Bytus Token Smart Contract Manual

Table of Contents

Introduction

The Bytus Token (BYTS) is an ERC20-compliant token implemented on the Ethereum blockchain. This manual provides detailed technical information about the smart contract implementation, focusing on its architecture, functions, and security features.

Technical Specifications

Token Details:

- **Name:** Bytus Token
- **Symbol:** BYTS
- **Decimals:** 3 (non-standard, different from typical 18)
- **Initial Supply:** 66,000,000 BYTS
- **Contract Standard:** ERC20
- **Solidity Version:** 0.8.20

Smart Contract Dependencies:

- OpenZeppelin Contracts v4.x
  - ERC20
  - ERC20Burnable

- Pausable
- Ownable

Smart Contract Architecture

The BytusToken contract architecture follows composition patterns using OpenZeppelin's contract inheritance structure:

Copy

```
OpenZeppelin Contracts
       │
       ├── ERC20 (Token standard implementation)
       │
       ├── ERC20Burnable (Token burning functionality)
       │
       ├── Pausable (Emergency pause mechanism)
       │
       ├── Ownable (Access control)
       │
       ↓
   BytusToken (Custom implementation)
```

Key Design Patterns

1. **Inheritance Pattern:** Extends functionality from established contract implementations
2. **Access Control Pattern:** Uses Ownable for administrative functions
3. **Circuit Breaker Pattern:** Implements Pausable for emergency situations
4. **Token Extension Pattern:** Extends standard ERC20 with burning capability

Contract Functions

Core ERC20 Functions

Function	Description	Access Control
`balanceOf(address)`	Returns token balance of an address	Public
`transfer(address, uint256)`	Transfers tokens to specified address	Public, Not Paused
`approve(address, uint256)`	Approves address to spend tokens	Public, Not Paused
`transferFrom(address, address, uint256)`	Transfers tokens on behalf of another address	Public, Not Paused
`allowance(address, address)`	Returns approved token amount	Public
`totalSupply()`	Returns total token supply	Public

Custom and Extended Functions

Function	Description	Access Control
`decimals()`	Returns token decimal places (3)	Public
`burn(uint256)`	Burns tokens from caller's address	Public, Not Paused
`burnFrom(address, uint256)`	Burns tokens from specified address	Public, Not Paused, Requires Allowance
`approveAndCall(address, uint256, bytes)`	Approves and calls contract in one step	Public, Not Paused

Administrative Functions

Function	Description	Access Control
`pause()`	Pauses all token transfers	Owner Only
`unpause()`	Unpauses all token transfers	Owner Only

Administrative Controls

Owner Privileges

The contract owner (deployer) has exclusive access to:

- Pause/unpause token transfers
- Transfer ownership to another address

Ownership Management

1. **Initial Ownership:** Set to contract deployer in constructor
2. **Ownership Transfer:** Using Ownable's transferOwnership function
3. **Ownership Renouncement:** Using Ownable's renounceOwnership function

Best Practices for Admin Control

- Consider using multi-signature wallet as owner
- Plan for ownership transitions in advance
- Document emergency procedures for pausing

Security Features

Pause Mechanism

The pause mechanism serves as an emergency circuit breaker:

1. **Implementation:**

solidity

Copy

```solidity
function _beforeTokenTransfer(address from, address to, uint256 amount) internal virtual ov
    super._beforeTokenTransfer(from, to, amount);
    require(!paused(), "Token transfers are paused");
}
```

2. **Affected Operations:**
   - All token transfers
   - Approval operations
   - Burning operations
3. **Not Affected:**
   - Read-only operations (balanceOf, totalSupply, etc.)
   - Administrative functions

Burn Functionality

Token burning permanently removes tokens from circulation:

1. **Direct Burning:**
   - `burn(uint256 amount)` - Burns from caller's balance

- Reduces totalSupply accordingly

2. **Delegated Burning:**
   - `burnFrom(address account, uint256 amount)` - Burns from specified address
   - Requires prior approval via approve or increaseAllowance
   - Decreases allowance automatically

Decimal Precision

The contract uses 3 decimal places instead of the standard 18:

solidity

Copy

```solidity
function decimals() public view virtual override returns (uint8) {
    return 3;
}
```

This affects:

- Token display in wallets and UIs
- Calculation of token amounts in applications
- Integration with other contracts

Deployment Process

Technical Deployment Steps

1. **Compilation:**

   bash

   Copy

   ```bash
   npx hardhat compile
   ```

2. **Deployment Script (deploy.js):**

javascript

```javascript
const { ethers } = require("hardhat");

async function main() {
  const [deployer] = await ethers.getSigners();
  console.log("Deploying contracts with account:", deployer.address);

  const BytusToken = await ethers.getContractFactory("BytusToken");
  const bytusToken = await BytusToken.deploy();

  await bytusToken.deployed();
  console.log("BytusToken deployed to:", bytusToken.address);
}

main()
  .then(() => process.exit(0))
  .catch((error) => {
    console.error(error);
    process.exit(1);
  });
```

3. **Local Deployment:**

bash

```bash
npx hardhat run scripts/deploy.js --network localhost
```

4. **Network Deployment:**

bash

```bash
npx hardhat run scripts/deploy.js --network goerli
```

Constructor Parameters

The contract constructor initializes with:

- Token name: "Bytus Token"
- Token symbol: "BYTS"
- Initial supply: 66,000,000 tokens
- Recipient of initial supply: Contract deployer

Contract Verification

Verification on Etherscan

For public networks, verify the contract code:

bash

```bash
npx hardhat verify --network goerli DEPLOYED_CONTRACT_ADDRESS
```

Verification Data Requirements

- Compiler version: 0.8.20
- Optimization: Yes (200 runs)
- Contract name: BytusToken
- Constructor arguments: None (handled internally)

Manual Verification Steps

1. Flatten the contract if needed:

   bash

   ```bash
   npx hardhat flatten contracts/BytusToken.sol > BytusTokenFlat.sol
   ```

2. Submit to Etherscan:
   - Upload BytusTokenFlat.sol
   - Select compiler version 0.8.20
   - Enable optimization at 200 runs
   - Verify and publish

Technical Integration Guide

Contract ABI

The contract ABI (Application Binary Interface) is required for interacting with the contract programmatically. The ABI is generated during compilation and can be found in:

```
artifacts/contracts/BytusToken.sol/BytusToken.json
```

Web3.js Integration

Example of connecting to the contract using Web3.js:

javascript

Copy

```javascript
const Web3 = require('web3');
const contractABI = require('./artifacts/contracts/BytusToken.sol/BytusToken.json').abi;

// Connect to network
const web3 = new Web3('http://localhost:8545'); // or your network endpoint

// Create contract instance
const bytusTokenAddress = '0x5FbDB2315678afecb367f032d93F642f64180aa3'; // contract address
const bytusToken = new web3.eth.Contract(contractABI, bytusTokenAddress);

// Example: Get total supply
bytusToken.methods.totalSupply().call()
  .then(supply => console.log('Total supply:', supply));
```

Ethers.js Integration

Example of connecting to the contract using Ethers.js:

javascript

Copy

```javascript
const { ethers } = require('ethers');
const contractABI = require('./artifacts/contracts/BytusToken.sol/BytusToken.json').abi;

// Connect to network
const provider = new ethers.providers.JsonRpcProvider('http://localhost:8545'); // or your netw

// Create contract instance
const bytusTokenAddress = '0x5FbDB2315678afecb367f032d93F642f64180aa3'; // contract address
const bytusToken = new ethers.Contract(bytusTokenAddress, contractABI, provider);

// Example: Get total supply
async function getTotalSupply() {
  const supply = await bytusToken.totalSupply();
  console.log('Total supply:', ethers.utils.formatUnits(supply, 3)); // Note: Using 3 decimals
}

getTotalSupply();
```

Important Notes for Integration

1. **Decimal Handling:** Always account for 3 decimal places instead of 18 when formatting amounts
2. **Pause Checks:** Check contract status before attempting transfers:

javascript

```javascript
const isPaused = await bytusToken.paused();
if (isPaused) {
  console.log('Contract is currently paused');
  return;
}
```

3. **Gas Considerations:** Provide sufficient gas for operations, especially for approve and burn functions

Contract Audit Considerations

Key Security Checkpoints

1. **Access Control:**
   - Verify Ownable is used correctly
   - Ensure critical functions are properly protected
2. **Numeric Operations:**
   - Check for overflow/underflow protection (SafeMath or Solidity 0.8.x)
   - Verify rounding behavior with 3 decimals
3. **External Interactions:**
   - Review approveAndCall for reentrancy risks
   - Check for proper event emissions
4. **Compliance:**
   - Ensure ERC20 standard compliance
   - Verify all required functions and events are implemented

Recommended Pre-Audit Preparations

1. **Documentation:**
   - Prepare detailed technical specification
   - Document all custom functions and behaviors
2. **Test Coverage:**
   - Implement comprehensive unit tests

- Include edge cases and failure scenarios

3. **Code Comments:**
   - Ensure clear comments for complex functions
   - Document security considerations inline
4. **Gas Optimization:**
   - Review for gas efficiency
   - Document any gas-intensive operations