

Audit of the Radiance DEX Contracts

Contact: Fabrice LE FESSANT
Telegram: @fabrice_dune
OCamlPro

August 6, 2021

Contents

1	Introduction	9
2	Overview	11
2.1	Code Architecture	11
2.2	Contracts Architecture	12
2.3	Sequences of Messages	12
2.3.1	<code>createDEXclient</code> sequence	12
2.3.2	<code>createDEXpair</code> sequence	12
2.3.3	<code>DEXConnector</code> deployment sequence	13
2.3.4	<code>connectPair</code> sequence	13
2.3.5	<code>processSwapA</code> sequence	13
2.3.6	<code>processLiquidity</code> sequence	14
2.3.7	<code>returnLiquidity</code> sequence	14
3	General Remarks	16
3.1	Readability	16
3.1.1	Typography of Static Variables	16
3.1.2	Typography of Global Variables	16
3.1.3	Typography of Internal Functions	16
3.1.4	Naming of Numbers	16
3.1.5	Better Units for Big Numbers	17
3.1.6	Use Method Calls instead of <code>tvm.encodeBody</code>	17
3.2	Gas Consumption	17
3.2.1	Accept Methods without Checks	17
3.2.2	<code>require</code> after <code>tvm.accept</code>	17
3.2.3	Not Enough Gas for Action	17
3.3	Architecture	18
3.3.1	No need for passing <code>souint</code> Arguments around	18
4	Contract DEXClient	19
4.1	Contract Inheritance	20
4.2	Type Definitions	20
4.2.1	Struct Connector	20
4.2.2	Struct Callback	20

4.2.3	Struct Pair	21
4.3	Constant Definitions	21
4.4	Static Variable Definitions	21
4.5	Variable Definitions	22
4.6	Modifier Definitions	22
4.6.1	Modifier alwaysAccept	22
4.6.2	Modifier checkOwnerAndAccept	22
4.7	Constructor Definitions	23
4.7.1	Constructor	23
4.8	Public Method Definitions	23
4.8.1	Receive function	23
4.8.2	Function connectCallback	23
4.8.3	Function connectPair	24
4.8.4	Function connectRoot	24
4.8.5	Function createNewPair	25
4.8.6	Function getAllDataPreparation	26
4.8.7	Function getBalance	26
4.8.8	Function getCallback	26
4.8.9	Function getConnectorAddress	27
4.8.10	Function getPairData	27
4.8.11	Function processLiquidity	27
4.8.12	Function processSwapA	28
4.8.13	Function processSwapB	29
4.8.14	Function returnLiquidity	29
4.8.15	Function sendTokens	30
4.8.16	Function setPair	31
4.8.17	Function tokensReceivedCallback	31
4.9	Internal Method Definitions	32
4.9.1	Function computeConnectorAddress	32
4.9.2	Function getFirstCallback	33
4.9.3	Function getQuotient	33
4.9.4	Function getRemainder	33
4.9.5	Function isReady	33
4.9.6	Function isReadyToProvide	34
4.9.7	Function thisBalance	34
5	Contract DEXConnector	35
5.1	Contract Inheritance	35
5.2	Constant Definitions	36
5.3	Static Variable Definitions	36
5.4	Variable Definitions	36
5.5	Modifier Definitions	37
5.5.1	Modifier alwaysAccept	37
5.5.2	Modifier checkOwnerAndAccept	37
5.6	Constructor Definitions	37
5.6.1	Constructor	37

5.7	Public Method Definitions	37
5.7.1	Receive function	37
5.7.2	Function burn	38
5.7.3	Function deployEmptyWallet	38
5.7.4	Function expectedWalletAddressCallback	39
5.7.5	Function getBalance	39
5.7.6	Function setBouncedCallback	39
5.7.7	Function setTransferCallback	40
5.7.8	Function transfer	40
5.8	Internal Method Definitions	41
5.8.1	Function getQuotient	41
5.8.2	Function getRemainder	41
6	Contract DEXPair	42
6.1	Contract Inheritance	43
6.2	Type Definitions	43
6.2.1	Struct Connector	43
6.2.2	Struct Callback	43
6.3	Constant Definitions	44
6.4	Static Variable Definitions	44
6.5	Variable Definitions	45
6.6	Modifier Definitions	45
6.6.1	Modifier alwaysAccept	45
6.6.2	Modifier checkOwnerAndAccept	46
6.6.3	Modifier checkPubKeyAndAccept	46
6.7	Constructor Definitions	46
6.7.1	Constructor	46
6.8	Public Method Definitions	46
6.8.1	Receive function	46
6.8.2	Function burnCallback	47
6.8.3	Function connect	48
6.8.4	Function connectCallback	48
6.8.5	Function getBalance	49
6.8.6	Function getCallback	49
6.8.7	Function getConnectorAddress	50
6.8.8	Function tokensReceivedCallback	50
6.9	Internal Method Definitions	54
6.9.1	Function acceptForProvide	54
6.9.2	Function cleanProcessing	55
6.9.3	Function computeConnectorAddress	55
6.9.4	Function connectRoot	55
6.9.5	Function getAmountOut	56
6.9.6	Function getFirstCallback	57
6.9.7	Function getQuotient	57
6.9.8	Function getRemainder	57
6.9.9	Function liquidityA	58

6.9.10	Function liquidityB	58
6.9.11	Function qtyAforB	58
6.9.12	Function qtyBforA	59
6.9.13	Function thisBalance	59
7	Contract DEXroot	60
7.1	Contract Inheritance	61
7.2	Type Definitions	61
7.2.1	Struct Pair	61
7.3	Constant Definitions	61
7.4	Static Variable Definitions	61
7.5	Variable Definitions	62
7.6	Modifier Definitions	62
7.6.1	Modifier alwaysAccept	62
7.6.2	Modifier checkOwnerAndAccept	63
7.6.3	Modifier checkCreatorAndAccept	63
7.7	Constructor Definitions	63
7.7.1	Constructor	63
7.8	Public Method Definitions	63
7.8.1	Receive function	63
7.8.2	Function checkPubKey	64
7.8.3	Function createDEXclient	64
7.8.4	Function createDEXpair	65
7.8.5	Function getBalanceTONgrams	67
7.8.6	Function getClientAddress	67
7.8.7	Function getConnectorAddress	67
7.8.8	Function getPairAddress	67
7.8.9	Function getPairByRoots01	68
7.8.10	Function getPairByRoots10	68
7.8.11	Function getRootTokenAddress	68
7.8.12	Function getRootsByPair	69
7.8.13	Function sendTransfer	69
7.8.14	Function setCreator	69
7.8.15	Function setDEXclientCode	69
7.8.16	Function setDEXconnectorCode	70
7.8.17	Function setDEXpairCode	70
7.8.18	Function setRootTokenCode	70
7.8.19	Function setTONTokenWalletCode	71
7.9	Internal Method Definitions	71
7.9.1	Function computeClientAddress	71
7.9.2	Function computeConnectorAddress	71
7.9.3	Function computePairAddress	72
7.9.4	Function computeRootTokenAddress	72

8	Contract RootTokenContract	74
8.1	Contract Inheritance	75
8.2	Static Variable Definitions	75
8.3	Variable Definitions	75
8.4	Modifier Definitions	75
8.4.1	Modifier onlyOwner	75
8.4.2	Modifier onlyInternalOwner	76
8.5	Constructor Definitions	76
8.5.1	Constructor	76
8.6	Public Method Definitions	76
8.6.1	Fallback function	76
8.6.2	OnBounce function	77
8.6.3	Function deployEmptyWallet	77
8.6.4	Function deployWallet	78
8.6.5	Function getDetails	79
8.6.6	Function getTotalSupply	80
8.6.7	Function getVersion	80
8.6.8	Function getWalletAddress	80
8.6.9	Function getWalletCode	81
8.6.10	Function mint	81
8.6.11	Function proxyBurn	81
8.6.12	Function sendExpectedWalletAddress	82
8.6.13	Function sendPausedCallbackTo	82
8.6.14	Function sendSurplusGas	83
8.6.15	Function setPaused	83
8.6.16	Function tokensBurned	83
8.6.17	Function transferOwner	84
8.7	Internal Method Definitions	85
8.7.1	Function getExpectedWalletAddress	85
8.7.2	Function isExternalOwner	85
8.7.3	Function isInternalOwner	86
8.7.4	Function isOwner	86
9	Contract TONTokenWallet	87
9.1	Contract Inheritance	88
9.2	Static Variable Definitions	88
9.3	Variable Definitions	88
9.4	Modifier Definitions	88
9.4.1	Modifier onlyRoot	88
9.4.2	Modifier onlyOwner	89
9.4.3	Modifier onlyInternalOwner	89
9.5	Constructor Definitions	89
9.5.1	Constructor	89
9.6	Public Method Definitions	89
9.6.1	Fallback function	89
9.6.2	OnBounce function	90

9.6.3	Function accept	90
9.6.4	Function allowance	91
9.6.5	Function approve	91
9.6.6	Function balance	92
9.6.7	Function burnByOwner	92
9.6.8	Function burnByRoot	93
9.6.9	Function destroy	94
9.6.10	Function disapprove	94
9.6.11	Function getDetails	94
9.6.12	Function getVersion	95
9.6.13	Function getWalletCode	95
9.6.14	Function internalTransfer	95
9.6.15	Function internalTransferFrom	96
9.6.16	Function setBouncedCallback	98
9.6.17	Function setReceiveCallback	98
9.6.18	Function transfer	98
9.6.19	Function transferFrom	100
9.6.20	Function transferToRecipient	101
9.7	Internal Method Definitions	103
9.7.1	Function getExpectedAddress	103

Table of Critical and Major Issues

Major issue: Accept-All Modifier in DEXClient	22
Major issue: Accept-All Method in DEXClient.connectCallback	23
Major issue: Accept-All Method in DEXClient.getAllDataPreparation	26
Major issue: Accept-All Method in DEXClient.getPairData . .	27
Major issue: Accept-All Method in DEXClient.setPair	31
Major issue: Accept-All Method in DEXClient.tokensReceivedCallback	31
Major issue: Accept-All Modifier in DEXConnector	37
Major issue: Accept-All Modifier in DEXPair	45
Critical issue: Unlimited Growth of callbacks	47
Major issue: Accept-All Method in DEXPair.burnCallback . . .	47
Major issue: Accept-All Method in DEXPair.connectCallback .	48
Critical issue: Swap Refund does not work in DEXPair.tokensReceivedCallback	50
Major issue: Accept-All Method in DEXPair.tokensReceivedCallback	50
Major issue: Accept-All Modifier in DEXroot	62
Major issue: Accept-All Modifier in DEXroot	63
Major issue: Accept-All Method in DEXroot.checkPubKey . . .	64
Major issue: Accept-All Method in DEXroot.createDEXclient .	64
Critical issue: Ownership of Liquidity Root Contract	65
Major issue: Accept-All Method in DEXroot.getBalanceTONgrams	67
Major issue: Accept-All Method in DEXroot.getPairByRoots01	68
Major issue: Accept-All Method in DEXroot.getPairByRoots10	68
Major issue: Accept-All Method in DEXroot.getRootsByPair .	69
Critical issue: Impersonate Accounts in DEXroot.setCreator .	69

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%3A07783c48e8789fa1163699e9e3071a4791728>

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.

During this audit, we classified our findings into three kinds of issues:

Critical Issues: such issues can lead to taking ownership of resources, or total disabling of the service;

Major Issues: such issues can lead to a decrease in the quality of the service, or temporary loss of availability;

Minor Issues: Such issues do not impact the service itself. For example, code improvements to improve readability, to improve sharing, etc.

In these three kinds of issues, we found:

3 Critical issues: one is a data structure with unlimited growth; one is an attacker taking ownership of the liquidity root contract; one is the possibility for an attacker to impersonate another user to create a client contract with its balance;

Many Major issues: all of them are related to the use of `tvm.accept`, allowing attackers to drain the balance of contracts. However, as the contracts are not expected to keep large amounts of TONs, we classified these attacks in the “Major issues” kind, as their impact will only be a loss of quality of service;

Minor issues: most of them are readability improvements;

Critical issues and major issues are listed in the “Table of Critical and Major Issues” before this chapter. **All critical issues and most major issues have been submitted to the Radiance team developers** on the Telegram channel above.

Chapter 2

Overview

2.1 Code Architecture

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.

2.2 Contracts Architecture

The DEX is organized around the `DEXroot` contract. The `DEXroot` contract performs mainly two operations, `createDEXclient` to create a new client contract for an external user, and `createDEXpair` for a client to create a new pair contract.

Once the some users have created some `DEXPair` for some tokens using their `DEXClient` contract, using the `createNewPair` public method, users can use the `connectPair` public method to register the pair in the client. Users should also use `connectRoot` public methods to associate `DEXConnector` contracts to each token of the `DEXPair`, i.e. a wallet associated to the two tokens of the pair, plus the liquidity token.

Once a user has registered his client to the pair and created the corresponding connectors, he may swap tokens using the `processSwapA` and `processSwapB` public methods, provide liquidities using the `processLiquidity` public method, or refund his liquidities using the `returnLiquidity` public method.

2.3 Sequences of Messages

2.3.1 `createDEXclient` sequence

- The user transfers some TONs to the `DEXroot` contract from his multisig;
- The user sends a `setCreator` message to the `DEXroot` contract to claim ownership for the former transferred tokens;
- The user sends a `createDEXclient` message to the `DEXroot` contract;
- The `DEXroot` contract deploys the `DEXClient` contract;

2.3.2 `createDEXpair` sequence

- The user sends a `createNewPair` message to his `DEXClient` contract, providing the two root token contracts to be swapped;
- The client contract sends a `createDEXpair` message to the `DEXroot` contract.
- The root contract:
 - Deploys a `RootTokenContract` contract to create the token later used for liquidity management;
 - Deploys a `DEXPair` contract for the pair;
- The `DEXPair` constructor deploys two `DEXConnector` contracts, one for each token of the pair (see the `DEXConnector` deployment sequence)

2.3.3 DEXConnector deployment sequence

- A contract (DEXClient or DEXPair, the *owner*) deploys a DEXConnector contract;
- The contract immediately sends a `deployEmptyWallet` message to the DEXConnector contract;
- The connector contract:
 - Sends a `deployEmptyWallet` message to the RootTokenContract contract;
 - Sends a `sendExpectedWalletAddress` message to the RootTokenContract contract;
- The root token contract:
 - Deploys the empty TONTokenWallet contract;
 - Sends a `expectedWalletAddressCallback` message to the DEXConnector contract
- The connector sends a `connectCallback` message to the initial *owner* contract (DEXClient or DEXPair);
- The *owner* contract sends:
 - A `setTransferCallback` to the DEXConnector contract;
 - A `setBouncedCallback` to the DEXConnector contract;
- The DEXConnector contract forwards the `setReceiveCallback` and `setBouncedCallback` messages to the TONTokenWallet contract;

2.3.4 connectPair sequence

- The user sends a `connectPair` message to his DEXClient contract;
- The client contract sends a `connect` message to the DEXPair contract;
- The DEXPair contract sends back its information through a `setPair` message;

2.3.5 processSwapA sequence

- The user sends a `processSwapA` message to his DEXClient contract;
- The client contract sends a `transfer` message to the DEXConnector contract, with a payload `(1,rootB,walletB)` ;
- The DEXConnector contract forwards the `transfer` message to the associated TONTokenWallet contract;

- The `TONTokenWallet` contract sends a `internalTransfer` message to the destination `TONTokenWallet` contract of the pair;
- The destination `TONTokenWallet` contract sends a `tokensReceivedCallback` message to his associated `DEXPair` contract;
- The `DEXPair` checks whether the swap is possible or not:
 - If the swap is not possible, it sends a `transfer` message to the `DEXConnector` to refund the user, with a `(8,0,0)` payload;
 - If the swap is possible, it sends a `transfer` message to the other `DEXConnector` to pay the user, with a `(0,0,0)` payload;

2.3.6 processLiquidity sequence

- The user sends a `processLiquidity` message to his `DEXClient` contract;
- The client contract sends a `transfer` message to the two associated `DEXConnector` contracts, with a payload `(2,dexclient,walletLiquidity)` ;
- The `DEXConnector` contracts forward the `transfer` messages to the associated `TONTokenWallet` contracts;
- The `TONTokenWallet` contracts send `internalTransfer` messages to the destination `TONTokenWallet` contracts of the pair;
- The destination `TONTokenWallet` contracts send `tokensReceivedCallback` messages to their associated `DEXPair` contract;
- TODO

2.3.7 returnLiquidity sequence

- The user sends a `returnLiquidity` message to his `DEXClient` contract;
- The client contract sends a `burn` message to the user's `DEXConnector` contract of the liquidity token, with a payload `(3,walletA,walletB)` ;
- The `DEXConnector` contracts forwards a `burnByOwner` message to his associated `TONTokenWallet` contract;
- The `TONTokenWallet` contract burns the tokens and sends a `tokensBurned` message to the associated `RootTokenContract` contract;
- The `RootTokenContract` contract sends a `burnCallback` message to the associated `DEXPair` contract (provided in the initial `burn` message);
- The `DEXPair` contract sends `transfer` messages to the two `DEXConnector` contracts to transfer back the tokens corresponding to the liquidity, with a payload of `(6,0,0)`;

- The `DEXConnector` contracts send `transfer` messages to the corresponding `TONTokenWallet` contracts;
- The `TONTokenWallet` contracts send `internalTransfer` to the dexclient `TONTokenWallet` contracts;
- The `TONTokenWallet` contracts of the dexclient send `tokensReceivedCallback` messages to the dexclient, that records the callbacks in the `callbacks` mapping;

Chapter 3

General Remarks

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

3.1 Readability

3.1.1 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

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

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 Numbers

A good coding convention is to define constants instead of using direct numbers for errors and other meaningful numbers.

This issue was found everywhere in the code of DEX contracts (for errors in `require()` and payload opcodes), but not for token contracts.

3.1.5 Better Units for Big Numbers

A good coding convention is to 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 5000000000 are difficult to read, whereas the equivalent `0.5 ton` is much easier.

3.1.6 Use Method Calls 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 Gas Consumption

3.2.1 Accept Methods without Checks

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`

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.

3.2.3 Not Enough Gas for Action

If there is not enough gas (message value or balance), the compute phase may execute, but the action phase may fail. In such a case, modifications done during the compute phase are committed to the blockchain, but no message emitted.

This issue was found in the code, for example in `DEXClient.connectRoot`.

3.3 Architecture

3.3.1 No need for passing `souint` Arguments around

It looks like there is little need for passing around the `souint` arguments as the same `soUINT` static value could be used for the `DEXroot` and all other contracts derived from it. The only required modification would be to make the `drivenRoot` field of `DEXConnector` a static variable. Such a change would probably simplify the interface of many functions.

Chapter 4

Contract DEXClient

Contents

4.1	Contract Inheritance	20
4.2	Type Definitions	20
4.2.1	Struct Connector	20
4.2.2	Struct Callback	20
4.2.3	Struct Pair	21
4.3	Constant Definitions	21
4.4	Static Variable Definitions	21
4.5	Variable Definitions	22
4.6	Modifier Definitions	22
4.6.1	Modifier alwaysAccept	22
4.6.2	Modifier checkOwnerAndAccept	22
4.7	Constructor Definitions	23
4.7.1	Constructor	23
4.8	Public Method Definitions	23
4.8.1	Receive function	23
4.8.2	Function connectCallback	23
4.8.3	Function connectPair	24
4.8.4	Function connectRoot	24
4.8.5	Function createNewPair	25
4.8.6	Function getAllDataPreparation	26
4.8.7	Function getBalance	26
4.8.8	Function getCallback	26
4.8.9	Function getConnectorAddress	27
4.8.10	Function getPairData	27
4.8.11	Function processLiquidity	27
4.8.12	Function processSwapA	28
4.8.13	Function processSwapB	29

4.8.14	Function returnLiquidity	29
4.8.15	Function sendTokens	30
4.8.16	Function setPair	31
4.8.17	Function tokensReceivedCallback	31
4.9	Internal Method Definitions	32
4.9.1	Function computeConnectorAddress	32
4.9.2	Function getFirstCallback	33
4.9.3	Function getQuotient	33
4.9.4	Function getRemainder	33
4.9.5	Function isReady	33
4.9.6	Function isReadyToProvide	34
4.9.7	Function thisBalance	34

In file DEXClient.sol

4.1 Contract Inheritance

ITokensReceivedCallback	
IDEXClient	
IDEXConnect	

4.2 Type Definitions

4.2.1 Struct Connector

- OK

```

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

4.2.2 Struct Callback

- Minor Issue (readability): `payload_arg0`, `payload_arg1`, `payload_arg2` should be renamed to more explicit names

```

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

- OK

```

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

- Minor Issue (readability): see Better Units for Big Numbers (3.1.5). Use ton unit to make big numbers easier to understand.

```

23 uint128 constant GRAMS_CONNECT_PAIR = 5000000000;
24 uint128 constant GRAMS_SET_CALLBACK_ADDR = 1000000000;
25 uint128 constant GRAMS_SWAP = 5000000000;
26 uint128 constant GRAMS_PROCESS_LIQUIDITY = 5000000000;
27 uint128 constant GRAMS_RETURN_LIQUIDITY = 5000000000;

```

4.4 Static Variable Definitions

- Minor Issue (readability): see Typography of Static Variables (3.1.1). Static variables should start with `s_` for example.
- Minor Issue: Deployment messages are limited to 16 kB, and contain the code of the contract, the static variables and the constructor arguments. As `codeDEXConnector` is a static variable, the deployment message will contain the code of DEXClient and DEXConnector at the same time. It could become an issue in the future if their codes increase in size. If it is important to use DEXConnector code static to distinguish clients, it might be worth replacing it by a hash and use another message to initialize the variable instead.

```

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

```

4.5 Variable Definitions

- Minor Issue (readability): see Typography of Global Variables (3.1.2). Global variables should start with `m_` or `g_` for example.

```

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

- **Major issue: Accept-All Modifier in DEXClient**
See Accept Methods without Checks (3.2.1). This modifier should be removed.

```

73  modifier alwaysAccept {
74      tvml.accept();
75      -;
76  }

```

4.6.2 Modifier checkOwnerAndAccept

- Minor Issue (readability): see Naming of Constants (3.1.4). A number is directly used in `require()`. It should be replaced by a constant.

```

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

```

4.7 Constructor Definitions

4.7.1 Constructor

- Minor Issue (readability): see Naming of Constants (3.1.4). A number is directly used in `require()`. It should be replaced by a constant.
- Minor Issue (Semantics): `counterCallback` should probably be initialized to 1 instead of 0, and keep 0 as the specific value of `getFirstCallback` when no callback is available.

```

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

```

4.8 Public Method Definitions

4.8.1 Receive function

- OK

```

413  receive() external {
414  }

```

4.8.2 Function connectCallback

- Minor Issue (readability): use Method Calls instead of `tvml.encodeBody` (3.1.6).

Major issue: Accept-All Method in DEXClient.connectCallback

- See No Accept-All Methods (3.2.1) The balance of the contract could be drained, by sending many `connectCallback` messages. Replace the `if` by a `require()` and perform the `tvml.accept` only afterwards.

```

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 = tvml.encodeBody(IDEXConnector(connector).
              setTransferCallback);
              connector.transfer({value: GRAMS_SET_CALLBACK_ADDR, bounce:
                  true, flag: 0, body:bodySTC});

```

```

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

```

4.8.3 Function connectPair

- Minor Issue (readability): use Method Calls instead of `tvm.encodeBody` (3.1.6).

```

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         }

```

4.8.4 Function connectRoot

- Minor Issue (gas loss): see Not Enough Gas for Action (3.2.3). The method should check for a minimal balance in a `require()` before `tvm.accept`
- Minor Issue (readability): use Method Calls instead of `tvm.encodeBody` (3.1.6).

```

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;

```



```

171     cr.souint = souint;
172     cr.status = false;
173     connectors[connector] = cr;
174     TvmCell body = tvm.encodeBody(IDEXConnector(connector).
        deployEmptyWallet, root);
175     connector.transfer({value:gramsToRoot, bounce:true, body:body
        });
176     statusConnected = true;
177 }
178 }

```

4.8.5 Function createNewPair

- Minor Issue: a `require` is executed after `tvm.accept`, which may cause replication of failure and heavy gas cost. In general, we recommend to remove `tvm.accept` from modifiers, so that it can be explicitly performed after all `requires`.
- Minor Issue (readability): use Method Calls instead of `tvm.encodeBody` (3.1.6).
- Minor Issue (readability): see Naming of Constants (3.1.4). A number is directly used in `require()`. It should be replaced by a constant.

```

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

Major issue: Accept-All Method in DEXClient.getAllDataPreparation

- See No Accept-All Methods (3.2.1) The balance of the contract could be drained, by sending many `getAllDataPreparation` messages, especially as the return values may be expensive to serialize. Accept only for owner or use it as a get-method (executed locally, without gas).

```

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

```

4.8.7 Function getBalance

- OK

```

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

```

4.8.8 Function getCallback

- Minor Issue (Gas loss): there is probably no need to spend gas with `tvm.accept` since the method can be executed locally (get-method).
- Minor Issue (Semantics): the method should probably fail with `require` if the callback id does not exist.

```

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

- OK

```

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

```

4.8.10 Function getPairData

Major issue: Accept-All Method in DEXClient.getPairData

See No Accept-All Methods (3.2.1) The balance of the contract could be drained, by sending many `getPairData` messages. Accept only for owner or use it as a get-method (executed locally, without gas).

- Minor Issue: the method should probably fail if the associated pair does not exist in `pairs`

```

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

```

4.8.11 Function processLiquidity

- Minor Issue (readability): use Method Calls instead of `tvm.encodeBody` (3.1.6).

- Minor Issue (readability): see Naming of Constants (3.1.4). A number should be named through a constant.. The payload opcode should be a constant.
- Minor Issue (gas loss): see Not Enough Gas for Action (3.2.3). The method should check the balance with `require` before performing `tvm.accept`
- Minor Issue: Repeated Code. The code could be simplified by using an internal function to perform the same computation for `rootA` and `rootB`.

```

251 function processLiquidity(address pairAddr, uint128 qtyA, uint128
    qtyB) public view checkOwnerAndAccept returns (bool
    processLiquidityStatus) {
252     processLiquidityStatus = false;
253     if (isReadyToProvide(pairAddr)) {
254         Pair cp = pairs[pairAddr];
255         address connectorA = rootConnector[cp.rootA];
256         address connectorB = rootConnector[cp.rootB];
257         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 = tvml.encodeBody(IDEXConnector(connectorA).
            transfer, cp.walletA, qtyA, payloadA);
264         TvmCell bodyB = tvml.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

- Minor Issue (readability): use Method Calls instead of `tvm.encodeBody` (3.1.6).
- Minor Issue (readability): see Naming of Constants (3.1.4). A number should be named through a constant.. The payload opcode should be a constant.
- Minor Issue (gas loss): see Not Enough Gas for Action (3.2.3). The method should check the balance with `require` before performing `tvm.accept`
- Minor Issue: Repeated Code. Methods `processSwapA` and `processSwapB` could be simplified by using an internal function for shared code.

```

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 = tvml.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

- Minor Issue (readability): see Naming of Constants (3.1.4). A number should be named through a constant.. The payload opcode should be a constant.
- Minor Issue (readability): use Method Calls instead of `tvm.encodeBody` (3.1.6).
- Minor Issue (gas loss): see Not Enough Gas for Action (3.2.3). The method should check the balance with `require` before performing `tvm.accept`

```

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 = tvml.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

- Minor Issue (readability): use Method Calls instead of `tvm.encodeBody` (3.1.6).

- Minor Issue (readability): see Naming of Constants (3.1.4). A number should be named through a constant.. The payload opcode should be a constant.
- Minor Issue (gas loss): see Not Enough Gas for Action (3.2.3). The method should check the balance with `require` before performing `tvm.accept`

```

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

- Minor Issue: the method should use `require` instead of `if` (before doing the `tvm.accept`).
- Minor Issue (readability): use Method Calls instead of `tvm.encodeBody` (3.1.6).
- Minor Issue (readability): see Naming of Constants (3.1.4). A number should be named through a constant.. The payload opcode should be a constant.
- Minor Issue (gas loss): see Not Enough Gas for Action (3.2.3). The method should check the balance with `require` before performing `tvm.accept`

```

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

Major issue: Accept-All Method in DEXClient.setPair

- See No Accept-All Methods (3.2.1) The balance of the contract could be drained, by sending many `setPair` messages. You should replace the `if` by a `require()` followed by `tvm.accept()`.
- Minor Issue (gas loss): see Not Enough Gas for Action (3.2.3).

```

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

```

4.8.17 Function tokensReceivedCallback

Major issue: Accept-All Method in DEXClient.tokensReceivedCallback

- See No Accept-All Methods (3.2.1)
 - The balance of the contract could be drained, by sending many unexpected `tokensReceivedCallback` messages by an attacker. The attack could be improved by sending wrong payloads, causing `slice.decode` to fail after `tvm.accept`, causing replication of failure on all shards.
 - An attacker could send many such messages also to include incorrect receipts in the `callbacks` mapping, and remove correct ones by sending more than 10 such messages. Yet, this part of the attack is probably not critical, as receipts are only used by humans.
 - Fix:Accept only when sender is one of the wallets of this dexclient or use the `msg.value` gas.

```

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()
315             ];}
316     }

```

4.9 Internal Method Definitions

4.9.1 Function computeConnectorAddress

- Minor Issue (readability): see Typography of Internal Functions (3.1.3). The function name should start with `_`.

```

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`

- Minor Issue: if no callback is present, the function returns 0. This value could be reserved for this usage by setting `counterCallback` to 1 in the constructor.
- Minor Issue (readability): see Typography of Internal Functions (3.1.3). The function name should start with `_`.

```

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;}

```

4.9.3 Function `getQuotient`

- Minor Issue (Useless code): This internal function is unused.
- Minor Issue (readability): see Typography of Internal Functions (3.1.3). The function name should start with `_`.

```

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

```

4.9.4 Function `getRemainder`

- Minor Issue (Useless code): This internal function is unused.
- Minor Issue (readability): see Typography of Internal Functions (3.1.3). The function name should start with `_`.

```

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

```

4.9.5 Function `isReady`

- Minor Issue (readability): see Typography of Internal Functions (3.1.3). The function name should start with `_`.

```

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

- Minor Issue (readability): see Typography of Internal Functions (3.1.3). The function name should start with ..
- Minor Issue (Repeated Code): This function could easily be derived from isReady with the additional check of rootWallet.exists(cp.rootAB)

```

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

- Minor Issue (readability): see Typography of Internal Functions (3.1.3). The function name should start with ..

```

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

```

Chapter 5

Contract DEXConnector

Contents

5.1	Contract Inheritance	35
5.2	Constant Definitions	36
5.3	Static Variable Definitions	36
5.4	Variable Definitions	36
5.5	Modifier Definitions	37
5.5.1	Modifier alwaysAccept	37
5.5.2	Modifier checkOwnerAndAccept	37
5.6	Constructor Definitions	37
5.6.1	Constructor	37
5.7	Public Method Definitions	37
5.7.1	Receive function	37
5.7.2	Function burn	38
5.7.3	Function deployEmptyWallet	38
5.7.4	Function expectedWalletAddressCallback	39
5.7.5	Function getBalance	39
5.7.6	Function setBouncedCallback	39
5.7.7	Function setTransferCallback	40
5.7.8	Function transfer	40
5.8	Internal Method Definitions	41
5.8.1	Function getQuotient	41
5.8.2	Function getRemainder	41

In file `DEXConnector.sol`

5.1 Contract Inheritance

IExpectedWalletAddressCallback	
IDEXConnector	

5.2 Constant Definitions

- Minor Issue (readability): see Better Units for Big Numbers (3.1.5). Use `ton` unit to make big numbers easier to understand.

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

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

5.3 Static Variable Definitions

- Minor Issue (readability): see Typography of Static Variables (3.1.1). Static variables should start with `s_` for example.
- 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`. Using `drivenRoot` as a static would also make the need to pass `souint` around useless, since the same `souint` could be used everywhere, from the `DEXRoot` to all the clients, pairs and connectors created from it.
- Minor issue: the name `dexclient` is misleading. In `DEXPair.connectRoot`, connectors are created for `DEXPair` passing `this` as `dexclient`, i.e. an address of type `DEXPair`. Maybe use `owner_address` ?

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

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

5.4 Variable Definitions

- Minor Issue (readability): see Typography of Global Variables (3.1.2). Global variables should start with `m_` or `g_` for example.

```
22  address public drivenRoot;
```

```
23  address public driven;
```

```
24  bool public statusConnected;
```

5.5 Modifier Definitions

5.5.1 Modifier alwaysAccept

- **Major issue: Accept-All Modifier in DEXConnector**
See Accept Methods without Checks (3.2.1). This modifier should be removed.

```

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

5.5.2 Modifier checkOwnerAndAccept

- Minor Issue (readability): see Naming of Constants (3.1.4). A number should be named through a constant.

```

32  modifier checkOwnerAndAccept {
33      // Check that message from contract owner.
34      require(msg.sender == dexclient, 101);
35      tvm.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 (readability): see Naming of Constants (3.1.4). A number is directly used in `require()`. It should be replaced by a constant.
- Minor Issue (readability): use 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 (readability): see Naming of Constants (3.1.4). A number is directly used in `require()`. It should be replaced by a constant.
- Minor Issue (readability): use 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 (readability): see Naming of Constants (3.1.4). A number is directly used in `require()`. It should be replaced by a constant.
- Minor Issue (readability): use 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 (readability): see Naming of Constants (3.1.4). A number is directly used in `require()`. It should be replaced by a constant.
- Minor Issue (readability): use 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(ITONTOKENWallet(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 (readability): see Naming of Constants (3.1.4). A number is directly used in `require()`. It should be replaced by a constant.
- Minor Issue (readability): use 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(ITONTOKENWallet(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 (readability): see Naming of Constants (3.1.4). A number is directly used in `require()`. It should be replaced by a constant.
- Minor Issue (readability): use 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(ITONTOKENWallet(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

- Minor Issue: This function is unused.
- Minor Issue (readability): see Typography of Internal Functions (3.1.3). The function name should start with `_`.

```

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

- Minor Issue: This function is unused.
- Minor Issue (readability): see Typography of Internal Functions (3.1.3). The function name should start with `_`.

```

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

Contents

6.1	Contract Inheritance	43
6.2	Type Definitions	43
6.2.1	Struct Connector	43
6.2.2	Struct Callback	43
6.3	Constant Definitions	44
6.4	Static Variable Definitions	44
6.5	Variable Definitions	45
6.6	Modifier Definitions	45
6.6.1	Modifier alwaysAccept	45
6.6.2	Modifier checkOwnerAndAccept	46
6.6.3	Modifier checkPubKeyAndAccept	46
6.7	Constructor Definitions	46
6.7.1	Constructor	46
6.8	Public Method Definitions	46
6.8.1	Receive function	46
6.8.2	Function burnCallback	47
6.8.3	Function connect	48
6.8.4	Function connectCallback	48
6.8.5	Function getBalance	49
6.8.6	Function getCallback	49
6.8.7	Function getConnectorAddress	50
6.8.8	Function tokensReceivedCallback	50
6.9	Internal Method Definitions	54
6.9.1	Function acceptForProvide	54
6.9.2	Function cleanProcessing	55
6.9.3	Function computeConnectorAddress	55
6.9.4	Function connectRoot	55

6.9.5	Function getAmountOut	56
6.9.6	Function getFirstCallback	57
6.9.7	Function getQuotient	57
6.9.8	Function getRemainder	57
6.9.9	Function liquidityA	58
6.9.10	Function liquidityB	58
6.9.11	Function qtyAforB	58
6.9.12	Function qtyBforA	59
6.9.13	Function thisBalance	59

In file `DEXPair.sol`

6.1 Contract Inheritance

IDEXPair	
IDEXConnect	
ITokensReceivedCallback	
IBurnTokensCallback	

6.2 Type Definitions

6.2.1 Struct Connector

- OK

```

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

6.2.2 Struct Callback

- Minor Issue (readability): `payload_arg0`, `payload_arg1`, `payload_arg2` should be renamed to more explicit names

```

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

- Minor Issue (readability): see Better Units for Big Numbers (3.1.5). Use `ton` unit to make big numbers easier to understand.

```

65     uint128 constant GRAMS_SET_CALLBACK_ADDR = 5000000000;

```

```

66     uint128 constant GRAMS_SEND_UNUSED_RETURN = 1000000000;

```

```

67     uint128 constant GRAMS_MINT = 500000000;

```

```

68     uint128 constant GRAMS_RETURN = 2000000000;

```

6.4 Static Variable Definitions

- Minor Issue (readability): see Typography of Static Variables (3.1.1). Static variables should start with `s_` for example.
- Minor Issue: Deployment messages are limited to 16 kB, and contain the code of the contract, the static variables and the constructor arguments. As `codeDEXConnector` is a static variable, the deployment message will contain the code of DEXClient and DEXConnector at the same time. It could become an issue in the future if their codes increase in size. If it is important to use DEXConnector code static to distinguish clients, it might be worth replacing it by a hash and use another message to initialize the variable instead.

```

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

- Minor Issue (readability): see Typography of Global Variables (3.1.2). Global variables should start with `m_` or `g_` for example.
- Minor Issue (readability): variables could have better names to understand their use, especially for mappings where it is important to discriminate other the different possible addresses. For example, `walletReserve` could be `walletReserve_of_root`, idem for `syncstatus` and `rootConnector`; `connectors` could be `info_of_connector`. `processingStatus`, `processingData` and `processingDest` could be renamed as `processingStatus_of_root_x_client` and so on.

```

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

- **Major issue: Accept-All Modifier in DEXPair**
See Accept Methods without Checks (3.2.1). This modifier should be removed.

```

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

```

6.6.2 Modifier checkOwnerAndAccept

- Minor Issue (readability): see Naming of Constants (3.1.4). A number is directly used in `require()`. It should be replaced by a constant.

```

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

```

6.6.3 Modifier checkPubKeyAndAccept

- Minor Issue (readability): see Naming of Constants (3.1.4). A number is directly used in `require()`. It should be replaced by a constant.

```

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

```

6.7 Constructor Definitions

6.7.1 Constructor

- Minor Issue (Semantics): `counterCallback` should probably be initialized to 1 instead of 0, and keep 0 as the specific value of `getFirstCallback` when no callback is available.

```

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

- OK

```

609  receive() external {
610  }

```

6.8.2 Function burnCallback

Critical issue: Unlimited Growth of callbacks

- while `burnCallback` adds a receipt in the same way as `tokensReceivedCallback`, it does not check for the presence of more than 10 callbacks, leading to an unlimited growth of the `callbacks` mapping. This unlimited growth of the contract storage will progressively increase the gas price of calling its methods, until it becomes unaffordable for anybody. The issue can thus be considered even critical on the long term.

Major issue: Accept-All Method in `DEXPair.burnCallback`

- See No Accept-All Methods (3.2.1) The balance of the contract could be drained, by sending many unexpected `burnCallback` messages by an attacker. To prevent the attack, use `require` instead of `if` and perform `tvm.accept` only afterwards.
- Minor Issue (Readability): rename `arg0`, `arg1` and `arg2` with better name, for example `opcode`, `walletA` and `walletB` respectively.
- Minor Issue (readability): use Method Calls instead of `tvm.encodeBody` (3.1.6).

```

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         tvn.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);

```

```

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

```

6.8.3 Function connect

- Minor Issue (readability): use Method Calls instead of `tvm.encodeBody` (3.1.6).

```

149 function connect() public override {
150     address dexclient = msg.sender;
151     tvm.rawReserve(address(this).balance - msg.value, 2);
152     TvmCell body = tvm.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

Major issue: Accept-All Method in DEXPair.connectCallback

- See No Accept-All Methods (3.2.1) The balance of the contract could be drained, by sending many `connectCallback` messages. Replace the `if` by a `require()` and perform `tvm.accept` only afterwards.
- Minor Issue (simplification): The calls to `setTransferCallback` and `setBouncedCallback` are always performed together. They could be replaced by just one call by modifying the `DEXConnector` contract.

- Minor Issue (readability): use Method Calls instead of `tvm.encodeBody` (3.1.6).

```

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

- OK

```

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

```

6.8.6 Function getCallback

- Minor Issue (Gas loss): there is probably no need to spend gas with `tvm.accept` since the method can be executed locally (get-method).
- Minor Issue (Semantics): the method should probably fail with `require` if the callback id does not exist.

```

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

- OK

```

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

```

6.8.8 Function tokensReceivedCallback

Critical issue: Swap Refund does not work in DEXPair.tokensReceivedCallback

- On line 294, in case of refund, a transfer is performed to `token_wallet`, but this variable is the first argument of the `tokensReceivedCallback`, i.e. the wallet of the pair that received the tokens. Thus, this transfer is actually a transfer from the wallet to itself, and the user will never received his refund. Fix: `token_wallet` should be replaced by `sender_wallet`,

Major issue: Accept-All Method in DEXPair.tokensReceivedCallback

See No Accept-All Methods (3.2.1)

- The balance of the contract could be drained, by sending many unwanted `tokensReceivedCallback` messages. Replace the `if` by a `require()` and perform the `tvm.accept` only afterwards.
- – The balance could also be drained by sending small amounts of tokens to “swap” directly to the `TONTokenWallet` of the pair (without using `DEXCclient` for that), with an invalid payload. Indeed, `slice.decode` would fail after `tvm.accept`, and the error would be executed on all shards, causing an heavy loss of gas.

- Minor issue (Repeated Code): the code between lines 373 and 482 is a repetition of the code between lines 261 and 370. The only difference is the `delete callbacks[getFirstCallback()]` line in the case `counterCallback > 10`. The duplication could be avoided by just performing the test before this line.
- Minor issue (Repeated code): the code between lines 261 and 277 is expected to perform the same task as the code in the method `burnCallback` between lines 532 and 547. An internal method could be used to avoid this code duplication, and would fix the issue found in `burnCallback` in these lines.
- Minor issue (Readability): the code of this function is too long, the code would benefit from splitting the code in several shorter internal functions.
- Minor Issue (readability): see Naming of Constants (3.1.4). A number should be named through a constant. Here, opcodes in payload should be named by constants.
- Minor Issue (readability): use Method Calls instead of `tvm.encodeBody` (3.1.6).

```

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 ==
        walletReserve[rootB]) {
260         if (counterCallback > 10) {
261             Callback cc = callbacks[counterCallback];
262             cc.token_wallet = token_wallet;
263             cc.token_root = token_root;
264             cc.amount = amount;
265             cc.sender_public_key = sender_public_key;
266             cc.sender_address = sender_address;
267             cc.sender_wallet = sender_wallet;
268             cc.original_gas_to = original_gas_to;
269             cc.updated_balance = updated_balance;
270             TvmSlice slice = payload.toSlice();
271             (uint8 arg0, address arg1, address arg2) = slice.decode(
                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         tvm.rawReserve(address(this).balance - msg.value, 2);
280         uint128 amountOut = getAmountOut(amount, token_root, arg1
281             );
282         if (!(amountOut > balanceReserve[arg1])){
283             balanceReserve[token_root] += amount;
284             balanceReserve[arg1] -= amountOut;
285             syncStatus[token_root] = balanceReserve[token_root] ==
                updated_balance ? true : false;
286             TvmBuilder builder;
287             builder.store(uint8(0), address(0), address(0));
288             TvmCell new_payload = builder.toCell();
289             TvmCell body = tvm.encodeBody(IDEXConnector(
                rootConnector[arg1]).transfer, arg2, amountOut,
                new_payload);
290             rootConnector[arg1].transfer({value: 0, bounce:true,
                flag: 128, body:body});
291         } else {
292             TvmBuilder builder;
293             builder.store(uint8(8), address(0), address(0));
294             TvmCell new_payload = builder.toCell();
295             TvmCell body = tvm.encodeBody(IDEXConnector(
                rootConnector[token_root]).transfer, token_wallet,
                amount, new_payload);
296             rootConnector[token_root].transfer({value: 0, bounce:
                true, flag: 128, body:body});
297         }
298     }
299     if (arg0 == 2) {
300         tvm.rawReserve(address(this).balance - msg.value, 2);
301         processingStatus[token_root][arg1] = true;
302         processingData[token_root][arg1] += amount;
303         processingDest[token_root][arg1] = sender_wallet;
304         if (processingStatus[rootA][arg1] == true &&
305             processingStatus[rootB][arg1] == true) {
306             uint128 amountA = processingData[rootA][arg1];
307             uint128 amountB = processingData[rootB][arg1];
308             if (totalSupply == 0 && balanceReserve[rootA] == 0 &&
309                 balanceReserve[rootB] == 0) {
310                 uint128 liquidity = math.min(amountA, amountB);
311                 balanceReserve[rootA] += amountA;
312                 balanceReserve[rootB] += amountB;
313                 totalSupply += liquidity;
314                 TvmCell body = tvm.encodeBody(IRootTokenContract(
315                     rootAB).mint, liquidity, arg2);
316                 rootAB.transfer({value: GRAMS_MINT, bounce:true, body
317                     :body});
318                 cleanProcessing(arg1);
319                 arg1.transfer({ value: 0, flag: 128});
320             } else {
321                 (uint128 provideA, uint128 provideB) =
322                     acceptForProvide(amountA, amountB);
323                 if (provideA > 0 && provideB > 0) {
324                     uint128 liquidity = math.min(liquidityA(provideA),
325                         liquidityB(provideB));
326                     uint128 unusedReturnA = amountA - provideA;
327                     uint128 unusedReturnB = amountB - provideB;

```

```

321     balanceReserve[rootA] += provideA;
322     balanceReserve[rootB] += provideB;
323     totalSupply += liquidity;
324     TvmCell body = tvm.encodeBody(IRootTokenContract(
325         rootAB).mint, liquidity, arg2);
326     rootAB.transfer({value: GRAMS_MINT, bounce:true,
327         body:body});
328     if (unusedReturnA > 0 && unusedReturnB > 0) {
329         TvmBuilder builder;
330         builder.store(uint8(7), address(0), address(0));
331         TvmCell new_payload = builder.toCell();
332         TvmCell bodyA = tvm.encodeBody(IDEXConnector(
333             rootConnector[rootA]).transfer,
334             processingDest[rootA][arg1], unusedReturnA,
335             new_payload);
336         TvmCell bodyB = tvm.encodeBody(IDEXConnector(
337             rootConnector[rootB]).transfer,
338             processingDest[rootB][arg1], unusedReturnB,
339             new_payload);
340         rootConnector[rootA].transfer({value:
341             GRAMS_SEND_UNUSED_RETURN, bounce:true, body:
342             bodyA});
343         rootConnector[rootB].transfer({value:
344             GRAMS_SEND_UNUSED_RETURN, bounce:true, body:
345             bodyB});
346         cleanProcessing(arg1);
347         arg1.transfer({ value: 0, flag: 128});
348     } else if (unusedReturnA > 0) {
349         TvmBuilder builder;
350         builder.store(uint8(7), address(0), address(0));
351         TvmCell new_payload = builder.toCell();
352         TvmCell bodyA = tvm.encodeBody(IDEXConnector(
353             rootConnector[rootA]).transfer,
354             processingDest[rootA][arg1], unusedReturnA,
355             new_payload);
356         rootConnector[rootA].transfer({value:
357             GRAMS_SEND_UNUSED_RETURN, bounce:true, body:
358             bodyA});
359         cleanProcessing(arg1);
360         arg1.transfer({ value: 0, flag: 128});
361     } else if (unusedReturnB > 0) {
362         TvmBuilder builder;
363         builder.store(uint8(7), address(0), address(0));
364         TvmCell new_payload = builder.toCell();
365         TvmCell bodyB = tvm.encodeBody(IDEXConnector(
366             rootConnector[rootB]).transfer,
367             processingDest[rootB][arg1], unusedReturnB,
368             new_payload);
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     } else {
375         cleanProcessing(arg1);
376         arg1.transfer({ value: 0, flag: 128});
377     }

```

```

356     } else {
357         TvmBuilder builder;
358         builder.store(uint8(9), address(0), address(0));
359         TvmCell new_payload = builder.toCell();
360         TvmCell bodyA = tvm.encodeBody(IDEXConnector(
361             rootConnector[rootA]).transfer, processingDest[
362                 rootA][arg1], amountA, new_payload);
363         TvmCell bodyB = tvm.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 }

```

6.9 Internal Method Definitions

6.9.1 Function acceptForProvide

- Minor issue (Readability): choose more explicit names for `arg0` and `arg1`.
- Minor issue (Readability): `crquotient` and `crremainder` could be computed in just one command using `math.muldivmod` instead of calling `getQuotient` and `getRemainder`.
- Minor Issue (readability): see Naming of Constants (3.1.4). A number is directly used in `require()`. It should be replaced by a constant.
- Minor Issue (readability): see Typography of Internal Functions (3.1.3). The function name should start with `_`.

```

214 function acceptForProvide(uint128 arg0, uint128 arg1) private
215     inline view returns (uint128, uint128) {
216     require(balanceReserve[rootA] > 0 && balanceReserve[rootB] > 0,
217         106);
218     uint128 qtyB = qtyBforA(arg0);
219     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

- Minor Issue (readability): see Typography of Internal Functions (3.1.3).
The function name should start with `_`.

```

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

- Minor Issue (readability): see Typography of Internal Functions (3.1.3).
The function name should start with `_`.

```

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

```

6.9.4 Function connectRoot

- Minor Issue (Readability): the `dexclient` static variable of `DEXConnector` is initialized with `address(this)`, clearly not a `DEXClient` but a `DEXPair`.

The field should be renamed to match the possibility to use another contract than `DEXClient`, for example `owner_address`.

- Minor Issue (readability): use Method Calls instead of `tvm.encodeBody` (3.1.6).
- Minor Issue (readability): see Typography of Internal Functions (3.1.3). The function name should start with `_`.

```

113 function connectRoot(address root, uint256 souint, uint128
114   gramsToConnector, uint128 gramsToRoot) private inline {
115   TvmCell stateInit = tvm.buildStateInit({
116     contr: DEXConnector,
117     varInit: { soUINT: souint, dexclient: address(this) },
118     code: codeDEXConnector,
119     pubkey: tvm.pubkey()
120   });
121   TvmCell init = tvm.encodeBody(DEXConnector);
122   address connector = tvm.deploy(stateInit, init,
123     gramsToConnector, address(this).wid);
124   Connector cr = connectors[connector];
125   cr.root_address = root;
126   cr.souint = souint;
127   cr.status = false;
128   connectors[connector] = cr;
129   TvmCell body = tvm.encodeBody(IDEXConnector(connector).
130     deployEmptyWallet, root);
131   connector.transfer({value:gramsToRoot, bounce:true, body:body})
132   ;
133 }

```

6.9.5 Function getAmountOut

- Minor Issue (Readability): `math.muldiv(x,y,1)` is equivalent to `x * y` in TON Solidity, and less readable.
- Minor Issue (Precision): Given that `numerator` is a factor, you could increase precision by using `math.muldivmod` to perform together the multiplication of `numerator` and the division in the last step, as it keeps the intermediate factor in a 514 bit buffer before performing the division.
- Minor Issue (readability): see Naming of Constants (3.1.4). A number is directly used in `require()`. It should be replaced by a constant.
- Minor Issue (readability): see Typography of Internal Functions (3.1.3). The function name should start with `_`.

```

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

```



```

160     uint128 denominator = amountInWithFee + math.muldiv(
161         balanceReserve[rootIn], 1000, 1);
162     return math.muldiv(1, numerator, denominator);
    }

```

6.9.6 Function getFirstCallback

- Minor Issue: if no callback is present, the function returns 0. This value could be reserved for this usage by setting `counterCallback` to 1 in the constructor.
- Minor Issue (readability): see Typography of Internal Functions (3.1.3). The function name should start with `_`.

```

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

```

6.9.7 Function getQuotient

- Minor Issue (Readability): this function does not provide any improvement over a simple division, as it does not benefit from the increased temporary buffer for the multiplication.
- Minor Issue (readability): see Typography of Internal Functions (3.1.3). The function name should start with `_`.

```

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

```

6.9.8 Function getRemainder

- Minor Issue (readability): see Typography of Internal Functions (3.1.3). The function name should start with `_`.

```

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

```

6.9.9 Function liquidityA

- Minor Issue (Readability): choose a better name than `arg0`
- Minor Issue (readability): see Naming of Constants (3.1.4). A number is directly used in `require()`. It should be replaced by a constant.
- Minor Issue (readability): see Typography of Internal Functions (3.1.3). The function name should start with `_`.

```

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

```

6.9.10 Function liquidityB

- Minor Issue (Readability): choose a better name than `arg0`
- Minor Issue (readability): see Naming of Constants (3.1.4). A number is directly used in `require()`. It should be replaced by a constant.
- Minor Issue (readability): see Typography of Internal Functions (3.1.3). The function name should start with `_`.

```

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

```

6.9.11 Function qtyAforB

- Minor Issue (Readability): choose a better name than `arg1`
- Minor Issue (readability): see Naming of Constants (3.1.4). A number is directly used in `require()`. It should be replaced by a constant.
- Minor Issue (readability): see Typography of Internal Functions (3.1.3). The function name should start with `_`.

```

184 function qtyAforB(uint128 arg1) private inline view returns (
185     uint128) {
186     require(arg1 > 0, 107);
187     require(balanceReserve[rootA] > 0 && balanceReserve[rootB] > 0,
188         106);

```

```

187     return math.muldiv(arg1, balanceReserve[rootA], balanceReserve[
188         rootB]);
    }

```

6.9.12 Function qtyBforA

- Minor Issue (Readability): choose a better name than `arg0`
- Minor Issue (readability): see Naming of Constants (3.1.4). A number is directly used in `require()`. It should be replaced by a constant.
- Minor Issue (readability): see Typography of Internal Functions (3.1.3). The function name should start with `_`.

```

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]);
    }

```

6.9.13 Function thisBalance

- Minor Issue (readability): see Typography of Internal Functions (3.1.3). The function name should start with `_`.

```

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

```

Chapter 7

Contract DEXroot

Contents

7.1	Contract Inheritance	61
7.2	Type Definitions	61
7.2.1	Struct Pair	61
7.3	Constant Definitions	61
7.4	Static Variable Definitions	61
7.5	Variable Definitions	62
7.6	Modifier Definitions	62
7.6.1	Modifier alwaysAccept	62
7.6.2	Modifier checkOwnerAndAccept	63
7.6.3	Modifier checkCreatorAndAccept	63
7.7	Constructor Definitions	63
7.7.1	Constructor	63
7.8	Public Method Definitions	63
7.8.1	Receive function	63
7.8.2	Function checkPubKey	64
7.8.3	Function createDEXclient	64
7.8.4	Function createDEXpair	65
7.8.5	Function getBalanceTONgrams	67
7.8.6	Function getClientAddress	67
7.8.7	Function getConnectorAddress	67
7.8.8	Function getPairAddress	67
7.8.9	Function getPairByRoots01	68
7.8.10	Function getPairByRoots10	68
7.8.11	Function getRootTokenAddress	68
7.8.12	Function getRootsByPair	69
7.8.13	Function sendTransfer	69
7.8.14	Function setCreator	69

7.8.15	Function setDEXclientCode	69
7.8.16	Function setDEXconnectorCode	70
7.8.17	Function setDEXpairCode	70
7.8.18	Function setRootTokenCode	70
7.8.19	Function setTONTOKENwalletCode	71
7.9	Internal Method Definitions	71
7.9.1	Function computeClientAddress	71
7.9.2	Function computeConnectorAddress	71
7.9.3	Function computePairAddress	72
7.9.4	Function computeRootTokenAddress	72

In file `DEXRoot.sol`

7.1 Contract Inheritance

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

7.2 Type Definitions

7.2.1 Struct Pair

- Minor Issue: keep naming should be consistent, `root0` and `root1`, or `rootA` and `rootB`, but not both. Idem for `rootLP` vs `rootAB`.

```

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

- Minor Issue: `soUINT` is never used, as in other contracts. It would be worth using the same number for all contracts derived from this root.
- Minor Issue (readability): see Typography of Static Variables (3.1.1). Static variables should start with `s_` for example.

```

13 uint256 static public soUINT;
```

7.5 Variable Definitions

- Minor Issue (readability): see Typography of Global Variables (3.1.2). Global variables should start with `m_` or `g_` for example.
- Minor Issue: naming should be more explicit. For example, `roots` could be renamed `pair_by_roots`, `pubkeys` and `clients` could be renamed `dexclient_by_pubkey` and `pubkey_by_dexclient`.

```

15  TvmCell public codeDEXclient;
16  TvmCell public codeDEXpair;
17  TvmCell public codeDEXconnector;
18  TvmCell public codeRootToken;
19  TvmCell public codeTONTTokenWallet;
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

- **Major issue: Accept-All Modifier in DEXroot**
See Accept Methods without Checks (3.2.1). This modifier should be removed.

```

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

```

7.6.2 Modifier checkOwnerAndAccept

- Minor Issue (readability): see Naming of Constants (3.1.4). A number is directly used in `require()`. It should be replaced by a constant.

```

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

```

7.6.3 Modifier checkCreatorAndAccept

Major issue: Accept-All Modifier in DEXroot

- See Accept Methods without Checks (3.2.1). The check on pubkey performed by this modifier is too weak to limit `tvml.accept`.
- Minor Issue (readability): see Naming of Constants (3.1.4). A number is directly used in `require()`. It should be replaced by a constant.

```

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

```

7.7 Constructor Definitions

7.7.1 Constructor

- Minor Issue (readability): see Naming of Constants (3.1.4). A number should be named through a constant.

```

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

```

7.8 Public Method Definitions

7.8.1 Receive function

- OK

```

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

```

7.8.2 Function checkPubKey

Major issue: Accept-All Method in DEXroot.checkPubKey

- See No Accept-All Methods (3.2.1) The balance of the contract could be drained by sending many checkPubKey messages. This method could be a get-method without tvvm.accept, to be executed locally.

```

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

```

7.8.3 Function createDEXclient

Major issue: Accept-All Method in DEXroot.createDEXclient

See No Accept-All Methods (3.2.1)

- - The balance of the contract could be drained, by sending many unexpected createDEXclient messages by an attacker.
 - The balance could be further drained by the require() failing after tvvm.accept, causing replication on all shards.
- Minor Issue: the method should check for correct initialization of the codeDEXclient, for example by using a bitmap set by all setDEX*code methods.
- Minor Issue (readability): see Better Units for Big Numbers (3.1.5). Use ton unit to make big numbers easier to understand.. Use 0.0031 ton instead of 3100000.
- Minor Issue (readability): see Naming of Constants (3.1.4). A number is directly used in require(). It should be replaced by a constant.

```

122 function createDEXclient(uint256 pubkey, uint256 souint) public
123     alwaysAccept returns (address deployedAddress, bool
124     statusCreate){
125     statusCreate = false;
126     deployedAddress = address(0);
127     uint128 prepay = balanceOf[creators[pubkey]];
128     require (!pubkeys.exists(pubkey) && !(prepay <
129     GRAMS_CREATE_DEX_CLIENT), 106);
130     delete balanceOf[creators[pubkey]];
131     TvmmCell stateInit = tvmm.buildStateInit({
132     contr: DEXClient,
133     varInit: {rootDEX: address(this), soUINT: souint,
134     codeDEXConnector: codeDEXconnector},
135     code: codeDEXclient,
136     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

Critical issue: Ownership of Liquidity Root Contract

This method allows ANY DEXClient to generate any non-existent DEXPair, including its liquidity root contract, passing all the static attributes, such as `name`, `symbol`, `decimals` and the `_randomNonce`, as arguments of the call. An attacker could use this possibility to take ownership of the liquidity root contract: the

- attacker would deploy that contract before the call to `createDEXpair`, but with constructor arguments giving him ownership of the contract; then, during the call to `createDEXpair`, the deployment would fail, but the contract would not notice it, and the pair would be created with the attacker-owned liquidity root contract. The fix would be to define the owner of the contract as a static variable in `RootTokenContract`.

- Minor Issue: the method should check for correct initialization of the `codeDEXpair` and `codeRootToken`, for example by using a bitmap set by all `setDEX*code` methods.
- Minor Issue (readability): see Naming of Constants (3.1.4). A number is directly used in `require()`. It should be replaced by a constant.
- Minor Issue (readability): see Better Units for Big Numbers (3.1.5). Use `ton` unit to make big numbers easier to understand.

```

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

```

```

301     cp.rootLP = rootAddress;
302     pairs[pairAddress] = cp;
303     pairKeys.push(pairAddress);
304     msg.sender.transfer({ value: 0, flag: 128});
305   } else {
306     msg.sender.transfer({ value: 0, flag: 128});
307   }
308 }

```

7.8.5 Function getBalanceTONgrams

Major issue: Accept-All Method in DEXroot.getBalanceTONgrams

- See No Accept-All Methods (3.2.1) The balance of the contract could be drained by sending many getBalanceTONgrams messages. This method could be a get-method without tvn.accept, to be executed locally.

```

334 function getBalanceTONgrams() public pure alwaysAccept returns (
335     uint128 balanceTONgrams){
336     return address(this).balance;
337 }

```

7.8.6 Function getClientAddress

- OK

```

118 function getClientAddress(uint256 clientPubKey, uint256
119     clientSoArg) public view responsible returns (address) {
120     return { value: 0, bounce: false, flag: 64 }
121         computeClientAddress(clientPubKey, clientSoArg);
122 }

```

7.8.7 Function getConnectorAddress

- OK

```

233 function getConnectorAddress(uint256 connectorPubKey, uint256
234     connectorSoArg, address connectorCommander) public view
235     responsible returns (address) {
236     return { value: 0, bounce: false, flag: 64 }
237         computeConnectorAddress(connectorPubKey, connectorSoArg,
238             connectorCommander);
239 }

```

7.8.8 Function getPairAddress

- OK

```

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

Major issue: Accept-All Method in DEXroot.getPairByRoots01

- See No Accept-All Methods (3.2.1) The balance of the contract could be drained by sending many getPairByRoots01 messages. This method could be a get-method without tvn.accept, to be executed locally.

```

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

```

7.8.10 Function getPairByRoots10

Major issue: Accept-All Method in DEXroot.getPairByRoots10

- See No Accept-All Methods (3.2.1) The balance of the contract could be drained by sending many getPairByRoots10 messages. This method could be a get-method without tvn.accept, to be executed locally.

```

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

```

7.8.11 Function getRootTokenAddress

- OK

```

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,
221           rootSymbol, rootDecimals);
           }

```

7.8.12 Function getRootsByPair

Major issue: Accept-All Method in DEXroot.getRootsByPair

- See No Accept-All Methods (3.2.1) The balance of the contract could be drained by sending many getRootsByPair messages. This method could be a get-method without tvn.accept, to be executed locally.

```

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

- OK

```

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

```

7.8.14 Function setCreator

Critical issue: Impersonate Accounts in DEXroot.setCreator

- Anybody can impersonate himself as the owner of any giverAddr. Later, in createDEXclient, ownership of giverAddr can be used to spend the corresponding balanceOf to create a DEXClient using somebody else's funds.

```

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

```

7.8.15 Function setDEXclientCode

- Minor Issue: the code should be checked against a code hash, hardcoded in the code, or set either in the constructor or in the static variables.

- Minor Issue: a bitmap should be used to verify that the contract has been properly initialized, i.e. all the codes of the sub-contracts have been correctly set.

```

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

```

7.8.16 Function setDEXconnectorCode

- Minor Issue: the code should be checked against a code hash, hardcoded in the code, or set either in the constructor or in the static variables.
- Minor Issue: a bitmap should be used to verify that the contract has been properly initialized, i.e. all the codes of the sub-contracts have been correctly set.

```

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

```

7.8.17 Function setDEXpairCode

- Minor Issue: the code should be checked against a code hash, hardcoded in the code, or set either in the constructor or in the static variables.
- Minor Issue: a bitmap should be used to verify that the contract has been properly initialized, i.e. all the codes of the sub-contracts have been correctly set.

```

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

```

7.8.18 Function setRootTokenCode

- Minor Issue: the code should be checked against a code hash, hardcoded in the code, or set either in the constructor or in the static variables.
- Minor Issue: a bitmap should be used to verify that the contract has been properly initialized, i.e. all the codes of the sub-contracts have been correctly set.

```

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

```

7.8.19 Function setTONTokenWalletCode

- Minor Issue: the code should be checked against a code hash, hardcoded in the code, or set either in the constructor or in the static variables.
- Minor Issue: a bitmap should be used to verify that the contract has been properly initialized, i.e. all the codes of the sub-contracts have been correctly set.

```

96  function setTONTokenWalletCode(TvmCell code) public
    checkOwnerAndAccept {
97      codeTONTokenWallet = code;
98  }

```

7.9 Internal Method Definitions

7.9.1 Function computeClientAddress

- Minor Issue: Repeated Code. All the `tvm.buildStateInit` calls for various contracts could be put in internal functions, used by both deployment methods and compute-address methods.
- Minor Issue (readability): see Typography of Internal Functions (3.1.3). The function name should start with `_`.

```

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

- Minor Issue (readability): see Typography of Internal Functions (3.1.3). The function name should start with `_`.

```

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

- Minor Issue (readability): see Typography of Internal Functions (3.1.3).
The function name should start with ..

```

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 = tvml.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

- Minor Issue (readability): see Typography of Internal Functions (3.1.3).
The function name should start with ..

```

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 = tvml.buildStateInit({
190         contr: RootTokenContract,
191         varInit: {
192             _randomNonce: souint,
193             name: name,
194             symbol: symbol,
195             decimals: decimals,
196             wallet_code: codeTONTOKENWallet
197         },
198         code: codeRootToken,
199         pubkey : pubkey
200     });

```



```
201     return address(tvm.hash(stateInit));  
202 }
```

Chapter 8

Contract RootTokenContract

Contents

8.1	Contract Inheritance	75
8.2	Static Variable Definitions	75
8.3	Variable Definitions	75
8.4	Modifier Definitions	75
8.4.1	Modifier onlyOwner	75
8.4.2	Modifier onlyInternalOwner	76
8.5	Constructor Definitions	76
8.5.1	Constructor	76
8.6	Public Method Definitions	76
8.6.1	Fallback function	76
8.6.2	OnBounce function	77
8.6.3	Function deployEmptyWallet	77
8.6.4	Function deployWallet	78
8.6.5	Function getDetails	79
8.6.6	Function getTotalSupply	80
8.6.7	Function getVersion	80
8.6.8	Function getWalletAddress	80
8.6.9	Function getWalletCode	81
8.6.10	Function mint	81
8.6.11	Function proxyBurn	81
8.6.12	Function sendExpectedWalletAddress	82
8.6.13	Function sendPausedCallbackTo	82
8.6.14	Function sendSurplusGas	83
8.6.15	Function setPaused	83
8.6.16	Function tokensBurned	83

8.6.17	Function transferOwner	84
8.7	Internal Method Definitions	85
8.7.1	Function getExpectedWalletAddress	85
8.7.2	Function isExternalOwner	85
8.7.3	Function isInternalOwner	86
8.7.4	Function isOwner	86

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         total_supply = 0;
60         paused = false;
61         start_gas_balance = address(this).balance;

```

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);
254
255         tvml.rawReserve(address(this).balance - msg.value, 2);
256
257         address wallet = new TONTOKENWallet{
258             value: deploy_grams,
259             flag: 1,
260             code: wallet_code,
261             pubkey: wallet_public_key_,
262             varInit: {
263                 root_address: address(this),
264                 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             tvn.accept();
187         } else {
188             tvn.rawReserve(math.max(start_gas_balance, address(this)
189                 ).balance - msg.value), 2);
190         }
191
192         TvmCell stateInit = tvn.buildStateInit({
193             contr: TONTokenWallet,
194             varInit: {
195                 root_address: address(this),
196                 code: wallet_code,
197                 wallet_public_key: wallet_public_key_,
198                 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 TONTokenWallet{
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     ITONTokenWallet(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,
82                 symbol,
83                 decimals,
84                 root_public_key,
85                 root_owner_address,
86                 total_supply
87             );
88     }

```

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) ||
124                 (owner_address_.value == 0 && wallet_public_key_ !=
125                 0),
126                 RootTokenContractErrors.
127                 error_define_public_key_or_owner_address);
128             return { value: 0, bounce: false, flag: 64 }
129                 getExpectedWalletAddress(wallet_public_key_,
130                 owner_address_);
131         }

```


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;

```

8.6.10 Function mint

```

282     function mint(
283         uint128 tokens,
284         address to
285     )
286         override
287         external
288         onlyOwner
289     {
290         tvml.accept();
291
292         ITONTOKENWallet(to).accept(tokens);
293
294         total_supply += tokens;
295     }

```

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         tvml.rawReserve(address(this).balance - msg.value, 2);

```

```

319         address send_gas_to_ = send_gas_to;
320         address expectedWalletAddress = getExpectedWalletAddress(0,
321             sender_address);
322
323         if (send_gas_to.value == 0) {
324             send_gas_to_ = sender_address;
325         }
326
327         IBurnableByRootTokenWallet(expectedWalletAddress).
328             burnByRoot{value: 0, flag: 128}(
329             tokens,
330             send_gas_to_,
331             callback_address,
332             callback_payload
333         );
    }

```

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         tvn.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         );
    }

```

8.6.13 Function sendPausedCallbackTo

```

423     function sendPausedCallbackTo(
424         uint64 callback_id,

```

```

425     address callback_addr
426 )
427     override
428     external
429 {
430     tvm.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         tvm.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         tvm.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

```

```

450         != 0),
        RootTokenContractErrors.
            error_define_public_key_or_owner_address);
451     tvml.accept();
452     root_public_key = root_public_key_;
453     root_owner_address = root_owner_address_;
454 }

```

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 = tvml.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(tvm.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

Contents

9.1	Contract Inheritance	88
9.2	Static Variable Definitions	88
9.3	Variable Definitions	88
9.4	Modifier Definitions	88
9.4.1	Modifier onlyRoot	88
9.4.2	Modifier onlyOwner	89
9.4.3	Modifier onlyInternalOwner	89
9.5	Constructor Definitions	89
9.5.1	Constructor	89
9.6	Public Method Definitions	89
9.6.1	Fallback function	89
9.6.2	OnBounce function	90
9.6.3	Function accept	90
9.6.4	Function allowance	91
9.6.5	Function approve	91
9.6.6	Function balance	92
9.6.7	Function burnByOwner	92
9.6.8	Function burnByRoot	93
9.6.9	Function destroy	94
9.6.10	Function disapprove	94
9.6.11	Function getDetails	94
9.6.12	Function getVersion	95
9.6.13	Function getWalletCode	95
9.6.14	Function internalTransfer	95
9.6.15	Function internalTransferFrom	96
9.6.16	Function setBouncedCallback	98
9.6.17	Function setReceiveCallback	98

9.6.18	Function transfer	98
9.6.19	Function transferFrom	100
9.6.20	Function transferToRecipient	101
9.7	Internal Method Definitions	103
9.7.1	Function getExpectedAddress	103

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.
           sender) ||
605             (wallet_public_key != 0 && wallet_public_key == msg
               .pubkey()),
606             TONTOKENWalletErrors.
               error_message_sender_is_not_my_owner);
607     };
608 }

```

9.4.3 Modifier onlyInternalOwner

```

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

```

9.5 Constructor Definitions

9.5.1 Constructor

```

43     constructor() public {
44         require(wallet_public_key == tvn.pubkey() && (owner_address
           .value == 0 || wallet_public_key == 0));
45         tvn.accept();
46
47         allow_non_notifiable = true;
48
49         if (owner_address.value != 0) {
50             ITokenWalletDeployedCallback(owner_address).
               notifyWalletDeployed{value: 0.00001 ton, flag: 1}(
                 root_address);
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         tvml.accept();
655
656         uint32 functionId = body.decode(uint32);
657         if (functionId == tvml.functionId(ITONTTokenWallet.
            internalTransfer)) {
658             uint128 tokens = body.decode(uint128);
659             balance_ += tokens;
660
661             if (bounced_callback.value != 0) {
662                 tvml.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                 tvml.rawReserve(math.max(TONTTokenWalletConstants.
                    target_gas_balance, address(this).balance - msg
                    .value), 2);
672                 owner_address.transfer({ value: 0, flag: 128 });
673             }
674             } else if (functionId == tvml.functionId(
                IBurnableTokenRootContract.tokensBurned)) {
675                 balance_ += body.decode(uint128);
676                 if (owner_address.value != 0) {
677                     tvml.rawReserve(math.max(TONTTokenWalletConstants.
                        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         tvn.accept();
104         balance_ += tokens;
105     }

```

9.6.4 Function allowance

```

107     function allowance() override external view responsible returns
108         (AllowanceInfo) {
109             return { value: 0, bounce: false, flag: 64 } (allowance_.
110                 hasValue() ? allowance_.get() : AllowanceInfo(0,
111                     address.makeAddrStd(0, 0)));
112     }

```

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             TONTOKENWALLET_ERRORS.error_non_zero_remaining);
130         if (owner_address.value != 0) {
131             tvn.rawReserve(math.max(TONTOKENWALLET_CONSTANTS.
132                 target_gas_balance, address(this).balance - msg.
133                     value), 2);
134         } else {
135             tvn.accept();
136         }
137     }
138
139     if (allowance_.hasValue()) {
140         if (allowance_.get().remaining_tokens ==
141             remaining_tokens) {

```

```

137         allowance_.set(AllowanceInfo(tokens, spender));
138     }
139 } else {
140     allowance_.set(AllowanceInfo(tokens, spender));
141 }
142
143 if (owner_address.value != 0 ) {
144     msg.sender.transfer({ value: 0, flag: 128 });
145 }
146 }

```

9.6.6 Function balance

```

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

```

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);
482         require((owner_address.value != 0 && msg.value > 0) ||
483             (owner_address.value == 0 && grams <= address(this)
484                 .balance && grams > 0), TONTokenWalletErrors.
485                 error_low_message_value);
486
487         if (owner_address.value != 0 ) {
488             tvm.rawReserve(math.max(TONTokenWalletConstants.
489                 target_gas_balance, address(this).balance - msg.
490                 value), 2);
491             balance_ -= tokens;
492             IBurnableTokenRootContract(root_address)
493                 .tokensBurned({ value: 0, flag: 128, bounce: true }(
494                     tokens,
495                     wallet_public_key,

```

```

492         owner_address,
493         send_gas_to.value != 0 ? send_gas_to :
494             owner_address,
495         callback_address,
496         callback_payload
497     );
498 } else {
499     tvm.accept();
500     balance_ -= tokens;
501     IBurnableTokenRootContract(root_address)
502         .tokensBurned{ value: grams, bounce: true }(
503         tokens,
504         wallet_public_key,
505         owner_address,
506         send_gas_to.value != 0 ? send_gas_to : address(
507             this),
508         callback_address,
509         callback_payload
510     );
511 }

```

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_, TONTOKENWALLET_ERRORS.
528         error_not_enough_balance);
529
530     tvm.rawReserve(address(this).balance - msg.value, 2);
531
532     balance_ -= tokens;
533
534     IBurnableTokenRootContract(root_address)
535         .tokensBurned{ value: 0, flag: 128, bounce: true }(
536         tokens,
537         wallet_public_key,
538         owner_address,
539         send_gas_to,
540         callback_address,
541         callback_payload
542     );
543 }

```

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         tvn.accept();
593         selfdestruct(gas_dest);
594     }

```

9.6.10 Function disapprove

```

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

```

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(

```

```

74         root_address,
75         wallet_public_key,
76         owner_address,
77         balance_,
78         receive_callback,
79         bounced_callback,
80         allow_non_notifiable
81     );
82 }

```

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         tvM.rawReserve(reserve, 2);
401     } else {
402         tvM.rawReserve(address(this).balance - msg.value, 2);
403     }
404
405     balance_ += tokens;
406
407     if (notify_receiver && receive_callback.value != 0) {
408         ITokensReceivedCallback(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.
            error_no_allowance_set);
434         require(msg.sender == allowance_.get().spender,
            TONTokenWalletErrors.error_wrong_spender);
435         require(tokens <= allowance_.get().remaining_tokens,
            TONTokenWalletErrors.error_not_enough_allowance);
436         require(tokens <= balance_, TONTokenWalletErrors.
            error_not_enough_balance);
437         require(tokens > 0);
438         require(to != address(this), TONTokenWalletErrors.
            error_wrong_recipient);
439
440         if (owner_address.value != 0 ) {
441             uint128 reserve = math.max(TONTokenWalletConstants.
                target_gas_balance, address(this).balance - msg.
                value);
442             require(address(this).balance > reserve +
                TONTokenWalletConstants.target_gas_balance,
                TONTokenWalletErrors.error_low_message_value);
443             tvm.rawReserve(reserve, 2);
444             tvm.rawReserve(math.max(TONTokenWalletConstants.
                target_gas_balance, address(this).balance - msg.
                value), 2);
445         } else {
446             require(msg.value > TONTokenWalletConstants.
                target_gas_balance, TONTokenWalletErrors.
                error_low_message_value);
447             tvm.rawReserve(address(this).balance - msg.value, 2);
448         }
449
450         balance_ -= tokens;
451
452         allowance_.set(AllowanceInfo(allowance_.get().
            remaining_tokens - tokens, allowance_.get().spender));
453
454         ITONTokenWallet(to).internalTransfer{ value: 0, bounce:
            true, flag: 129 }(
455             tokens,
456             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         tvml.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         tvml.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             tvm.rawReserve(reserve, 2);
344
345             ITONTokenWallet(from).internalTransferFrom{ value: 0,
346                 flag: 129 }(
347                 to,
348                 tokens,
349                 send_gas_to.value != 0 ? send_gas_to :
350                     owner_address,
351                 notify_receiver,
352                 payload
353             );
354         } else {
355             require(address(this).balance > grams,
356                 TONTokenWalletErrors.error_low_message_value);
357             require(grams > TONTokenWalletConstants.
358                 target_gas_balance * 2, TONTokenWalletErrors.
359                 error_low_message_value);
360             tvm.accept();
361             ITONTokenWallet(from).internalTransferFrom{ value:
362                 grams, flag: 1 }(
363                 to,
364                 tokens,
365                 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_, TONTOKENWALLET_ERRORS.
189         error_not_enough_balance);
190     require(recipient_address.value == 0 ||
191         recipient_public_key == 0, TONTOKENWALLET_ERRORS.
192         error_wrong_recipient);
193
194     if (owner_address.value != 0) {
195         uint128 reserve = math.max(TONTOKENWALLET_CONSTANTS.
196             target_gas_balance, address(this).balance - msg.
197             value);
198         require(address(this).balance > reserve +
199             TONTOKENWALLET_CONSTANTS.target_gas_balance +
200             deploy_grams, TONTOKENWALLET_ERRORS.
201             error_low_message_value);
202         require(recipient_address != owner_address,
203             TONTOKENWALLET_ERRORS.error_wrong_recipient);
204         tvml.rawReserve(reserve, 2);
205     } else {
206         require(address(this).balance > deploy_grams +
207             transfer_grams, TONTOKENWALLET_ERRORS.
208             error_low_message_value);
209         require(transfer_grams > TONTOKENWALLET_CONSTANTS.
210             target_gas_balance, TONTOKENWALLET_ERRORS.
211             error_low_message_value);
212         require(recipient_public_key != wallet_public_key);
213         tvml.accept();
214     }
215
216     TvmCell stateInit = tvml.buildStateInit({
217         contr: TONTOKENWALLET,
218         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     }
```