STT Token Technical Specification

Document Information

- **Project**: SilverTimes Token (STT)
- **Version**: 1.0
- **Date**: November 12, 2025
- **Status**: Draft

Table of Contents

---

1. Overview

1.1 Purpose

The SilverTimes Token (STT) is an ERC-20 compatible token designed to represent digital silver assets with advanced governance and control features. The token implements industry-standard security patterns and administrative controls for institutional-grade asset management.

1.2 Token Details

- **Token Name**: SilverTimes Token
- **Token Symbol**: STT
- **Decimals**: 18
- **Initial Supply**: To be determined by governance
- **Token Standard**: ERC-20 with extensions
- **Blockchain**: Ethereum (with potential multi-chain deployment)

1.3 Business Requirements Summary

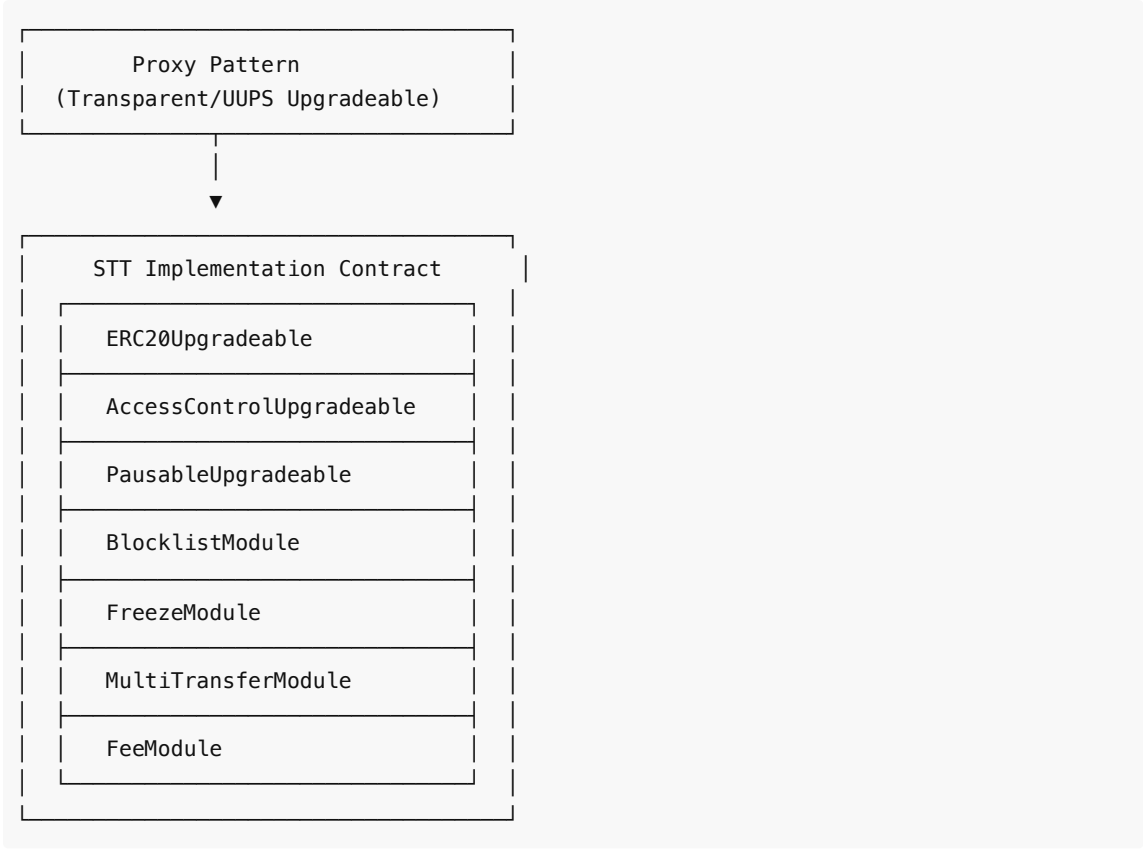
The STT token must support the following core functionalities:

- Upgradability for future enhancements
- Blocklist management for compliance
- Freeze functionality for security incidents
- Multi-transfer capabilities for efficiency
- Controlled minting and burning
- Transaction fee mechanism
- Emergency pause functionality

---

2. Technical Architecture

2.1 Smart Contract Architecture



2.2 Technology Stack

- **Language**: Solidity ^0.8.20
- **Framework**: Hardhat / Foundry
- **Upgradability**: OpenZeppelin Upgradeable Contracts
- **Testing**: Hardhat + Mocha/Chai or Foundry
- **Security**: Slither, Mythril, Manual Audit

2.3 Design Patterns

- **Proxy Pattern**: Transparent Proxy or UUPS (Universal Upgradeable Proxy Standard)
- **Access Control**: Role-Based Access Control (RBAC)
- **Circuit Breaker**: Pausable pattern for emergency situations
- **Checks-Effects-Interactions**: Preventing reentrancy attacks

---

3. Functional Requirements

3.1 Upgradability

3.1.1 Business Requirement

The smart contract must be upgradable to allow for future enhancements, bug fixes, and regulatory compliance updates without disrupting existing token holders.

3.1.2 Technical Implementation

Pattern: UUPS (Universal Upgradeable Proxy Standard) or Transparent Proxy Pattern

Key Components:

```
// Using OpenZeppelin UUPS pattern
contract STTToken is
    Initializable,
    ERC20Upgradeable,
    UUPSUpgradeable,
    AccessControlUpgradeable
{
    bytes32 public constant UPGRADER_ROLE = keccak256("UPGRADER_ROLE");

    function initialize() public initializer {
        __ERC20_init("SilverTimes Token", "STT");
        __UUPSUpgradeable_init();
        __AccessControl_init();

        _grantRole(DEFAULT_ADMIN_ROLE, msg.sender);
        _grantRole(UPGRADER_ROLE, msg.sender);
    }

    function _authorizeUpgrade(address newImplementation)
        internal
        override
        onlyRole(UPGRADER_ROLE)
    {}
}
```

Roles Required:

- `UPGRADER_ROLE` : Address authorized to upgrade the contract implementation

Storage Considerations:

- Must maintain storage layout compatibility across upgrades
- Use storage gaps for future extensions
- Document storage slots explicitly

Upgrade Process:

1. Deploy new implementation contract
2. Call `upgradeTo(newImplementation)` from UPGRADER_ROLE account
3. Verify upgrade through transparent testing
4. Monitor post-upgrade behavior

3.2 Blocklist

3.2.1 Business Requirement

The system must maintain a blocklist of addresses that are prohibited from sending, receiving, or holding tokens due to regulatory compliance, sanctions, or security concerns.

3.2.2 Technical Implementation

Storage:

```
// Mapping to track blocklisted addresses
mapping(address => bool) private _blocklisted;

// Event emissions
event AddressBlocklisted(address indexed account, string reason);
event AddressUnblocklisted(address indexed account);
```

Access Control:

```
bytes32 public constant BLOCKLIST_MANAGER_ROLE =
keccak256("BLOCKLIST_MANAGER_ROLE");
```

Core Functions:

```
/**
 * @dev Add address to blocklist
 * @param account Address to blocklist
 * @param reason Reason for blocklisting (for audit trail)
 */
function addToBlocklist(address account, string calldata reason)
    external
    onlyRole(BLOCKLIST_MANAGER_ROLE)
{
    require(account != address(0), "Cannot blocklist zero address");
    require(!_blocklisted[account], "Address already blocklisted");

    _blocklisted[account] = true;
    emit AddressBlocklisted(account, reason);
}

/**
 * @dev Remove address from blocklist
 * @param account Address to remove from blocklist
 */
function removeFromBlocklist(address account)
    external
    onlyRole(BLOCKLIST_MANAGER_ROLE)
{
    require(_blocklisted[account], "Address not blocklisted");

    _blocklisted[account] = false;
    emit AddressUnblocklisted(account);
}

/**
 * @dev Check if address is blocklisted
 * @param account Address to check
 * @return bool True if blocklisted
 */
function isBlocklisted(address account) public view returns (bool) {
```

```
    return _blocklisted[account];
}

/**
 * @dev Batch blocklist multiple addresses
 * @param accounts Array of addresses to blocklist
 * @param reason Reason for blocklisting
 */
function batchAddToBlocklist(address[] calldata accounts, string calldata reason)
    external
    onlyRole(BLOCKLIST_MANAGER_ROLE)
{
    for (uint256 i = 0; i < accounts.length; i++) {
        if (!_blocklisted[accounts[i]] && accounts[i] != address(0)) {
            _blocklisted[accounts[i]] = true;
            emit AddressBlocklisted(accounts[i], reason);
        }
    }
}
```

Transfer Override:

```
function _beforeTokenTransfer(
    address from,
    address to,
    uint256 amount
) internal virtual override {
    super._beforeTokenTransfer(from, to, amount);

    require(!_blocklisted[from], "Sender is blocklisted");
    require(!_blocklisted[to], "Recipient is blocklisted");
}
```

Roles Required:

- `BLOCKLIST_MANAGER_ROLE` : Authorized to manage blocklist

3.3 Freeze

3.3.1 Business Requirement

Ability to freeze individual accounts or the entire contract in case of security incidents, disputed transactions, or regulatory requirements.

3.3.2 Technical Implementation

Storage:

```
// Individual account freeze status
mapping(address => bool) private _frozenAccounts;

// Events
```

```
event AccountFrozen(address indexed account, string reason);
event AccountUnfrozen(address indexed account);
```

Access Control:

```
bytes32 public constant FREEZE_MANAGER_ROLE = keccak256("FREEZE_MANAGER_ROLE");
```

Core Functions:

```
/**
 * @dev Freeze an account
 * @param account Address to freeze
 * @param reason Reason for freezing
 */
function freezeAccount(address account, string calldata reason)
    external
    onlyRole(FREEZE_MANAGER_ROLE)
{
    require(account != address(0), "Cannot freeze zero address");
    require(!_frozenAccounts[account], "Account already frozen");

    _frozenAccounts[account] = true;
    emit AccountFrozen(account, reason);
}

/**
 * @dev Unfreeze an account
 * @param account Address to unfreeze
 */
function unfreezeAccount(address account)
    external
    onlyRole(FREEZE_MANAGER_ROLE)
{
    require(_frozenAccounts[account], "Account not frozen");

    _frozenAccounts[account] = false;
    emit AccountUnfrozen(account);
}

/**
 * @dev Check if account is frozen
 * @param account Address to check
 * @return bool True if frozen
 */
function isFrozen(address account) public view returns (bool) {
    return _frozenAccounts[account];
}

/**
 * @dev Batch freeze multiple accounts
 * @param accounts Array of addresses to freeze
```

```
 * @param reason Reason for freezing
 */
function batchFreezeAccounts(address[] calldata accounts, string calldata reason)
    external
    onlyRole(FREEZE_MANAGER_ROLE)
{
    for (uint256 i = 0; i < accounts.length; i++) {
        if (!_frozenAccounts[accounts[i]] && accounts[i] != address(0)) {
            _frozenAccounts[accounts[i]] = true;
            emit AccountFrozen(accounts[i], reason);
        }
    }
}
```

Transfer Override:

```
function _beforeTokenTransfer(
    address from,
    address to,
    uint256 amount
) internal virtual override {
    super._beforeTokenTransfer(from, to, amount);

    require(!_frozenAccounts[from], "Sender account is frozen");
    require(!_frozenAccounts[to], "Recipient account is frozen");
}
```

Roles Required:

- `FREEZE_MANAGER_ROLE` : Authorized to freeze/unfreeze accounts

3.4 Multi-Transfer

3.4.1 Business Requirement

Efficient batch transfer functionality to reduce gas costs and improve operational efficiency when distributing tokens to multiple recipients.

3.4.2 Technical Implementation

```
/**
 * @dev Transfer tokens to multiple recipients in a single transaction
 * @param recipients Array of recipient addresses
 * @param amounts Array of token amounts corresponding to recipients
 * @return success True if all transfers succeeded
 */
function multiTransfer(
    address[] calldata recipients,
    uint256[] calldata amounts
) external whenNotPaused returns (bool success) {
    require(
```

```
        recipients.length == amounts.length,
        "Recipients and amounts length mismatch"
    );
    require(recipients.length > 0, "Empty recipients array");
    require(recipients.length <= 200, "Too many recipients"); // Gas limit
protection

    uint256 totalAmount = 0;

    // Calculate total amount and validate inputs
    for (uint256 i = 0; i < recipients.length; i++) {
        require(recipients[i] != address(0), "Invalid recipient address");
        require(amounts[i] > 0, "Amount must be greater than 0");
        totalAmount += amounts[i];
    }

    // Check sender has sufficient balance
    require(balanceOf(msg.sender) >= totalAmount, "Insufficient balance");

    // Execute transfers
    for (uint256 i = 0; i < recipients.length; i++) {
        _transfer(msg.sender, recipients[i], amounts[i]);
    }

    emit MultiTransferExecuted(msg.sender, recipients.length, totalAmount);
    return true;
}

/**
 * @dev Transfer same amount to multiple recipients
 * @param recipients Array of recipient addresses
 * @param amount Amount to send to each recipient
 * @return success True if all transfers succeeded
 */
function multiTransferEqual(
    address[] calldata recipients,
    uint256 amount
) external whenNotPaused returns (bool success) {
    require(recipients.length > 0, "Empty recipients array");
    require(recipients.length <= 200, "Too many recipients");
    require(amount > 0, "Amount must be greater than 0");

    uint256 totalAmount = amount * recipients.length;
    require(balanceOf(msg.sender) >= totalAmount, "Insufficient balance");

    for (uint256 i = 0; i < recipients.length; i++) {
        require(recipients[i] != address(0), "Invalid recipient address");
        _transfer(msg.sender, recipients[i], amount);
    }

    emit MultiTransferExecuted(msg.sender, recipients.length, totalAmount);
    return true;
```

```
}

// Events
event MultiTransferExecuted(
    address indexed sender,
    uint256 recipientCount,
    uint256 totalAmount
);
```

Gas Optimization Considerations:

- Limit maximum recipients per transaction (recommended: 200)
- Use calldata instead of memory for arrays
- Minimize storage operations
- Consider gas refund mechanisms

Safety Checks:

- Validate array lengths match
- Check for zero addresses
- Verify sufficient balance upfront
- Respect freeze and blocklist restrictions

3.5 Mint & Burn Control

3.5.1 Business Requirement

Controlled token creation (minting) and destruction (burning) to manage supply according to physical silver reserves and redemption processes.

3.5.2 Technical Implementation

Access Control:

```
bytes32 public constant MINTER_ROLE = keccak256("MINTER_ROLE");
bytes32 public constant BURNER_ROLE = keccak256("BURNER_ROLE");
```

Storage:

```
// Track total minted and burned for auditing
uint256 private _totalMinted;
uint256 private _totalBurned;

// Minting cap (optional)
uint256 private _mintingCap;
bool private _capEnabled;
```

Mint Functions:

```
/**
 * @dev Mint new tokens
 * @param to Recipient address
```

```
 * @param amount Amount to mint
 */
function mint(address to, uint256 amount)
    external
    onlyRole(MINTER_ROLE)
    whenNotPaused
{
    require(to != address(0), "Cannot mint to zero address");
    require(amount > 0, "Amount must be greater than 0");

    if (_capEnabled) {
        require(
            totalSupply() + amount <= _mintingCap,
            "Minting would exceed cap"
        );
    }

    _mint(to, amount);
    _totalMinted += amount;

    emit TokensMinted(to, amount, msg.sender);
}

/**
 * @dev Batch mint to multiple addresses
 * @param recipients Array of recipient addresses
 * @param amounts Array of amounts to mint
 */
function batchMint(
    address[] calldata recipients,
    uint256[] calldata amounts
) external onlyRole(MINTER_ROLE) whenNotPaused {
    require(
        recipients.length == amounts.length,
        "Array length mismatch"
    );
    require(recipients.length > 0, "Empty arrays");
    require(recipients.length <= 100, "Too many recipients");

    uint256 totalAmount = 0;
    for (uint256 i = 0; i < amounts.length; i++) {
        totalAmount += amounts[i];
    }

    if (_capEnabled) {
        require(
            totalSupply() + totalAmount <= _mintingCap,
            "Batch mint would exceed cap"
        );
    }

    for (uint256 i = 0; i < recipients.length; i++) {
```

```
        require(recipients[i] != address(0), "Invalid recipient");
        require(amounts[i] > 0, "Amount must be > 0");

        _mint(recipients[i], amounts[i]);
        _totalMinted += amounts[i];
        emit TokensMinted(recipients[i], amounts[i], msg.sender);
    }
}

/**
 * @dev Set minting cap
 * @param cap Maximum total supply
 */
function setMintingCap(uint256 cap, bool enabled)
    external
    onlyRole(DEFAULT_ADMIN_ROLE)
{
    _mintingCap = cap;
    _capEnabled = enabled;
    emit MintingCapUpdated(cap, enabled);
}
```

Burn Functions:

```
/**
 * @dev Burn tokens from caller's balance
 * @param amount Amount to burn
 */
function burn(uint256 amount) external whenNotPaused {
    require(amount > 0, "Amount must be greater than 0");

    _burn(msg.sender, amount);
    _totalBurned += amount;

    emit TokensBurned(msg.sender, amount, msg.sender);
}

/**
 * @dev Burn tokens from specified account (requires approval or BURNER_ROLE)
 * @param from Account to burn from
 * @param amount Amount to burn
 */
function burnFrom(address from, uint256 amount)
    external
    whenNotPaused
{
    require(amount > 0, "Amount must be greater than 0");
    require(from != address(0), "Cannot burn from zero address");

    if (!hasRole(BURNER_ROLE, msg.sender)) {
        // If not BURNER_ROLE, check allowance
```

```
        uint256 currentAllowance = allowance(from, msg.sender);
        require(
            currentAllowance >= amount,
            "Burn amount exceeds allowance"
        );
        _approve(from, msg.sender, currentAllowance - amount);
    }

    _burn(from, amount);
    _totalBurned += amount;

    emit TokensBurned(from, amount, msg.sender);
}

/**
 * @dev Force burn tokens from blocklisted/frozen accounts
 * @param from Account to burn from
 * @param amount Amount to burn
 */
function forceBurn(address from, uint256 amount)
    external
    onlyRole(BURNER_ROLE)
{
    require(
        _blocklisted[from] || _frozenAccounts[from],
        "Account must be blocklisted or frozen"
    );
    require(amount > 0, "Amount must be greater than 0");

    _burn(from, amount);
    _totalBurned += amount;

    emit TokensForceBurned(from, amount, msg.sender);
}
```

View Functions:

```
function totalMinted() external view returns (uint256) {
    return _totalMinted;
}

function totalBurned() external view returns (uint256) {
    return _totalBurned;
}

function mintingCap() external view returns (uint256 cap, bool enabled) {
    return (_mintingCap, _capEnabled);
}
```

Events:

```
event TokensMinted(address indexed to, uint256 amount, address indexed minter);
event TokensBurned(address indexed from, uint256 amount, address indexed burner);
event TokensForceBurned(address indexed from, uint256 amount, address indexed
burner);
event MintingCapUpdated(uint256 cap, bool enabled);
```

Roles Required:

- `MINTER_ROLE` : Authorized to mint new tokens
- `BURNER_ROLE` : Authorized to force burn from frozen/blocklisted accounts
- `DEFAULT_ADMIN_ROLE` : Can set minting cap

3.6 Transaction Fee

3.6.1 Business Requirement

Implement a configurable transaction fee mechanism where a percentage of each transfer is collected and sent to a designated treasury address.

3.6.2 Technical Implementation

Storage:

```
// Fee configuration
uint256 private _transferFeeBasisPoints; // Fee in basis points (100 = 1%)
address private _feeCollector;
uint256 private _totalFeesCollected;

// Fee exemptions
mapping(address => bool) private _feeExempt;

// Constants
uint256 private constant BASIS_POINTS_DIVISOR = 10000;
uint256 private constant MAX_FEE_BASIS_POINTS = 1000; // Max 10%
```

Access Control:

```
bytes32 public constant FEE_MANAGER_ROLE = keccak256("FEE_MANAGER_ROLE");
```

Core Functions:

```
/**
 * @dev Set transaction fee percentage
 * @param basisPoints Fee in basis points (100 = 1%, max 1000 = 10%)
 */
function setTransferFee(uint256 basisPoints)
    external
    onlyRole(FEE_MANAGER_ROLE)
{
    require(
        basisPoints <= MAX_FEE_BASIS_POINTS,
```

```
        "Fee exceeds maximum"
    );

    uint256 oldFee = _transferFeeBasisPoints;
    _transferFeeBasisPoints = basisPoints;

    emit TransferFeeUpdated(oldFee, basisPoints);
}

/**
 * @dev Set fee collector address
 * @param collector Address to receive fees
 */
function setFeeCollector(address collector)
    external
    onlyRole(FEE_MANAGER_ROLE)
{
    require(collector != address(0), "Invalid collector address");

    address oldCollector = _feeCollector;
    _feeCollector = collector;

    emit FeeCollectorUpdated(oldCollector, collector);
}

/**
 * @dev Exempt address from transaction fees
 * @param account Address to exempt
 * @param exempt True to exempt, false to remove exemption
 */
function setFeeExemption(address account, bool exempt)
    external
    onlyRole(FEE_MANAGER_ROLE)
{
    require(account != address(0), "Invalid address");

    _feeExempt[account] = exempt;
    emit FeeExemptionUpdated(account, exempt);
}

/**
 * @dev Check if address is fee exempt
 * @param account Address to check
 * @return bool True if exempt
 */
function isFeeExempt(address account) external view returns (bool) {
    return _feeExempt[account];
}

/**
 * @dev Get current fee configuration
 * @return basisPoints Fee in basis points
```

```
 * @return collector Fee collector address
 * @return totalCollected Total fees collected
 */
function getFeeInfo() external view returns (
    uint256 basisPoints,
    address collector,
    uint256 totalCollected
) {
    return (_transferFeeBasisPoints, _feeCollector, _totalFeesCollected);
}

/**
 * @dev Calculate fee for a transfer amount
 * @param amount Transfer amount
 * @return fee Fee amount
 * @return netAmount Amount after fee
 */
function calculateFee(uint256 amount) public view returns (
    uint256 fee,
    uint256 netAmount
) {
    if (_transferFeeBasisPoints == 0) {
        return (0, amount);
    }

    fee = (amount * _transferFeeBasisPoints) / BASIS_POINTS_DIVISOR;
    netAmount = amount - fee;

    return (fee, netAmount);
}
```

Transfer Override with Fee Logic:

```
function _transfer(
    address from,
    address to,
    uint256 amount
) internal virtual override {
    require(from != address(0), "Transfer from zero address");
    require(to != address(0), "Transfer to zero address");
    require(amount > 0, "Amount must be greater than 0");

    // Check if fee should be applied
    bool applyFee = _transferFeeBasisPoints > 0
        && _feeCollector != address(0)
        && !_feeExempt[from]
        && !_feeExempt[to]
        && from != _feeCollector
        && to != _feeCollector;

    if (applyFee) {
```

```
        (uint256 fee, uint256 netAmount) = calculateFee(amount);

        // Transfer net amount to recipient
        super._transfer(from, to, netAmount);

        // Transfer fee to collector
        if (fee > 0) {
            super._transfer(from, _feeCollector, fee);
            _totalFeesCollected += fee;

            emit TransferFeeCollected(from, to, fee, netAmount);
        }
    } else {
        // Standard transfer without fee
        super._transfer(from, to, amount);
    }
}
```

Events:

```
event TransferFeeUpdated(uint256 oldFee, uint256 newFee);
event FeeCollectorUpdated(address indexed oldCollector, address indexed
newCollector);
event FeeExemptionUpdated(address indexed account, bool exempt);
event TransferFeeCollected(
    address indexed from,
    address indexed to,
    uint256 fee,
    uint256 netAmount
);
```

Roles Required:

- `FEE_MANAGER_ROLE` : Authorized to configure fees and exemptions

Important Notes:

- Fees are calculated in basis points (1 basis point = 0.01%)
- Maximum fee is capped at 10% (1000 basis points)
- Fee collector and contract addresses should be fee-exempt
- Minting and burning operations do not incur fees

3.7 Pause

3.7.1 Business Requirement

Emergency pause functionality to halt all token transfers and operations in case of security incidents, smart contract vulnerabilities, or regulatory requirements.

3.7.2 Technical Implementation

Using OpenZeppelin Pausable:

```
import "@openzeppelin/contracts-upgradeable/security/PausableUpgradeable.sol";

contract STTToken is
    ERC20Upgradeable,
    PausableUpgradeable,
    AccessControlUpgradeable
{
    bytes32 public constant PAUSER_ROLE = keccak256("PAUSER_ROLE");

    /**
     * @dev Pause all token operations
     */
    function pause() external onlyRole(PAUSER_ROLE) {
        _pause();
    }

    /**
     * @dev Unpause token operations
     */
    function unpause() external onlyRole(PAUSER_ROLE) {
        _unpause();
    }

    /**
     * @dev Check if contract is paused
     * @return bool True if paused
     */
    function isPaused() external view returns (bool) {
        return paused();
    }
}
```

Protected Operations: All the following operations are automatically blocked when paused:

```
function _beforeTokenTransfer(
    address from,
    address to,
    uint256 amount
) internal virtual override {
    super._beforeTokenTransfer(from, to, amount);

    require(!paused(), "Token transfers are paused");
    // ... other checks
}

// Explicit pause checks on critical functions
function mint(address to, uint256 amount)
    external
    onlyRole(MINTER_ROLE)
    whenNotPaused  // <-- Pause modifier
{
```

```
    // ...
}

function burn(uint256 amount)
    external
    whenNotPaused  // <-- Pause modifier
{
    // ...
}

function multiTransfer(
    address[] calldata recipients,
    uint256[] calldata amounts
) external whenNotPaused  // <-- Pause modifier
{
    // ...
}
```

View Functions (Always Available): These functions remain accessible even when paused:

- `balanceOf(address)`
- `totalSupply()`
- `allowance(address, address)`
- `isBlocklisted(address)`
- `isFrozen(address)`
- `getFeeInfo()`
- All other view/pure functions

Emergency Admin Functions (Available When Paused):

```
/**
 * @dev Emergency withdrawal for paused contract
 * Only callable by admin when paused
 */
function emergencyWithdraw(address token, uint256 amount)
    external
    onlyRole(DEFAULT_ADMIN_ROLE)
    whenPaused
{
    // Allow admin to rescue tokens if needed
    IERC20(token).transfer(msg.sender, amount);
    emit EmergencyWithdrawal(token, amount);
}
```

Events:

```
event Paused(address account);
event Unpaused(address account);
event EmergencyWithdrawal(address indexed token, uint256 amount);
```

Roles Required:

- `PAUSER_ROLE` : Authorized to pause/unpause the contract

Pause Scenarios:

1. **Security Incident**: Vulnerability discovered in contract logic
2. **Suspicious Activity**: Unusual transaction patterns detected
3. **Regulatory Compliance**: Required by regulatory authority
4. **System Maintenance**: Planned upgrade or maintenance
5. **Oracle Failure**: Price feed or external dependency failure

---

4. Smart Contract Specifications

4.1 Complete Contract Structure

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

import "@openzeppelin/contracts-upgradeable/token/ERC20/ERC20Upgradeable.sol";
import "@openzeppelin/contracts-upgradeable/access/AccessControlUpgradeable.sol";
import "@openzeppelin/contracts-upgradeable/security/PausableUpgradeable.sol";
import "@openzeppelin/contracts-upgradeable/proxy/utils/Initializable.sol";
import "@openzeppelin/contracts-upgradeable/proxy/utils/UUPSUpgradeable.sol";

/**
 * @title SilverTimes Token (STT)
 * @author SilverTimes Team
 * @notice ERC20 token representing digital silver with advanced governance
 * @dev Implements upgradability, blocklist, freeze, multi-transfer,
 *      controlled minting/burning, transaction fees, and pause functionality
 */
contract STTToken is
    Initializable,
    ERC20Upgradeable,
    AccessControlUpgradeable,
    PausableUpgradeable,
    UUPSUpgradeable
{
    // ============ Roles ============
    bytes32 public constant UPGRADER_ROLE = keccak256("UPGRADER_ROLE");
    bytes32 public constant PAUSER_ROLE = keccak256("PAUSER_ROLE");
    bytes32 public constant MINTER_ROLE = keccak256("MINTER_ROLE");
    bytes32 public constant BURNER_ROLE = keccak256("BURNER_ROLE");
    bytes32 public constant BLOCKLIST_MANAGER_ROLE =
keccak256("BLOCKLIST_MANAGER_ROLE");
    bytes32 public constant FREEZE_MANAGER_ROLE = keccak256("FREEZE_MANAGER_ROLE");
    bytes32 public constant FEE_MANAGER_ROLE = keccak256("FEE_MANAGER_ROLE");

    // ============ State Variables ============

    // Blocklist
    mapping(address => bool) private _blocklisted;
```

```
    // Freeze
    mapping(address => bool) private _frozenAccounts;

    // Minting/Burning tracking
    uint256 private _totalMinted;
    uint256 private _totalBurned;
    uint256 private _mintingCap;
    bool private _capEnabled;

    // Transaction Fees
    uint256 private _transferFeeBasisPoints;
    address private _feeCollector;
    uint256 private _totalFeesCollected;
    mapping(address => bool) private _feeExempt;
    uint256 private constant BASIS_POINTS_DIVISOR = 10000;
    uint256 private constant MAX_FEE_BASIS_POINTS = 1000;

    // ============ Events ============

    // Blocklist events
    event AddressBlocklisted(address indexed account, string reason);
    event AddressUnblocklisted(address indexed account);

    // Freeze events
    event AccountFrozen(address indexed account, string reason);
    event AccountUnfrozen(address indexed account);

    // Mint/Burn events
    event TokensMinted(address indexed to, uint256 amount, address indexed minter);
    event TokensBurned(address indexed from, uint256 amount, address indexed
burner);
    event TokensForceBurned(address indexed from, uint256 amount, address indexed
burner);
    event MintingCapUpdated(uint256 cap, bool enabled);

    // Multi-transfer events
    event MultiTransferExecuted(address indexed sender, uint256 recipientCount,
uint256 totalAmount);

    // Fee events
    event TransferFeeUpdated(uint256 oldFee, uint256 newFee);
    event FeeCollectorUpdated(address indexed oldCollector, address indexed
newCollector);
    event FeeExemptionUpdated(address indexed account, bool exempt);
    event TransferFeeCollected(address indexed from, address indexed to, uint256
fee, uint256 netAmount);

    // Emergency events
    event EmergencyWithdrawal(address indexed token, uint256 amount);

    // ============ Constructor & Initializer ============
```

```
/// @custom:oz-upgrades-unsafe-allow constructor
constructor() {
    _disableInitializers();
}

/**
 * @dev Initialize the contract
 * @param admin Address to receive admin role
 * @param feeCollector_ Initial fee collector address
 */
function initialize(address admin, address feeCollector_) public initializer {
    require(admin != address(0), "Invalid admin address");
    require(feeCollector_ != address(0), "Invalid fee collector");

    __ERC20_init("SilverTimes Token", "STT");
    __AccessControl_init();
    __Pausable_init();
    __UUPSUpgradeable_init();

    // Grant roles to admin
    _grantRole(DEFAULT_ADMIN_ROLE, admin);
    _grantRole(UPGRADER_ROLE, admin);
    _grantRole(PAUSER_ROLE, admin);
    _grantRole(MINTER_ROLE, admin);
    _grantRole(BURNER_ROLE, admin);
    _grantRole(BLOCKLIST_MANAGER_ROLE, admin);
    _grantRole(FREEZE_MANAGER_ROLE, admin);
    _grantRole(FEE_MANAGER_ROLE, admin);

    // Initialize fee collector
    _feeCollector = feeCollector_;
    _feeExempt[feeCollector_] = true;
    _feeExempt[admin] = true;

    // Set initial fee to 0
    _transferFeeBasisPoints = 0;
}

// ============ Upgrade Authorization ============

function _authorizeUpgrade(address newImplementation)
    internal
    override
    onlyRole(UPGRADER_ROLE)
{}

// ============ Blocklist Functions ============

// (Implementation as specified in section 3.2)

// ============ Freeze Functions ============
```

```
// (Implementation as specified in section 3.3)

// ============ Mint Functions ============

// (Implementation as specified in section 3.5)

// ============ Burn Functions ============

// (Implementation as specified in section 3.5)

// ============ Multi-Transfer Functions ============

// (Implementation as specified in section 3.4)

// ============ Fee Functions ============

// (Implementation as specified in section 3.6)

// ============ Pause Functions ============

// (Implementation as specified in section 3.7)

// ============ Transfer Override ============

/**
 * @dev Override transfer to implement all checks and fee logic
 */
function _beforeTokenTransfer(
    address from,
    address to,
    uint256 amount
) internal virtual override {
    super._beforeTokenTransfer(from, to, amount);

    // Pause check
    require(!paused(), "Token transfers are paused");

    // Blocklist checks
    require(!_blocklisted[from], "Sender is blocklisted");
    require(!_blocklisted[to], "Recipient is blocklisted");

    // Freeze checks
    require(!_frozenAccounts[from], "Sender account is frozen");
    require(!_frozenAccounts[to], "Recipient account is frozen");
}

/**
 * @dev Override transfer to implement fee logic
 */
function _transfer(
    address from,
```

```
        address to,
        uint256 amount
    ) internal virtual override {
        // Apply fee logic (implementation from section 3.6)
        // ...
    }

    // ============ Storage Gap ============

    /**
     * @dev Gap for future storage variables in upgrades
     */
    uint256[50] private __gap;
}
```

4.2 Role Management

Role Hierarchy:

```
DEFAULT_ADMIN_ROLE (Super Admin)
├── Can grant/revoke all roles
├── Can set minting cap
└── Emergency functions

UPGRADER_ROLE
└── Can upgrade contract implementation

PAUSER_ROLE
└── Can pause/unpause contract

MINTER_ROLE
└── Can mint new tokens

BURNER_ROLE
└── Can force burn from frozen/blocklisted accounts

BLOCKLIST_MANAGER_ROLE
└── Can add/remove addresses from blocklist

FREEZE_MANAGER_ROLE
└── Can freeze/unfreeze accounts

FEE_MANAGER_ROLE
└── Can configure transaction fees
```

Role Assignment Best Practices:

1. Use multi-sig wallet for `DEFAULT_ADMIN_ROLE`
2. Separate operational roles across different addresses
3. Implement time-locks for critical role operations
4. Regularly audit role assignments
5. Document all role changes in governance records

4.3 Gas Optimization

Optimization Strategies:

1. **Use calldata for read-only arrays**: Reduces gas by avoiding memory copies
2. **Batch operations**: Multi-transfer, batch blocklist, batch freeze
3. **Storage packing**: Pack related boolean flags into single slots
4. **Minimal storage writes**: Cache values in memory when possible
5. **Short-circuit logic**: Place cheaper checks first
6. **Event indexing**: Limit indexed parameters to essential queries

Estimated Gas Costs (approximate):

- Standard transfer: ~50,000 gas
- Transfer with fee: ~80,000 gas
- Multi-transfer (10 recipients): ~180,000 gas
- Mint: ~70,000 gas
- Burn: ~60,000 gas
- Freeze account: ~45,000 gas
- Blocklist address: ~45,000 gas
- Pause: ~30,000 gas

---

5. Security Considerations

5.1 Access Control

Security Measures:

1. Role-based access control (RBAC) using OpenZeppelin
2. Multi-signature requirement for admin operations
3. Time-locks on critical functions (upgrades, role changes)
4. Role separation (no single address should have all roles)

Recommendations:

- Use Gnosis Safe multi-sig for admin roles
- Implement 2-of-3 or 3-of-5 signature schemes
- Add 24-48 hour time-lock for upgrades
- Regular role audits and rotation

5.2 Reentrancy Protection

Built-in Protections:

1. Checks-Effects-Interactions pattern
2. OpenZeppelin's ReentrancyGuard (if needed for future extensions)
3. No external calls during balance updates

Critical Functions:

- All transfer functions follow CEI pattern
- Fee collection happens atomically
- No recursive calls allowed

5.3 Integer Overflow/Underflow

Protections:

- Solidity 0.8+ has built-in overflow checks
- SafeMath not required but can add extra validation
- Explicit bounds checking on fee calculations

5.4 Front-Running

Mitigation Strategies:

1. Fee changes emit events before taking effect
2. Consider time-delayed fee changes
3. Multi-transfer batching reduces MEV opportunities
4. Slippage protection in DEX integrations

5.5 Blocklist/Freeze Abuse

Safeguards:

1. Separate roles for blocklist and freeze operations
2. Events with reasons for audit trail
3. Off-chain governance for blocklist decisions
4. Regular review of frozen/blocklisted accounts
5. Appeal mechanism for users

5.6 Upgrade Safety

Upgrade Process:

1. Deploy new implementation to testnet
2. Comprehensive testing (unit, integration, fork)
3. Security audit of new implementation
4. Multi-sig approval for upgrade transaction
5. Time-lock before upgrade execution
6. Monitor contract behavior post-upgrade

Storage Safety:

- Maintain storage layout compatibility
- Use storage gaps for future variables
- Never change order of existing state variables
- Document storage layout explicitly

5.7 Fee Manipulation

Protections:

1. Maximum fee cap (10%)
2. Fee changes require privileged role
3. Events for all fee configuration changes
4. Fee-exempt addresses for operational accounts

---

6. Testing Requirements

6.1 Unit Tests

Coverage Requirements: Minimum 95% code coverage

Test Categories:

1. **Initialization Tests**
    - Proper role assignment
    - Initial state verification
    - Fee collector setup
2. **Transfer Tests**
    - Standard transfers
    - Transfers with fees
    - Fee-exempt transfers
    - Blocked/frozen address transfers
    - Zero amount transfers
    - Zero address transfers
3. **Blocklist Tests**
    - Add to blocklist
    - Remove from blocklist
    - Batch blocklist operations
    - Transfer from/to blocklisted address
    - Blocklist events
4. **Freeze Tests**
    - Freeze account
    - Unfreeze account
    - Batch freeze operations
    - Transfer from/to frozen account
    - Freeze events
5. **Multi-Transfer Tests**
    - Multi-transfer success
    - Multi-transfer with insufficient balance
    - Multi-transfer array mismatch
    - Multi-transfer gas limits
    - Multi-transfer events
6. **Mint/Burn Tests**
    - Mint tokens
    - Mint to zero address (should fail)
    - Mint with cap enabled
    - Mint exceeding cap (should fail)
    - Batch mint
    - Burn tokens
    - Burn from
    - Force burn
    - Burn events
7. **Fee Tests**

- Set transfer fee
- Set fee above maximum (should fail)
- Set fee collector
- Fee calculation accuracy
- Fee exemption
- Fee collection events

8. **Pause Tests**
    - Pause contract
    - Unpause contract
    - Operations while paused (should fail)
    - View functions while paused (should succeed)
9. **Upgrade Tests**
    - Upgrade authorization
    - Storage layout preservation
    - State preservation after upgrade
10. **Access Control Tests**
    - Role assignment
    - Role revocation
    - Unauthorized access attempts
    - Multi-role operations

6.2 Integration Tests

Test Scenarios:

1. Complete token lifecycle (mint → transfer → burn)
2. Fee collection workflow
3. Blocklist + freeze combination
4. Multi-transfer with fee collection
5. Pause → unfreeze → unpause scenario
6. Upgrade with state preservation

6.3 Fork Tests

Mainnet Fork Testing:

1. Deploy on Ethereum mainnet fork
2. Test interactions with DEXs (Uniswap, Curve)
3. Test oracle integration
4. Gas cost analysis on realistic scenarios

6.4 Security Audits

Audit Requirements:

1. **Internal Audit**: Development team review
2. **External Audit**: Professional security firm (e.g., OpenZeppelin, Trail of Bits, Consensys Diligence)
3. **Bug Bounty**: Public bug bounty program on ImmuneFi
4. **Continuous Monitoring**: Real-time monitoring with Forta or similar

Tools:

- Slither: Static analysis
- Mythril: Symbolic execution
- Echidna: Fuzzing
- Manticore: Dynamic analysis

---

7. Deployment Plan

7.1 Pre-Deployment

Checklist:

- [ ] All tests passing (100% critical path coverage)
- [ ] External security audit completed
- [ ] Documentation finalized
- [ ] Multi-sig wallets set up
- [ ] Role assignment plan documented
- [ ] Fee configuration decided
- [ ] Initial supply determined
- [ ] Testnet deployment successful

7.2 Testnet Deployment

Networks:

1. **Sepolia** (Ethereum testnet)
2. **Goerli** (Ethereum testnet - being deprecated)
3. **Mumbai** (Polygon testnet) - if multi-chain

Steps:

1. Deploy implementation contract
2. Deploy proxy contract
3. Initialize with test parameters
4. Verify contracts on Etherscan
5. Execute test transactions
6. Verify all features working
7. Perform load testing
8. Monitor for 1-2 weeks

7.3 Mainnet Deployment

Steps:

1. **Deploy Implementation**

   ```
   npx hardhat run scripts/deploy-implementation.ts --network mainnet
   ```

2. **Deploy Proxy**

   ```
   npx hardhat run scripts/deploy-proxy.ts --network mainnet
   ```

3. **Initialize Contract**

```
await proxy.initialize(
  adminMultisigAddress,
  feeCollectorAddress
);
```

4. **Verify Contracts**

```
npx hardhat verify --network mainnet IMPLEMENTATION_ADDRESS
npx hardhat verify --network mainnet PROXY_ADDRESS
```

5. **Configure Roles**
    - Grant MINTER_ROLE to treasury address
    - Grant PAUSER_ROLE to security team
    - Grant FEE_MANAGER_ROLE to operations team
    - etc.
6. **Set Initial Parameters**
    - Set fee collector address
    - Set initial transaction fee (if any)
    - Configure fee exemptions
    - Set minting cap (if applicable)
7. **Initial Token Distribution**
    - Mint initial supply
    - Distribute to stakeholders
    - Fund liquidity pools
8. **Post-Deployment Verification**
    - Verify all roles assigned correctly
    - Test all critical functions
    - Monitor contract behavior
    - Announce contract address publicly

7.4 Post-Deployment

Monitoring:

1. Set up Tenderly/Forta monitoring
2. Configure alerts for suspicious activity
3. Monitor gas usage and optimize
4. Track fee collection
5. Monitor total supply changes

Documentation:

1. Publish contract addresses
2. Publish ABI and documentation
3. Create integration guides
4. Publish security audit reports

Community:

1. Announce deployment on social channels
2. Publish verification guide
3. Set up support channels
4. Launch bug bounty program

---

8. API Reference

8.1 Core ERC-20 Functions

```
function name() public view returns (string memory)
function symbol() public view returns (string memory)
function decimals() public view returns (uint8)
function totalSupply() public view returns (uint256)
function balanceOf(address account) public view returns (uint256)
function transfer(address to, uint256 amount) public returns (bool)
function allowance(address owner, address spender) public view returns (uint256)
function approve(address spender, uint256 amount) public returns (bool)
function transferFrom(address from, address to, uint256 amount) public returns
(bool)
```

8.2 Blocklist Functions

```
function addToBlocklist(address account, string calldata reason) external
function removeFromBlocklist(address account) external
function isBlocklisted(address account) public view returns (bool)
function batchAddToBlocklist(address[] calldata accounts, string calldata reason)
external
```

8.3 Freeze Functions

```
function freezeAccount(address account, string calldata reason) external
function unfreezeAccount(address account) external
function isFrozen(address account) public view returns (bool)
function batchFreezeAccounts(address[] calldata accounts, string calldata reason)
external
```

8.4 Multi-Transfer Functions

```
function multiTransfer(address[] calldata recipients, uint256[] calldata amounts)
external returns (bool)
function multiTransferEqual(address[] calldata recipients, uint256 amount) external
returns (bool)
```

8.5 Mint Functions

```
function mint(address to, uint256 amount) external
function batchMint(address[] calldata recipients, uint256[] calldata amounts)
external
function setMintingCap(uint256 cap, bool enabled) external
function totalMinted() external view returns (uint256)
function mintingCap() external view returns (uint256 cap, bool enabled)
```

8.6 Burn Functions

```
function burn(uint256 amount) external
function burnFrom(address from, uint256 amount) external
function forceBurn(address from, uint256 amount) external
function totalBurned() external view returns (uint256)
```

8.7 Fee Functions

```
function setTransferFee(uint256 basisPoints) external
function setFeeCollector(address collector) external
function setFeeExemption(address account, bool exempt) external
function isFeeExempt(address account) external view returns (bool)
function getFeeInfo() external view returns (uint256 basisPoints, address collector,
uint256 totalCollected)
function calculateFee(uint256 amount) public view returns (uint256 fee, uint256
netAmount)
```

8.8 Pause Functions

```
function pause() external
function unpause() external
function isPaused() external view returns (bool)
```

8.9 Role Management

```
function grantRole(bytes32 role, address account) external
function revokeRole(bytes32 role, address account) external
function renounceRole(bytes32 role, address account) external
function hasRole(bytes32 role, address account) public view returns (bool)
function getRoleAdmin(bytes32 role) public view returns (bytes32)
```

---

9. Appendix

9.1 Glossary

- **Basis Points**: 1/100th of a percent (100 bps = 1%)
- **Blocklist**: List of addresses prohibited from interacting with the token
- **Burn**: Permanent destruction of tokens

- **CEI Pattern**: Checks-Effects-Interactions security pattern
- **ERC-20**: Ethereum token standard
- **Freeze**: Temporary suspension of an account's ability to transfer
- **Mint**: Creation of new tokens
- **Multi-sig**: Multi-signature wallet requiring multiple approvals
- **Pausable**: Ability to temporarily halt all contract operations
- **Proxy Pattern**: Upgradability pattern separating logic from storage
- **RBAC**: Role-Based Access Control
- **UUPS**: Universal Upgradeable Proxy Standard

9.2 References

1. OpenZeppelin Contracts: https://docs.openzeppelin.com/contracts/
2. EIP-20 (ERC-20): https://eips.ethereum.org/EIPS/eip-20
3. EIP-1967 (Proxy Storage): https://eips.ethereum.org/EIPS/eip-1967
4. Solidity Documentation: https://docs.soliditylang.org/
5. Ethereum Yellow Paper: https://ethereum.github.io/yellowpaper/

END OF DOCUMENT