){delete callbacks[getFirstCallback()
573     ];}
574 }
575 }

```

6.8.3 Function connect

```

149 function connect() public override {
150     address dexclient = msg.sender;
151     tvml.rawReserve(address(this).balance - msg.value, 2);
152     TvmCell body = tvml.encodeBody(IDEXClient(dexclient).setPair,
153         rootA, walletReserve[rootA], rootB, walletReserve[rootB],
154         rootAB);
155     dexclient.transfer({ value: 0, flag: 128, body:body});
156 }

```

6.8.4 Function connectCallback

```

132 function connectCallback(address wallet) public override
133     alwaysAccept {
134     address connector = msg.sender;
135     if (connectors.exists(connector)) {
136         Connector cr = connectors[connector];

```

```

136     walletReserve[cr.root_address] = wallet;
137     syncStatus[cr.root_address] = true;
138     rootConnector[cr.root_address] = connector;
139     TvmCell bodySTC = tvm.encodeBody(IDEXConnector(connector).
        setTransferCallback);
140     connector.transfer({value: GRAMS_SET_CALLBACK_ADDR, bounce:
        true, flag: 0, body:bodySTC});
141     TvmCell bodySBC = tvm.encodeBody(IDEXConnector(connector).
        setBouncedCallback);
142     connector.transfer({value: GRAMS_SET_CALLBACK_ADDR, bounce:
        true, flag: 0, body:bodySBC});
143     cr.status = true;
144     connectors[connector] = cr;
145 }
146 }

```

6.8.5 Function getBalance

```

604 function getBalance() public pure responsible returns (uint128) {
605     return { value: 0, bounce: false, flag: 64 } thisBalance();
606 }

```

6.8.6 Function getCallback

```

571 function getCallback(uint id) public view checkPubKeyAndAccept
    returns (
572     address token_wallet,
573     address token_root,
574     uint128 amount,
575     uint256 sender_public_key,
576     address sender_address,
577     address sender_wallet,
578     address original_gas_to,
579     uint128 updated_balance,
580     uint8 payload_arg0,
581     address payload_arg1,
582     address payload_arg2
583 ){
584     Callback cc = callbacks[id];
585     token_wallet = cc.token_wallet;
586     token_root = cc.token_root;
587     amount = cc.amount;
588     sender_public_key = cc.sender_public_key;
589     sender_address = cc.sender_address;

```

```

590     sender_wallet = cc.sender_wallet;
591     original_gas_to = cc.original_gas_to;
592     updated_balance = cc.updated_balance;
593     payload_arg0 = cc.payload_arg0;
594     payload_arg1 = cc.payload_arg1;
595     payload_arg2 = cc.payload_arg2;
596 }

```

6.8.7 Function getConnectorAddress

```

108     function getConnectorAddress(uint256 connectorSoArg) public view
109         responsible returns (address) {
110             return { value: 0, bounce: false, flag: 64 }
111                 computeConnectorAddress( connectorSoArg);
112         }

```

6.8.8 Function tokensReceivedCallback

```

248     function tokensReceivedCallback(
249         address token_wallet,
250         address token_root,
251         uint128 amount,
252         uint256 sender_public_key,
253         address sender_address,
254         address sender_wallet,
255         address original_gas_to,
256         uint128 updated_balance,
257         TvmCell payload
258     ) public override alwaysAccept {
259         if (msg.sender == walletReserve[rootA] || msg.sender ==
260             walletReserve[rootB]) {
261             if (counterCallback > 10) {
262                 Callback cc = callbacks[counterCallback];
263                 cc.token_wallet = token_wallet;
264                 cc.token_root = token_root;
265                 cc.amount = amount;
266                 cc.sender_public_key = sender_public_key;
267                 cc.sender_address = sender_address;
268                 cc.sender_wallet = sender_wallet;
269                 cc.original_gas_to = original_gas_to;
270                 cc.updated_balance = updated_balance;
271                 TvmSlice slice = payload.toSlice();
272                 (uint8 arg0, address arg1, address arg2) = slice.decode(
273                     uint8, address, address);

```

```

272     cc.payload_arg0 = arg0;
273     cc.payload_arg1 = arg1;
274     cc.payload_arg2 = arg2;
275     callbacks[counterCallback] = cc;
276     counterCallback++;
277     delete callbacks[getFirstCallback()];
278     if (arg0 == 1) {
279         tvn.rawReserve(address(this).balance - msg.value, 2);
280         uint128 amountOut = getAmountOut(amount, token_root, arg1
        );
281         if (!(amountOut > balanceReserve[arg1])){
282             balanceReserve[token_root] += amount;
283             balanceReserve[arg1] -= amountOut;
284             syncStatus[token_root] = balanceReserve[token_root] ==
                updated_balance ? true : false;
285             TvmBuilder builder;
286             builder.store(uint8(0), address(0), address(0));
287             TvmCell new_payload = builder.toCell();
288             TvmCell body = tvn.encodeBody(IDEXConnector(
                rootConnector[arg1]).transfer, arg2, amountOut,
                new_payload);
289             rootConnector[arg1].transfer({value: 0, bounce:true,
                flag: 128, body:body});
290         } else {
291             TvmBuilder builder;
292             builder.store(uint8(8), address(0), address(0));
293             TvmCell new_payload = builder.toCell();
294             TvmCell body = tvn.encodeBody(IDEXConnector(
                rootConnector[token_root]).transfer, token_wallet,
                amount, new_payload);
295             rootConnector[token_root].transfer({value: 0, bounce:
                true, flag: 128, body:body});
296         }
297     }
298     if (arg0 == 2) {
299         tvn.rawReserve(address(this).balance - msg.value, 2);
300         processingStatus[token_root][arg1] = true;
301         processingData[token_root][arg1] += amount;
302         processingDest[token_root][arg1] = sender_wallet;
303         if (processingStatus[rootA][arg1] == true &&
            processingStatus[rootB][arg1] == true) {
304             uint128 amountA = processingData[rootA][arg1];
305             uint128 amountB = processingData[rootB][arg1];
306             if (totalSupply == 0 && balanceReserve[rootA] == 0 &&
                balanceReserve[rootB] == 0) {
307                 uint128 liquidity = math.min(amountA, amountB);
308                 balanceReserve[rootA] += amountA;
309                 balanceReserve[rootB] += amountB;
310                 totalSupply += liquidity;
311                 TvmCell body = tvn.encodeBody(IRootTokenContract(
                    rootAB).mint, liquidity, arg2);
312                 rootAB.transfer({value: GRAMS_MINT, bounce:true, body
                    :body});
313                 cleanProcessing(arg1);
314                 arg1.transfer({ value: 0, flag: 128});
315             } else {
316                 (uint128 provideA, uint128 provideB) =

```

```

317         acceptForProvide(amountA, amountB);
318     if (provideA > 0 && provideB > 0) {
319         uint128 liquidity = math.min(liquidityA(provideA),
320             liquidityB(provideB));
321         uint128 unusedReturnA = amountA - provideA;
322         uint128 unusedReturnB = amountB - provideB;
323         balanceReserve[rootA] += provideA;
324         balanceReserve[rootB] += provideB;
325         totalSupply += liquidity;
326         TvmCell body = tvml.encodeBody(IRootTokenContract(
327             rootAB).mint, liquidity, arg2);
328         rootAB.transfer({value: GRAMS_MINT, bounce:true,
329             body:body});
330         if (unusedReturnA > 0 && unusedReturnB > 0) {
331             TvmBuilder builder;
332             builder.store(uint8(7), address(0), address(0));
333             TvmCell new_payload = builder.toCell();
334             TvmCell bodyA = tvml.encodeBody(IDEXConnector(
335                 rootConnector[rootA]).transfer,
336                 processingDest[rootA][arg1], unusedReturnA,
337                 new_payload);
338             TvmCell bodyB = tvml.encodeBody(IDEXConnector(
339                 rootConnector[rootB]).transfer,
340                 processingDest[rootB][arg1], unusedReturnB,
341                 new_payload);
342             rootConnector[rootA].transfer({value:
343                 GRAMS_SEND_UNUSED_RETURN, bounce:true, body:
344                 bodyA});
345             rootConnector[rootB].transfer({value:
346                 GRAMS_SEND_UNUSED_RETURN, bounce:true, body:
347                 bodyB});
348             cleanProcessing(arg1);
349             arg1.transfer({ value: 0, flag: 128});
350         } else if (unusedReturnA > 0) {
351             TvmBuilder builder;
352             builder.store(uint8(7), address(0), address(0));
353             TvmCell new_payload = builder.toCell();
354             TvmCell bodyA = tvml.encodeBody(IDEXConnector(
355                 rootConnector[rootA]).transfer,
356                 processingDest[rootA][arg1], unusedReturnA,
357                 new_payload);
358             rootConnector[rootA].transfer({value:
359                 GRAMS_SEND_UNUSED_RETURN, bounce:true, body:
360                 bodyA});
361             cleanProcessing(arg1);
362             arg1.transfer({ value: 0, flag: 128});
363         } else if (unusedReturnB > 0) {
364             TvmBuilder builder;
365             builder.store(uint8(7), address(0), address(0));
366             TvmCell new_payload = builder.toCell();
367             TvmCell bodyB = tvml.encodeBody(IDEXConnector(
368                 rootConnector[rootB]).transfer,
369                 processingDest[rootB][arg1], unusedReturnB,
370                 new_payload);
371             rootConnector[rootB].transfer({value:
372                 GRAMS_SEND_UNUSED_RETURN, bounce:true, body:
373                 bodyB});

```

```

350         cleanProcessing(arg1);
351         arg1.transfer({ value: 0, flag: 128});
352     } else {
353         cleanProcessing(arg1);
354         arg1.transfer({ value: 0, flag: 128});
355     }
356 } else {
357     TvmBuilder builder;
358     builder.store(uint8(9), address(0), address(0));
359     TvmCell new_payload = builder.toCell();
360     TvmCell bodyA = tvn.encodeBody(IDEXConnector(
361         rootConnector[rootA]).transfer, processingDest[
362         rootA][arg1], amountA, new_payload);
363     TvmCell bodyB = tvn.encodeBody(IDEXConnector(
364         rootConnector[rootB]).transfer, processingDest[
365         rootB][arg1], amountB, new_payload);
366     rootConnector[rootA].transfer({value:
367         GRAMS_SEND_UNUSED_RETURN, bounce:true, body:
368         bodyA});
369     rootConnector[rootB].transfer({value:
370         GRAMS_SEND_UNUSED_RETURN, bounce:true, body:
371         bodyB});
372     cleanProcessing(arg1);
373     arg1.transfer({ value: 0, flag: 128});
374 }
375 } else {
376     arg1.transfer({ value: 0, flag: 128});
377 }
378 }
379 } else {
380     Callback cc = callbacks[counterCallback];
381     cc.token_wallet = token_wallet;
382     cc.token_root = token_root;
383     cc.amount = amount;
384     cc.sender_public_key = sender_public_key;
385     cc.sender_address = sender_address;
386     cc.sender_wallet = sender_wallet;
387     cc.original_gas_to = original_gas_to;
388     cc.updated_balance = updated_balance;
389     TvmSlice slice = payload.toSlice();
390     (uint8 arg0, address arg1, address arg2) = slice.decode(
391         uint8, address, address);
392     cc.payload_arg0 = arg0;
393     cc.payload_arg1 = arg1;
394     cc.payload_arg2 = arg2;
395     callbacks[counterCallback] = cc;
396     counterCallback++;
397     if (arg0 == 1) {
398         tvn.rawReserve(address(this).balance - msg.value, 2);
399         uint128 amountOut = getAmountOut(amount, token_root, arg1
400         );
401         if (!(amountOut > balanceReserve[arg1])){
402             balanceReserve[token_root] += amount;
403             balanceReserve[arg1] -= amountOut;
404             syncStatus[token_root] = balanceReserve[token_root] ==
405             updated_balance ? true : false;

```

```

396         TvmBuilder builder;
397         builder.store(uint8(0), address(0), address(0));
398         TvmCell new_payload = builder.toCell();
399         TvmCell body = tvml.encodeBody(IDEXConnector(
400             rootConnector[arg1]).transfer, arg2, amountOut,
401             new_payload);
402         rootConnector[arg1].transfer({value: 0, bounce:true,
403             flag: 128, body:body});
404     } else {
405         TvmBuilder builder;
406         builder.store(uint8(8), address(0), address(0));
407         TvmCell new_payload = builder.toCell();
408         TvmCell body = tvml.encodeBody(IDEXConnector(
409             rootConnector[token_root]).transfer, token_wallet,
410             amount, new_payload);
411         rootConnector[token_root].transfer({value: 0, bounce:
412             true, flag: 128, body:body});
413     }
414 }
415 if (arg0 == 2) {
416     tvml.rawReserve(address(this).balance - msg.value, 2);
417     processingStatus[token_root][arg1] = true;
418     processingData[token_root][arg1] += amount;
419     processingDest[token_root][arg1] = sender_wallet;
420     if (processingStatus[rootA][arg1] == true &&
421         processingStatus[rootB][arg1] == true) {
422         uint128 amountA = processingData[rootA][arg1];
423         uint128 amountB = processingData[rootB][arg1];
424         if (totalSupply == 0 && balanceReserve[rootA] == 0 &&
425             balanceReserve[rootB] == 0) {
426             uint128 liquidity = math.min(amountA, amountB);
427             balanceReserve[rootA] += amountA;
428             balanceReserve[rootB] += amountB;
429             totalSupply += liquidity;
430             TvmCell body = tvml.encodeBody(IRootTokenContract(
431                 rootAB).mint, liquidity, arg2);
432             rootAB.transfer({value: GRAMS_MINT, bounce:true, body
433                 :body});
434             cleanProcessing(arg1);
435             arg1.transfer({ value: 0, flag: 128});
436         } else {
437             (uint128 provideA, uint128 provideB) =
438                 acceptForProvide(amountA, amountB);
439             if (provideA > 0 && provideB > 0) {
440                 uint128 liquidity = math.min(liquidityA(provideA),
441                     liquidityB(provideB));
442                 uint128 unusedReturnA = amountA - provideA;
443                 uint128 unusedReturnB = amountB - provideB;
444                 balanceReserve[rootA] += provideA;
445                 balanceReserve[rootB] += provideB;
446                 totalSupply += liquidity;
447                 TvmCell body = tvml.encodeBody(IRootTokenContract(
448                     rootAB).mint, liquidity, arg2);
449                 rootAB.transfer({value: GRAMS_MINT, bounce:true,
450                     body:body});
451                 if (unusedReturnA > 0 && unusedReturnB > 0) {
452                     TvmBuilder builder;

```



```

439     builder.store(uint8(7), address(0), address(0));
440     TvmCell new_payload = builder.toCell();
441     TvmCell bodyA = tvn.encodeBody(IDEXConnector(
        rootConnector[rootA]).transfer,
        processingDest[rootA][arg1], unusedReturnA,
        new_payload);
442     TvmCell bodyB = tvn.encodeBody(IDEXConnector(
        rootConnector[rootB]).transfer,
        processingDest[rootB][arg1], unusedReturnB,
        new_payload);
443     rootConnector[rootA].transfer({value:
        GRAMS_SEND_UNUSED_RETURN, bounce:true, body:
        bodyA});
444     rootConnector[rootB].transfer({value:
        GRAMS_SEND_UNUSED_RETURN, bounce:true, body:
        bodyB});
445     cleanProcessing(arg1);
446     arg1.transfer({ value: 0, flag: 128});
447 } else if (unusedReturnA > 0) {
448     TvmBuilder builder;
449     builder.store(uint8(7), address(0), address(0));
450     TvmCell new_payload = builder.toCell();
451     TvmCell bodyA = tvn.encodeBody(IDEXConnector(
        rootConnector[rootA]).transfer,
        processingDest[rootA][arg1], unusedReturnA,
        new_payload);
452     rootConnector[rootA].transfer({value:
        GRAMS_SEND_UNUSED_RETURN, bounce:true, body:
        bodyA});
453     cleanProcessing(arg1);
454     arg1.transfer({ value: 0, flag: 128});
455 } else if (unusedReturnB > 0) {
456     TvmBuilder builder;
457     builder.store(uint8(7), address(0), address(0));
458     TvmCell new_payload = builder.toCell();
459     TvmCell bodyB = tvn.encodeBody(IDEXConnector(
        rootConnector[rootB]).transfer,
        processingDest[rootB][arg1], unusedReturnB,
        new_payload);
460     rootConnector[rootB].transfer({value:
        GRAMS_SEND_UNUSED_RETURN, bounce:true, body:
        bodyB});
461     cleanProcessing(arg1);
462     arg1.transfer({ value: 0, flag: 128});
463 } else {
464     cleanProcessing(arg1);
465     arg1.transfer({ value: 0, flag: 128});
466 }
467 } else {
468     TvmBuilder builder;
469     builder.store(uint8(9), address(0), address(0));
470     TvmCell new_payload = builder.toCell();
471     TvmCell bodyA = tvn.encodeBody(IDEXConnector(
        rootConnector[rootA]).transfer, processingDest[
        rootA][arg1], amountA, new_payload);
472     TvmCell bodyB = tvn.encodeBody(IDEXConnector(
        rootConnector[rootB]).transfer, processingDest[

```

```

473         rootB][arg1], amountB, new_payload);
        rootConnector[rootA].transfer({value:
            GRAMS_SEND_UNUSED_RETURN, bounce:true, body:
            bodyA});
474         rootConnector[rootB].transfer({value:
            GRAMS_SEND_UNUSED_RETURN, bounce:true, body:
            bodyB});
475         cleanProcessing(arg1);
476         arg1.transfer({ value: 0, flag: 128});
477     }
478 }
479 } else {
480     arg1.transfer({ value: 0, flag: 128});
481 }
482 }
483
484 }
485 }
486 }

```

6.9 Internal Method Definitions

6.9.1 Function acceptForProvide

```

214 function acceptForProvide(uint128 arg0, uint128 arg1) private
    inline view returns (uint128, uint128) {
215     require(balanceReserve[rootA] > 0 && balanceReserve[rootB] > 0,
        106);
216     uint128 qtyB = qtyBforA(arg0);
217     uint128 qtyA = qtyAforB(arg1);
218     uint128 minAmountA = math.min(arg0, qtyA);
219     uint128 minAmountB = math.min(arg1, qtyB);
220     uint128 crmin = math.min(balanceReserve[rootA], balanceReserve[
        rootB]);
221     uint128 crmax = math.max(balanceReserve[rootA], balanceReserve[
        rootB]);
222     uint128 crquotient = getQuotient(crmin, crmax);
223     uint128 crremainder = getRemainder(crmin, crmax);
224     uint128 amountMin = math.min(minAmountA, minAmountB);
225     uint128 amountOther = amountMin * crquotient + math.muldiv(
        amountMin, crremainder, crmin);
226     uint128 acceptForProvideA = minAmountA < minAmountB ? amountMin
        : amountOther;
227     uint128 acceptForProvideB = minAmountB < minAmountA ? amountMin
        : amountOther;
228     return (acceptForProvideA, acceptForProvideB);
229 }

```

6.9.2 Function cleanProcessing

```
232 function cleanProcessing(address dexclient) private inline {
233     delete processingStatus[rootA][dexclient];
234     delete processingStatus[rootB][dexclient];
235     delete processingData[rootA][dexclient];
236     delete processingData[rootB][dexclient];
237     delete processingDest[rootA][dexclient];
238     delete processingDest[rootB][dexclient];
239 }
```

6.9.3 Function computeConnectorAddress

```
95 function computeConnectorAddress(uint256 souint) private inline
    view returns (address) {
96     TvmCell stateInit = tvm.buildStateInit({
97         contr: DEXConnector,
98         varInit: { soUINT: souint, dexclient: address(this) },
99         code: codeDEXConnector,
100         pubkey: tvm.pubkey()
101     });
102     return address(tvm.hash(stateInit));
103 }
```

6.9.4 Function connectRoot

```
113 function connectRoot(address root, uint256 souint, uint128
    gramsToConnector, uint128 gramsToRoot) private inline {
114     TvmCell stateInit = tvm.buildStateInit({
115         contr: DEXConnector,
116         varInit: { soUINT: souint, dexclient: address(this) },
117         code: codeDEXConnector,
118         pubkey: tvm.pubkey()
119     });
120     TvmCell init = tvm.encodeBody(DEXConnector);
121     address connector = tvm.deploy(stateInit, init,
        gramsToConnector, address(this).wid);
122     Connector cr = connectors[connector];
```

```

123     cr.root_address = root;
124     cr.souint = souint;
125     cr.status = false;
126     connectors[connector] = cr;
127     TvmCell body = tvm.encodeBody(IDEXConnector(connector).
        deployEmptyWallet, root);
128     connector.transfer({value:gramsToRoot, bounce:true, body:body})
        ;
129 }

```

6.9.5 Function getAmountOut

```

157 function getAmountOut(uint128 amountIn, address rootIn, address
    rootOut) private inline view returns (uint128){
158     uint128 amountInWithFee = math.muldiv(amountIn,997,1);
159     uint128 numerator = math.muldiv(amountInWithFee,balanceReserve[
        rootOut],1);
160     uint128 denominator = amountInWithFee + math.muldiv(
        balanceReserve[rootIn],1000,1);
161     return math.muldiv(1,numerator,denominator);
162 }

```

6.9.6 Function getFirstCallback

```

242 function getFirstCallback() private view returns (uint) {
243     optional(uint, Callback) rc = callbacks.min();
244     if (rc.hasValue()) {(uint number, ) = rc.get();return number;}
        else {return 0;}
245 }

```

6.9.7 Function getQuotient

```

165 function getQuotient(uint128 min, uint128 max) private inline
    pure returns (uint128) {
166     (uint128 quotient, ) = math.muldivmod(1, max, min);
167     return quotient;
168 }

```

6.9.8 Function getRemainder

```
171 function getRemainder(uint128 min, uint128 max) private inline
    pure returns (uint128) {
172     (, uint128 remainder) = math.muldivmod(1, max, min);
173     return remainder;
174 }
```

6.9.9 Function liquidityA

```
191 function liquidityA(uint128 arg0) private inline view returns (
    uint128) {
192     require(arg0 > 0, 105);
193     require(totalSupply > 0, 110);
194     require(balanceReserve[rootA] > 0, 108);
195     return math.muldiv(arg0, totalSupply, balanceReserve[rootA]);
196 }
```

6.9.10 Function liquidityB

```
199 function liquidityB(uint128 arg1) private inline view returns (
    uint128) {
200     require(arg1 > 0, 105);
201     require(totalSupply > 0, 110);
202     require(balanceReserve[rootB] > 0, 109);
203     return math.muldiv(arg1, totalSupply, balanceReserve[rootB]);
204 }
```

6.9.11 Function qtyAforB

```
184 function qtyAforB(uint128 arg1) private inline view returns (
    uint128) {
```

```

185     require(arg1 > 0, 107);
186     require(balanceReserve[rootA] > 0 && balanceReserve[rootB] > 0,
187           106);
187     return math.muldiv(arg1, balanceReserve[rootA], balanceReserve[
188           rootB]);
188 }

```

6.9.12 Function qtyBforA

```

177 function qtyBforA(uint128 arg0) private inline view returns (
178     uint128) {
178     require(arg0 > 0, 105);
179     require(balanceReserve[rootA] > 0 && balanceReserve[rootB] > 0,
180           106);
180     return math.muldiv(arg0, balanceReserve[rootB], balanceReserve[
181           rootA]);
181 }

```

6.9.13 Function thisBalance

```

599 function thisBalance() private inline pure returns (uint128) {
600     return address(this).balance;
601 }

```

Chapter 7

Contract DEXroot

In file `DEXRoot.sol`

7.1 Contract Inheritance

IDEXRoot	
----------	--

7.2 Type Definitions

7.2.1 Struct Pair

```
24 struct Pair {  
25     address root0;  
26     address root1;  
27     address rootLP;  
28 }
```

7.3 Constant Definitions

```
42 uint128 constant public GRAMS_CREATE_DEX_CLIENT = 1 ton;
```

7.4 Static Variable Definitions

```
13 uint256 static public soUINT;
```

7.5 Variable Definitions

```
15 TvmCell public codeDEXclient;
```

```
16 TvmCell public codeDEXpair;
```

```

17  TvmCell public codeDEXconnector;
18  TvmCell public codeRootToken;
19  TvmCell public codeTONTOKENwallet;
21  mapping(address => mapping(address => address)) roots;
30  mapping(address => Pair) public pairs;
31  address[] public pairKeys;
33  mapping(uint256 => address) public pubkeys;
34  mapping(address => uint256) public clients;
35  address[] public clientKeys;
37  mapping(address => uint128) public balanceOf;
38  mapping(uint256 => address) public creators;

```

7.6 Modifier Definitions

7.6.1 Modifier alwaysAccept

```

45  modifier alwaysAccept {
46      tvm.accept();
47      _;
48  }

```

7.6.2 Modifier checkOwnerAndAccept

```

51  modifier checkOwnerAndAccept {
52      require(msg.pubkey() == tvm.pubkey(), 101);
53      tvm.accept();
54      _;
55  }

```

7.6.3 Modifier checkCreatorAndAccept

```

58  modifier checkCreatorAndAccept {
59      require(msg.pubkey() != 0, 103);
60      tvm.accept();
61      _;
62  }

```


7.7 Constructor Definitions

7.7.1 Constructor

```
65     constructor() public {  
66         require(tvm.pubkey() == msg.pubkey(), 102);  
67         tvn.accept();  
68     }
```

7.8 Public Method Definitions

7.8.1 Receive function

```
76     receive() external {  
77         balanceOf[msg.sender] += msg.value;  
78     }
```

7.8.2 Function checkPubKey

```
328     function checkPubKey(uint256 pubkey) public view alwaysAccept  
329         returns (bool status, address dexclient) {  
330         status = pubkeys.exists(pubkey);  
331         dexclient = pubkeys[pubkey];  
332     }
```

7.8.3 Function createDEXclient

```
122     function createDEXclient(uint256 pubkey, uint256 souint) public  
123         alwaysAccept returns (address deployedAddress, bool  
124         statusCreate){
```

```

123     statusCreate = false;
124     deployedAddress = address(0);
125     uint128 prepay = balanceOf[creators[pubkey]];
126     require (!pubkeys.exists(pubkey) && !(prepay <
        GRAMS_CREATE_DEX_CLIENT), 106);
127     delete balanceOf[creators[pubkey]];
128     TvmCell stateInit = tvm.buildStateInit({
129         contr: DEXClient,
130         varInit: {rootDEX: address(this), soUINT: souint,
            codeDEXConnector: codeDEXconnector},
131         code: codeDEXclient,
132         pubkey: pubkey
133     });
134     deployedAddress = new DEXClient{
135         stateInit: stateInit,
136         flag: 0,
137         bounce : false,
138         value : (prepay - 3100000)
139     };
140     pubkeys[pubkey] = deployedAddress;
141     clients[deployedAddress] = pubkey;
142     clientKeys.push(deployedAddress);
143     statusCreate = true;
144 }

```

7.8.4 Function createDEXpair

```

237 function createDEXpair(
238     address root0,
239     address root1,
240     uint256 pairSoArg,
241     uint256 connectorSoArg0,
242     uint256 connectorSoArg1,
243     uint256 rootSoArg,
244     bytes rootName,
245     bytes rootSymbol,
246     uint8 rootDecimals,
247     uint128 gramsForPair,
248     uint128 gramsForRoot,
249     uint128 gramsForConnector,
250     uint128 gramsForWallet
251 ) public override {
252     require(root0 != address(0) && root1 != address(0), 104);
253     require(!(gramsForPair < 500000000) && !(gramsForRoot <
        500000000) && !(gramsForConnector < 500000000) && !(
        gramsForWallet < 500000000), 105);
254     tvm.rawReserve(address(this).balance - msg.value, 2);
255     uint128 gramsNeeded = gramsForPair + (2 * gramsForConnector)
        + (2 * gramsForWallet) + gramsForRoot;
256     if (clients.exists(msg.sender) && !(msg.value < gramsNeeded)
        && !(root0 == root1) && !roots[root0].exists(root1) && !

```

```

257     roots[root1].exists(root0)) {
258     TvmCell stateInitR = tvvm.buildStateInit({
259         contr: RootTokenContract,
260         varInit: {
261             _randomNonce: rootSoArg,
262             name: rootName,
263             symbol: rootSymbol,
264             decimals: rootDecimals,
265             wallet_code: codeTONTTokenWallet
266         },
267         code: codeRootToken,
268         pubkey : clients[msg.sender]
269     });
270     address root01 = address(tvm.hash(stateInitR));
271     TvmCell stateInitP = tvvm.buildStateInit({
272         contr: DEXPair,
273         varInit: {
274             rootDEX: address(this),
275             soUINT: pairSoArg,
276             creator: msg.sender,
277             codeDEXConnector: codeDEXconnector,
278             rootA: root0,
279             rootB: root1,
280             rootAB: root01
281         },
282         code: codeDEXpair,
283         pubkey : clients[msg.sender]
284     });
285     address pairAddress = new DEXPair{
286         stateInit: stateInitP,
287         flag: 0,
288         bounce : false,
289         value : gramsForPair + (2 * gramsForConnector) + (2 *
290             gramsForWallet)
291     }(connectorSoArg0, connectorSoArg1, gramsForConnector,
292         gramsForWallet);
293     address rootAddress = new RootTokenContract{
294         stateInit: stateInitR,
295         flag: 0,
296         bounce : false,
297         value : gramsForRoot
298     }(0, pairAddress);
299     roots[root0][root1] = pairAddress;
300     roots[root1][root0] = pairAddress;
301     Pair cp = pairs[pairAddress];
302     cp.root0 = root0;
303     cp.root1 = root1;
304     cp.rootLP = rootAddress;
305     pairs[pairAddress] = cp;
306     pairKeys.push(pairAddress);
307     msg.sender.transfer({ value: 0, flag: 128});
308 } else {
309     msg.sender.transfer({ value: 0, flag: 128});
310 }
311 }

```

7.8.5 Function getBalanceTONgrams

```
334 function getBalanceTONgrams() public pure alwaysAccept returns (
    uint128 balanceTONgrams){
335     return address(this).balance;
336 }
```

7.8.6 Function getClientAddress

```
118 function getClientAddress(uint256 clientPubKey, uint256
    clientSoArg) public view responsible returns (address) {
119     return { value: 0, bounce: false, flag: 64 }
        computeClientAddress(clientPubKey,clientSoArg);
120 }
```

7.8.7 Function getConnectorAddress

```
233 function getConnectorAddress(uint256 connectorPubKey, uint256
    connectorSoArg, address connectorCommander) public view
    responsible returns (address) {
234     return { value: 0, bounce: false, flag: 64 }
        computeConnectorAddress(connectorPubKey, connectorSoArg,
            connectorCommander);
235 }
```

7.8.8 Function getPairAddress

```
171 function getPairAddress(
172     uint256 pairPubKey,
173     uint256 pairSoArg,
174     address pairCreator,
```

```

175     address pairRootA,
176     address pairRootB,
177     address pairRootAB
178 ) public view responsible returns (address) {
179     return { value: 0, bounce: false, flag: 64 } computePairAddress
        (pairPubKey,pairSoArg,pairCreator,pairRootA,pairRootB,
        pairRootAB);
180 }

```

7.8.9 Function getPairByRoots01

```

314 function getPairByRoots01(address root0, address root1) public
    view alwaysAccept returns (address pairAddr) {
315     pairAddr = roots[root0][root1];
316 }

```

7.8.10 Function getPairByRoots10

```

318 function getPairByRoots10(address root1, address root0) public
    view alwaysAccept returns (address pairAddr) {
319     pairAddr = roots[root1][root0];
320 }

```

7.8.11 Function getRootTokenAddress

```

213 function getRootTokenAddress(
214     uint256 rootPubKey,
215     uint256 rootSoArg,
216     bytes rootName,
217     bytes rootSymbol,
218     uint8 rootDecimals
219 ) public view responsible returns (address) {
220     return { value: 0, bounce: false, flag: 64 }
        computeRootTokenAddress(rootPubKey,rootSoArg,rootName,
        rootSymbol,rootDecimals);
221 }

```

7.8.12 Function getRootsByPair

```
322     function getRootsByPair(address pairAddr) public view
        alwaysAccept returns (address root0, address root1) {
323         Pair cp = pairs[pairAddr];
324         root0 = cp.root0;
325         root1 = cp.root1;
326     }
```

7.8.13 Function sendTransfer

```
71     function sendTransfer(address dest, uint128 value, bool bounce)
        public pure checkOwnerAndAccept {
72         dest.transfer(value, bounce, 0);
73     }
```

7.8.14 Function setCreator

```
100     function setCreator(address giverAddr) public
        checkCreatorAndAccept {
101         uint256 pubkey = msg.pubkey();
102         creators[pubkey] = giverAddr;
103     }
```

7.8.15 Function setDEXclientCode

```
80     function setDEXclientCode(TvmCell code) public
        checkOwnerAndAccept {
81         codeDEXclient = code;
82     }
```

7.8.16 Function setDEXconnectorCode

```
88  function setDEXconnectorCode(TvmCell code) public
      checkOwnerAndAccept {
89      codeDEXconnector = code;
90  }
```

7.8.17 Function setDEXpairCode

```
84  function setDEXpairCode(TvmCell code) public checkOwnerAndAccept
      {
85      codeDEXpair = code;
86  }
```

7.8.18 Function setRootTokenCode

```
92  function setRootTokenCode(TvmCell code) public
      checkOwnerAndAccept {
93      codeRootToken = code;
94  }
```

7.8.19 Function setTONTOKENWalletCode

```
96  function setTONTOKENWalletCode(TvmCell code) public
      checkOwnerAndAccept {
97      codeTONTOKENWallet = code;
98  }
```

7.9 Internal Method Definitions

7.9.1 Function computeClientAddress

```
108 function computeClientAddress(uint256 pubkey, uint256 souint)
    private inline view returns (address) {
109     TvmCell stateInit = tvm.buildStateInit({
110         contr: DEXClient,
111         varInit: {rootDEX:address(this),soUINT:souint,
                    codeDEXConnector:codeDEXconnector},
112         code: codeDEXclient,
113         pubkey: pubkey
114     });
115     return address(tvm.hash(stateInit));
116 }
```

7.9.2 Function computeConnectorAddress

```
223 function computeConnectorAddress(uint256 pubkey, uint256 souint,
    address commander) private inline view returns (address) {
224     TvmCell stateInit = tvm.buildStateInit({
225         contr: DEXConnector,
226         varInit: { soUINT: souint, dexclient: commander },
227         code: codeDEXconnector,
228         pubkey: pubkey
229     });
230     return address(tvm.hash(stateInit));
231 }
```

7.9.3 Function computePairAddress

```
146 function computePairAddress(
147     uint256 pubkey,
148     uint256 souint,
149     address creator,
150     address rootA,
151     address rootB,
```



```

152     address rootAB
153 ) private inline view returns (address){
154     TvmCell stateInit = tvm.buildStateInit({
155         contr: DEXPair,
156         varInit: {
157             rootDEX: address(this),
158             soUINT: souint,
159             creator: creator,
160             codeDEXConnector: codeDEXconnector,
161             rootA: rootA,
162             rootB: rootB,
163             rootAB: rootAB
164         },
165         code: codeDEXpair,
166         pubkey : pubkey
167     });
168     return address(tvm.hash(stateInit));
169 }

```

7.9.4 Function computeRootTokenAddress

```

182 function computeRootTokenAddress(
183     uint256 pubkey,
184     uint256 souint,
185     bytes name,
186     bytes symbol,
187     uint8 decimals
188 ) private inline view returns (address){
189     TvmCell stateInit = tvm.buildStateInit({
190         contr: RootTokenContract,
191         varInit: {
192             _randomNonce: souint,
193             name: name,
194             symbol: symbol,
195             decimals: decimals,
196             wallet_code: codeTONTTokenWallet
197         },
198         code: codeRootToken,
199         pubkey : pubkey
200     });
201     return address(tvm.hash(stateInit));
202 }

```

Chapter 8

Contract RootTokenContract

In file `RootTokenContract.sol`

8.1 Contract Inheritance

IRootTokenContract	
IBurnableTokenRootContract	
IBurnableByRootTokenRootContract	
IPausable	
ITransferOwner	
ISendSurplusGas	
IVersioned	

8.2 Static Variable Definitions

```
28  uint256 static _randomNonce;
```

```
30  bytes public static name;
```

```
31  bytes public static symbol;
```

```
32  uint8 public static decimals;
```

```
34  TvmCell static wallet_code;
```

8.3 Variable Definitions

```
36  uint128 total_supply;
```

```

38     uint256 root_public_key;
39     address root_owner_address;
40     uint128 public start_gas_balance;
42     bool public paused;

```

8.4 Modifier Definitions

8.4.1 Modifier onlyOwner

```

458     modifier onlyOwner() {
459         require(isOwner(), RootTokenContractErrors.
460             error_message_sender_is_not_my_owner);
461     }

```

8.4.2 Modifier onlyInternalOwner

```

463     modifier onlyInternalOwner() {
464         require(isInternalOwner(), RootTokenContractErrors.
465             error_message_sender_is_not_my_owner);
466     }

```

8.5 Constructor Definitions

8.5.1 Constructor

```

48     constructor(uint256 root_public_key_, address
49         root_owner_address_) public {
50         require((root_public_key_ != 0 && root_owner_address_.value
51             == 0) ||
52             (root_public_key_ == 0 && root_owner_address_.value
53                 != 0),
54             RootTokenContractErrors.
55                 error_define_public_key_or_owner_address);
56         tvn.accept();
57         root_public_key = root_public_key_;
58         root_owner_address = root_owner_address_;
59
60         total_supply = 0;
61         paused = false;
62
63         start_gas_balance = address(this).balance;
64     }

```

8.6 Public Method Definitions

8.6.1 Fallback function

```
523     fallback() external {  
524     }
```

8.6.2 OnBounce function

```
514     onBounce(TvmSlice slice) external {  
515         tvml.accept();  
516         uint32 functionId = slice.decode(uint32);  
517         if (functionId == tvml.functionId(ITONTOKENWallet.accept)) {  
518             uint128 latest_bounced_tokens = slice.decode(uint128);  
519             total_supply -= latest_bounced_tokens;  
520         }  
521     }
```

8.6.3 Function deployEmptyWallet

```
237     function deployEmptyWallet(  
238         uint128 deploy_grams,  
239         uint256 wallet_public_key_,  
240         address owner_address_,  
241         address gas_back_address  
242     )  
243     override  
244     external  
245     returns (  
246         address  
247     ) {  
248         require((owner_address_.value != 0 && wallet_public_key_ ==  
249             0) ||  
250             (owner_address_.value == 0 && wallet_public_key_ !=  
251                 0),  
252             RootTokenContractErrors.  
253                 error_define_public_key_or_owner_address);
```

```

251         tvm.rawReserve(address(this).balance - msg.value, 2);
252
253
254         address wallet = new TONTokenWallet{
255             value: deploy_grams,
256             flag: 1,
257             code: wallet_code,
258             pubkey: wallet_public_key_,
259             varInit: {
260                 root_address: address(this),
261                 code: wallet_code,
262                 wallet_public_key: wallet_public_key_,
263                 owner_address: owner_address_
264             }
265         }();
266
267         if (gas_back_address.value != 0) {
268             gas_back_address.transfer({ value: 0, flag: 128 });
269         } else {
270             msg.sender.transfer({ value: 0, flag: 128 });
271         }
272
273         return wallet;
274     }

```

8.6.4 Function deployWallet

```

164     function deployWallet(
165         uint128 tokens,
166         uint128 deploy_grams,
167         uint256 wallet_public_key_,
168         address owner_address_,
169         address gas_back_address
170     )
171     override
172     external
173     onlyOwner
174     returns(
175         address
176     ) {
177         require(tokens >= 0);
178         require((owner_address_.value != 0 && wallet_public_key_ ==
179             0) ||
180             (owner_address_.value == 0 && wallet_public_key_ !=
181             0),
182             RootTokenContractErrors.
183             error_define_public_key_or_owner_address);
184
185         if (root_owner_address.value == 0) {
186             tvm.accept();
187         } else {

```

```

185         tvm.rawReserve(math.max(start_gas_balance, address(this)
186                               ).balance - msg.value), 2);
187     }
188     TvmCell stateInit = tvm.buildStateInit({
189         contr: TONTTokenWallet,
190         varInit: {
191             root_address: address(this),
192             code: wallet_code,
193             wallet_public_key: wallet_public_key_,
194             owner_address: owner_address_,
195         },
196         pubkey: wallet_public_key_,
197         code: wallet_code
198     });
199
200     address wallet;
201
202     if(deploy_grams > 0) {
203         wallet = new TONTTokenWallet{
204             stateInit: stateInit,
205             value: deploy_grams,
206             wid: address(this).wid,
207             flag: 1
208         }();
209     } else {
210         wallet = address(tvm.hash(stateInit));
211     }
212
213     ITONTTokenWallet(wallet).accept(tokens);
214
215     total_supply += tokens;
216
217     if (root_owner_address.value != 0) {
218         if (gas_back_address.value != 0) {
219             gas_back_address.transfer({ value: 0, flag: 128 });
220         } else {
221             msg.sender.transfer({ value: 0, flag: 128 });
222         }
223     }
224
225     return wallet;
226 }

```

8.6.5 Function getDetails

```

77     function getDetails() override external view responsible
78         returns (IRootTokenContractDetails) {
79         return { value: 0, bounce: false, flag: 64 }
80         IRootTokenContractDetails(
81             name,

```

```

80         symbol,
81         decimals,
82         root_public_key,
83         root_owner_address,
84         total_supply
85     );
86 }

```

8.6.6 Function getTotalSupply

```

92     function getTotalSupply() override external view responsible
93         returns (uint128) {
94             return { value: 0, bounce: false, flag: 64 } total_supply;
95         }

```

8.6.7 Function getVersion

```

63     function getVersion() override external pure responsible
64         returns (uint32) {
65             return 4;
66         }

```

8.6.8 Function getWalletAddress

```

111     function getWalletAddress(
112         uint256 wallet_public_key_,
113         address owner_address_
114     )
115     override
116     external
117     view
118     responsible
119     returns (
120         address
121     ) {
122         require((owner_address_.value != 0 && wallet_public_key_ ==
123             0) ||

```

```

123         (owner_address_.value == 0 && wallet_public_key_ !=
124             0),
125         RootTokenContractErrors.
126             error_define_public_key_or_owner_address);
127     return { value: 0, bounce: false, flag: 64 }
128     getExpectedWalletAddress(wallet_public_key_,
129         owner_address_);
130 }

```

8.6.9 Function getWalletCode

```

100     function getWalletCode() override external view responsible
101         returns (TvmCell) {
102         return { value: 0, bounce: false, flag: 64 } wallet_code;
103     }

```

8.6.10 Function mint

```

282     function mint(
283         uint128 tokens,
284         address to
285     )
286     override
287     external
288     onlyOwner
289     {
290         tvm.accept();
291         ITONTTokenWallet(to).accept(tokens);
292         total_supply += tokens;
293     }

```

8.6.11 Function proxyBurn

```

307     function proxyBurn(
308         uint128 tokens,
309         address sender_address,

```



```

310     address send_gas_to,
311     address callback_address,
312     TvmCell callback_payload
313 )
314     override
315     external
316     onlyInternalOwner
317 {
318     tvm.rawReserve(address(this).balance - msg.value, 2);
319
320     address send_gas_to_ = send_gas_to;
321     address expectedWalletAddress = getExpectedWalletAddress(0,
322         sender_address);
323
324     if (send_gas_to.value == 0) {
325         send_gas_to_ = sender_address;
326     }
327
328     IBurnableByRootTokenWallet(expectedWalletAddress).
329         burnByRoot{value: 0, flag: 128}(
330         tokens,
331         send_gas_to_,
332         callback_address,
333         callback_payload
334     );
335 }

```

8.6.12 Function sendExpectedWalletAddress

```

134     function sendExpectedWalletAddress(
135         uint256 wallet_public_key_,
136         address owner_address_,
137         address to
138     )
139     override
140     external
141     {
142         tvm.rawReserve(address(this).balance - msg.value, 2);
143
144         address wallet = getExpectedWalletAddress(
145             wallet_public_key_, owner_address_);
146         IExpectedWalletAddressCallback(to).
147             expectedWalletAddressCallback{value: 0, flag: 128}(
148             wallet,
149             wallet_public_key_,
150             owner_address_
151         );
152     }

```

8.6.13 Function sendPausedCallbackTo

```
423     function sendPausedCallbackTo(  
424         uint64 callback_id,  
425         address callback_addr  
426     )  
427         override  
428         external  
429     {  
430         tvn.rawReserve(address(this).balance - msg.value, 2);  
431         IPausedCallback(callback_addr).pausedCallback{ value: 0,  
            flag: 128 }(callback_id, paused);  
432     }
```

8.6.14 Function sendSurplusGas

```
386     function sendSurplusGas(  
387         address to  
388     )  
389         override  
390         external  
391         onlyInternalOwner  
392     {  
393         tvn.rawReserve(start_gas_balance, 2);  
394         IReceiveSurplusGas(to).receiveSurplusGas{ value: 0, flag:  
            128 }();  
395     }
```

8.6.15 Function setPaused

```
407     function setPaused(  
408         bool value  
409     )  
410         override  
411         external  
412         onlyOwner  
413     {  
414         tvn.accept();
```

```

415     paused = value;
416 }

```

8.6.16 Function tokensBurned

```

347     function tokensBurned(
348         uint128 tokens,
349         uint256 sender_public_key,
350         address sender_address,
351         address send_gas_to,
352         address callback_address,
353         TvmCell callback_payload
354     ) override external {
355
356         require(!paused, RootTokenContractErrors.error_paused);
357
358         address expectedWalletAddress = getExpectedWalletAddress(
359             sender_public_key, sender_address);
360
361         require(msg.sender == expectedWalletAddress,
362             RootTokenContractErrors.
363             error_message_sender_is_not_good_wallet);
364
365         tvmm.rawReserve(address(this).balance - msg.value, 2);
366
367         total_supply -= tokens;
368
369         if (callback_address.value == 0) {
370             send_gas_to.transfer({ value: 0, flag: 128 });
371         } else {
372             IBurnTokensCallback(callback_address).burnCallback({
373                 value: 0, flag: 128}({
374                     tokens,
375                     callback_payload,
376                     sender_public_key,
377                     sender_address,
378                     expectedWalletAddress,
379                     send_gas_to
380                 });
381         }
382     }
383 }

```

8.6.17 Function transferOwner

```

440     function transferOwner(
441         uint256 root_public_key_,
442         address root_owner_address_
443     )
444         override
445         external
446         onlyOwner
447     {
448         require((root_public_key_ != 0 && root_owner_address_.value
449             == 0) ||
450             (root_public_key_ == 0 && root_owner_address_.value
451                 != 0),
452             RootTokenContractErrors.
453                 error_define_public_key_or_owner_address);
454         tvn.accept();
455         root_public_key = root_public_key_;
456         root_owner_address = root_owner_address_;
457     }

```

8.7 Internal Method Definitions

8.7.1 Function getExpectedWalletAddress

```

485     function getExpectedWalletAddress(
486         uint256 wallet_public_key_,
487         address owner_address_
488     )
489         private
490         inline
491         view
492     returns (
493         address
494     ) {
495         TvmCell stateInit = tvn.buildStateInit({
496             contr: TONTOKENWallet,
497             varInit: {
498                 root_address: address(this),
499                 code: wallet_code,
500                 wallet_public_key: wallet_public_key_,
501                 owner_address: owner_address_
502             },
503             pubkey: wallet_public_key_,
504             code: wallet_code
505         });
506
507         return address(tvn.hash(stateInit));
508     }

```

8.7.2 Function isExternalOwner

```
476     function isExternalOwner() private inline view returns (bool) {  
477         return root_public_key != 0 && root_public_key == msg.  
           pubkey();  
478     }
```

8.7.3 Function isInternalOwner

```
472     function isInternalOwner() private inline view returns (bool) {  
473         return root_owner_address.value != 0 && root_owner_address  
           == msg.sender;  
474     }
```

8.7.4 Function isOwner

```
468     function isOwner() private inline view returns (bool) {  
469         return isInternalOwner() || isExternalOwner();  
470     }
```

Chapter 9

Contract TONTokenWallet

In file TONTokenWallet.sol

9.1 Contract Inheritance

ITONTokenWallet	
IDestroyable	
IBurnableByOwnerTokenWallet	
IBurnableByRootTokenWallet	
IVersioned	

9.2 Static Variable Definitions

```
24     address static root_address;  
25     TvmCell static code;  
27     uint256 static wallet_public_key;  
29     address static owner_address;
```

9.3 Variable Definitions

```
31     uint128 balance_;  
32     optional(AllowanceInfo) allowance_;  
34     address receive_callback;  
35     address bounced_callback;
```

```
36     bool allow_non_notifiable;
```

9.4 Modifier Definitions

9.4.1 Modifier onlyRoot

```
598     modifier onlyRoot() {
599         require(root_address == msg.sender, TONTokenWalletErrors.
600             error_message_sender_is_not_my_root);
601     };
```

9.4.2 Modifier onlyOwner

```
603     modifier onlyOwner() {
604         require((owner_address.value != 0 && owner_address == msg.
605             sender) ||
606             (wallet_public_key != 0 && wallet_public_key == msg.
607                 .pubkey()),
608             TONTokenWalletErrors.
609                 error_message_sender_is_not_my_owner);
610     };
```

9.4.3 Modifier onlyInternalOwner

```
610     modifier onlyInternalOwner() {
611         require(owner_address.value != 0 && owner_address == msg.
612             sender);
613     };
```

9.5 Constructor Definitions

9.5.1 Constructor

```
43     constructor() public {
44         require(wallet_public_key == tvn.pubkey() && (owner_address
45             .value == 0 || wallet_public_key == 0));
46         tvn.accept();
47         allow_non_notifiable = true;
48
49         if (owner_address.value != 0) {
50             ITokenWalletDeployedCallback(owner_address).
51                 notifyWalletDeployed{value: 0.00001 ton, flag: 1}(
52                     root_address);
53     }
```

```

51     }
52 }

```

9.6 Public Method Definitions

9.6.1 Fallback function

```

683     fallback() external {
684     }

```

9.6.2 OnBounce function

```

653     onBounce(TvmSlice body) external {
654         tvvm.accept();
655
656         uint32 functionId = body.decode(uint32);
657         if (functionId == tvvm.functionId(ITONTOKENWallet.
            internalTransfer)) {
658             uint128 tokens = body.decode(uint128);
659             balance_ += tokens;
660
661             if (bounced_callback.value != 0) {
662                 tvvm.rawReserve(address(this).balance - msg.value,
                    2);
663                 ITokensBouncedCallback(bounced_callback).
                    tokensBouncedCallback( value: 0, flag: 128 )(
664                     address(this),
665                     root_address,
666                     tokens,
667                     msg.sender,
668                     balance_
669                 );
670             } else if (owner_address.value != 0) {
671                 tvvm.rawReserve(math.max(TONTOKENWalletConstants.
                    target_gas_balance, address(this).balance - msg
672                     .value), 2);
673                 owner_address.transfer({ value: 0, flag: 128 });
674             }
675             } else if (functionId == tvvm.functionId(
                IBurnableTokenRootContract.tokensBurned)) {
676                 balance_ += body.decode(uint128);
677                 if (owner_address.value != 0) {

```



```

677         tvm.rawReserve(math.max(TONTOKENWalletConstants.
        target_gas_balance, address(this).balance - msg
        .value), 2);
678         owner_address.transfer({ value: 0, flag: 128 });
679     }
680 }
681

```

9.6.3 Function accept

```

96     function accept(
97         uint128 tokens
98     )
99     override
100     external
101     onlyRoot
102     {
103         tvm.accept();
104         balance_ += tokens;
105     }

```

9.6.4 Function allowance

```

107     function allowance() override external view responsible returns
        (AllowanceInfo) {
108         return { value: 0, bounce: false, flag: 64 } (allowance_.
            hasValue() ? allowance_.get() : AllowanceInfo(0,
            address.makeAddrStd(0, 0)));
109     }

```

9.6.5 Function approve

```

119     function approve(
120         address spender,
121         uint128 remaining_tokens,
122         uint128 tokens
123     )
124     override

```

```

125     external
126     onlyOwner
127     {
128         require(remaining_tokens == 0 || !allowance_.hasValue(),
129                 TONTokenWalletErrors.error_non_zero_remaining);
130         if (owner_address.value != 0 ) {
131             tvml.rawReserve(math.max(TONTokenWalletConstants.
132                                     target_gas_balance, address(this).balance - msg.
133                                     value), 2);
134         } else {
135             tvml.accept();
136         }
137
138         if (allowance_.hasValue()) {
139             if (allowance_.get().remaining_tokens ==
140                 remaining_tokens) {
141                 allowance_.set(AllowanceInfo(tokens, spender));
142             }
143         } else {
144             allowance_.set(AllowanceInfo(tokens, spender));
145         }
146
147         if (owner_address.value != 0 ) {
148             msg.sender.transfer({ value: 0, flag: 128 });
149         }
150     }

```

9.6.6 Function balance

```

58     function balance() override external view responsible returns (
59         uint128) {
60         return { value: 0, bounce: false, flag: 64 } balance_;
61     }

```

9.6.7 Function burnByOwner

```

473     function burnByOwner(
474         uint128 tokens,
475         uint128 grams,
476         address send_gas_to,
477         address callback_address,
478         TvmCell callback_payload
479     ) override external onlyOwner {
480         require(tokens > 0);

```

```

481         require(tokens <= balance_, TONTokenWalletErrors.
482             error_not_enough_balance);
483         require((owner_address.value != 0 && msg.value > 0) ||
484             (owner_address.value == 0 && grams <= address(this)
485                 .balance && grams > 0), TONTokenWalletErrors.
486                 error_low_message_value);
487
488         if (owner_address.value != 0 ) {
489             tvmm.rawReserve(math.max(TONTokenWalletConstants.
490                 target_gas_balance, address(this).balance - msg.
491                 value), 2);
492             balance_ -= tokens;
493             IBurnableTokenRootContract(root_address)
494                 .tokensBurned{ value: 0, flag: 128, bounce: true }(
495                 tokens,
496                 wallet_public_key,
497                 owner_address,
498                 send_gas_to.value != 0 ? send_gas_to :
499                     owner_address,
500                 callback_address,
501                 callback_payload
502             );
503         } else {
504             tvmm.accept();
505             balance_ -= tokens;
506             IBurnableTokenRootContract(root_address)
507                 .tokensBurned{ value: grams, bounce: true }(
508                 tokens,
509                 wallet_public_key,
510                 owner_address,
511                 send_gas_to.value != 0 ? send_gas_to : address(
512                     this),
513                 callback_address,
514                 callback_payload
515             );
516         }
517     }
518 }

```

9.6.8 Function burnByRoot

```

520     function burnByRoot(
521         uint128 tokens,
522         address send_gas_to,
523         address callback_address,
524         TvmCell callback_payload
525     ) override external onlyRoot {
526         require(tokens > 0);
527         require(tokens <= balance_, TONTokenWalletErrors.
528             error_not_enough_balance);
529
530         tvmm.rawReserve(address(this).balance - msg.value, 2);

```

```

530
531     balance_ -= tokens;
532
533     IBurnableTokenRootContract(root_address)
534         .tokensBurned{ value: 0, flag: 128, bounce: true }(
535         tokens,
536         wallet_public_key,
537         owner_address,
538         send_gas_to,
539         callback_address,
540         callback_payload
541     );
542 }

```

9.6.9 Function destroy

```

584     function destroy(
585         address gas_dest
586     )
587         override
588         public
589         onlyOwner
590     {
591         require(balance_ == 0);
592         tvml.accept();
593         selfdestruct(gas_dest);
594     }

```

9.6.10 Function disapprove

```

148     function disapprove() override external onlyOwner {
149         if (owner_address.value != 0 ) {
150             tvml.rawReserve(math.max(TONTokenWalletConstants.
151                                     target_gas_balance, address(this).balance - msg.
152                                     value), 2);
153         } else {
154             tvml.accept();
155         }
156
157         allowance_.reset();
158
159         if (owner_address.value != 0 ) {
160             msg.sender.transfer({ value: 0, flag: 128 });
161         }
162     }

```

9.6.11 Function getDetails

```
72     function getDetails() override external view responsible
73         returns (ITONTOKENWalletDetails) {
74         return { value: 0, bounce: false, flag: 64 }
75             ITONTOKENWalletDetails(
76                 root_address,
77                 wallet_public_key,
78                 owner_address,
79                 balance_,
80                 receive_callback,
81                 bounced_callback,
82                 allow_non_notifiable
83             );
84     }
```

9.6.12 Function getVersion

```
54     function getVersion() override external pure responsible
55         returns (uint32) {
56         return 4;
57     }
```

9.6.13 Function getWalletCode

```
87     function getWalletCode() override external view responsible
88         returns (TvmCell) {
89         return { value: 0, bounce: false, flag: 64 } code;
90     }
```

9.6.14 Function internalTransfer

```

370     function internalTransfer(
371         uint128 tokens,
372         uint256 sender_public_key,
373         address sender_address,
374         address send_gas_to,
375         bool notify_receiver,
376         TvmCell payload
377     )
378     override
379     external
380     {
381         require(notify_receiver || allow_non_notifiable ||
382             receive_callback.value == 0,
383             TONTOKENWalletErrors.
384                 error_recipient_has_disallow_non_notifiable);
385         address expectedSenderAddress = getExpectedAddress(
386             sender_public_key, sender_address);
387         require(msg.sender == expectedSenderAddress,
388             TONTOKENWalletErrors.
389                 error_message_sender_is_not_good_wallet);
390         require(sender_address != owner_address ||
391             sender_public_key != wallet_public_key,
392             TONTOKENWalletErrors.error_wrong_recipient);
393
394         if (owner_address.value != 0 ) {
395             uint128 reserve = math.max(TONTOKENWalletConstants.
396                 target_gas_balance, address(this).balance - msg.
397                 value);
398             require(address(this).balance > reserve,
399                 TONTOKENWalletErrors.error_low_message_value);
400             tvmm.rawReserve(reserve, 2);
401         } else {
402             tvmm.rawReserve(address(this).balance - msg.value, 2);
403         }
404
405         balance_ += tokens;
406
407         if (notify_receiver && receive_callback.value != 0) {
408             ITokenReceivedCallback(receive_callback).
409                 tokensReceivedCallback({ value: 0, flag: 128 })(
410                 address(this),
411                 root_address,
412                 tokens,
413                 sender_public_key,
414                 sender_address,
415                 msg.sender,
416                 send_gas_to,
417                 balance_,
418                 payload
419             );
420         } else {
421             send_gas_to.transfer({ value: 0, flag: 128 });
422         }
423     }

```

9.6.15 Function internalTransferFrom

```
423     function internalTransferFrom(  
424         address to,  
425         uint128 tokens,  
426         address send_gas_to,  
427         bool notify_receiver,  
428         TvmCell payload  
429     )  
430     override  
431     external  
432     {  
433         require(allowance_.hasValue(), TONTokenWalletErrors.  
434             error_no_allowance_set);  
435         require(msg.sender == allowance_.get().spender,  
436             TONTokenWalletErrors.error_wrong_spender);  
437         require(tokens <= allowance_.get().remaining_tokens,  
438             TONTokenWalletErrors.error_not_enough_allowance);  
439         require(tokens <= balance_, TONTokenWalletErrors.  
440             error_not_enough_balance);  
441         require(tokens > 0);  
442         require(to != address(this), TONTokenWalletErrors.  
443             error_wrong_recipient);  
444  
445         if (owner_address.value != 0 ) {  
446             uint128 reserve = math.max(TONTokenWalletConstants.  
447                 target_gas_balance, address(this).balance - msg.  
448                 value);  
449             require(address(this).balance > reserve +  
450                 TONTokenWalletConstants.target_gas_balance,  
451                 TONTokenWalletErrors.error_low_message_value);  
452             tvn.rawReserve(reserve, 2);  
453             tvn.rawReserve(math.max(TONTokenWalletConstants.  
454                 target_gas_balance, address(this).balance - msg.  
455                 value), 2);  
456         } else {  
457             require(msg.value > TONTokenWalletConstants.  
458                 target_gas_balance, TONTokenWalletErrors.  
459                 error_low_message_value);  
460             tvn.rawReserve(address(this).balance - msg.value, 2);  
461         }  
462  
463         balance_ -= tokens;  
464  
465         allowance_.set(AllowanceInfo(allowance_.get().  
466             remaining_tokens - tokens, allowance_.get().spender));  
467  
468         ITONTokenWallet(to).internalTransfer{ value: 0, bounce:  
469             true, flag: 129 }(tokens,  
470             wallet_public_key,
```

```

457         owner_address,
458         send_gas_to,
459         notify_receiver,
460         payload
461     );
462 }

```

9.6.16 Function setBouncedCallback

```

568     function setBouncedCallback(
569         address bounced_callback_
570     )
571     override
572     external
573     onlyOwner
574     {
575         tvn.accept();
576         bounced_callback = bounced_callback_;
577     }

```

9.6.17 Function setReceiveCallback

```

550     function setReceiveCallback(
551         address receive_callback_,
552         bool allow_non_notifiable_
553     )
554     override
555     external
556     onlyOwner
557     {
558         tvn.accept();
559         receive_callback = receive_callback_;
560         allow_non_notifiable = allow_non_notifiable_;
561     }

```

9.6.18 Function transfer

```

262     function transfer(

```



```

263     address to,
264     uint128 tokens,
265     uint128 grams,
266     address send_gas_to,
267     bool notify_receiver,
268     TvmCell payload
269 ) override external onlyOwner {
270     require(tokens > 0);
271     require(tokens <= balance_, TONTOKENWalletErrors.
272         error_not_enough_balance);
273     require(to.value != 0, TONTOKENWalletErrors.
274         error_wrong_recipient);
275     require(to != address(this), TONTOKENWalletErrors.
276         error_wrong_recipient);
277
278     if (owner_address.value != 0 ) {
279         uint128 reserve = math.max(TONTOKENWalletConstants.
280             target_gas_balance, address(this).balance - msg.
281             value);
282         require(address(this).balance > reserve +
283             TONTOKENWalletConstants.target_gas_balance,
284             TONTOKENWalletErrors.error_low_message_value);
285         tvmm.rawReserve(reserve, 2);
286         balance_ -= tokens;
287
288         ITONTOKENWallet(to).internalTransfer{ value: 0, flag:
289             129, bounce: true }(
290             tokens,
291             wallet_public_key,
292             owner_address,
293             send_gas_to.value != 0 ? send_gas_to :
294                 owner_address,
295             notify_receiver,
296             payload
297         );
298     } else {
299         require(address(this).balance > grams,
300             TONTOKENWalletErrors.error_low_message_value);
301         require(grams > TONTOKENWalletConstants.
302             target_gas_balance, TONTOKENWalletErrors.
303             error_low_message_value);
304         tvmm.accept();
305         balance_ -= tokens;
306
307         ITONTOKENWallet(to).internalTransfer{ value: grams,
308             bounce: true, flag: 1 }(
309             tokens,
310             wallet_public_key,
311             owner_address,
312             send_gas_to.value != 0 ? send_gas_to : address(this
313             ),
314             notify_receiver,
315             payload
316         );
317     }
318 }

```

9.6.19 Function transferFrom

```
317     function transferFrom(  
318         address from,  
319         address to,  
320         uint128 tokens,  
321         uint128 grams,  
322         address send_gas_to,  
323         bool notify_receiver,  
324         TvmCell payload  
325     )  
326     override  
327     external  
328     onlyOwner  
329     {  
330         require(to.value != 0, TONTokenWalletErrors.  
331             error_wrong_recipient);  
332         require(tokens > 0);  
333         require(from != to, TONTokenWalletErrors.  
334             error_wrong_recipient);  
335  
336         if (owner_address.value != 0 ) {  
337             uint128 reserve = math.max(TONTokenWalletConstants.  
338                 target_gas_balance, address(this).balance - msg.  
339                 value);  
340             require(address(this).balance > reserve + (  
341                 TONTokenWalletConstants.target_gas_balance * 2),  
342                 TONTokenWalletErrors.error_low_message_value);  
343             tvn.rawReserve(reserve, 2);  
344  
345             ITONTokenWallet(from).internalTransferFrom{ value: 0,  
346                 flag: 129 }(to,  
347                     tokens,  
348                     send_gas_to.value != 0 ? send_gas_to :  
349                         owner_address,  
350                     notify_receiver,  
351                     payload  
352                 );  
353         } else {  
354             require(address(this).balance > grams,  
355                 TONTokenWalletErrors.error_low_message_value);  
356             require(grams > TONTokenWalletConstants.  
357                 target_gas_balance * 2, TONTokenWalletErrors.  
358                 error_low_message_value);  
359             tvn.accept();  
360             ITONTokenWallet(from).internalTransferFrom{ value:  
361                 grams, flag: 1 }(to,  
362                     tokens,  
363                     send_gas_to.value != 0 ? send_gas_to : address(this
```

```

354         ),
355         notify_receiver,
356         payload
357     );
358 }

```

9.6.20 Function transferToRecipient

```

177 function transferToRecipient(
178     uint256 recipient_public_key,
179     address recipient_address,
180     uint128 tokens,
181     uint128 deploy_grams,
182     uint128 transfer_grams,
183     address send_gas_to,
184     bool notify_receiver,
185     TvmCell payload
186 ) override external onlyOwner {
187     require(tokens > 0);
188     require(tokens <= balance_, TONTokenWalletErrors.
189         error_not_enough_balance);
189     require(recipient_address.value == 0 ||
190         recipient_public_key == 0, TONTokenWalletErrors.
191         error_wrong_recipient);
192
193     if (owner_address.value != 0 ) {
194         uint128 reserve = math.max(TONTokenWalletConstants.
195             target_gas_balance, address(this).balance - msg.
196             value);
197         require(address(this).balance > reserve +
198             TONTokenWalletConstants.target_gas_balance +
199             deploy_grams, TONTokenWalletErrors.
200             error_low_message_value);
201         require(recipient_address != owner_address,
202             TONTokenWalletErrors.error_wrong_recipient);
203         tvml.rawReserve(reserve, 2);
204     } else {
205         require(address(this).balance > deploy_grams +
206             transfer_grams, TONTokenWalletErrors.
207             error_low_message_value);
208         require(transfer_grams > TONTokenWalletConstants.
209             target_gas_balance, TONTokenWalletErrors.
210             error_low_message_value);
211         require(recipient_public_key != wallet_public_key);
212         tvml.accept();
213     }
214
215     TvmCell stateInit = tvml.buildStateInit({
216         contr: TONTokenWallet,
217         varInit: {

```

```

206         root_address: root_address,
207         code: code,
208         wallet_public_key: recipient_public_key,
209         owner_address: recipient_address
210     },
211     pubkey: recipient_public_key,
212     code: code
213 });
214
215 address to;
216
217 if(deploy_grams > 0) {
218     to = new TONTOKENWallet{
219         stateInit: stateInit,
220         value: deploy_grams,
221         wid: address(this).wid,
222         flag: 1
223     };
224 } else {
225     to = address(tvm.hash(stateInit));
226 }
227
228 if (owner_address.value != 0 ) {
229     balance_ -= tokens;
230     ITONTOKENWallet(to).internalTransfer{ value: 0, flag:
231         129, bounce: true }(
232         tokens,
233         wallet_public_key,
234         owner_address,
235         send_gas_to.value != 0 ? send_gas_to :
236             owner_address,
237         notify_receiver,
238         payload
239     );
240 } else {
241     balance_ -= tokens;
242     ITONTOKENWallet(to).internalTransfer{ value:
243         transfer_grams, flag: 1, bounce: true }(
244         tokens,
245         wallet_public_key,
246         owner_address,
247         send_gas_to.value != 0 ? send_gas_to : address(this
248             ),
249         notify_receiver,
250         payload
251     );
252 }
253 }

```

9.7 Internal Method Definitions

9.7.1 Function getExpectedAddress

```
620     function getExpectedAddress(  
621         uint256 wallet_public_key_,  
622         address owner_address_  
623     )  
624         private  
625         inline  
626         view  
627     returns (   
628         address  
629     ) {  
630         TvmCell stateInit = tvm.buildStateInit({  
631             contr: TONTOKENWallet,  
632             varInit: {  
633                 root_address: root_address_,  
634                 code: code,  
635                 wallet_public_key: wallet_public_key_,  
636                 owner_address: owner_address_  
637             },  
638             pubkey: wallet_public_key_,  
639             code: code  
640         });  
641  
642         return address(tvm.hash(stateInit));  
643     }
```