

Technical Contracts Documentation - QuantPayChain MVP

⚠ AUTOMATIC TRANSLATION — REVIEW REQUIRED

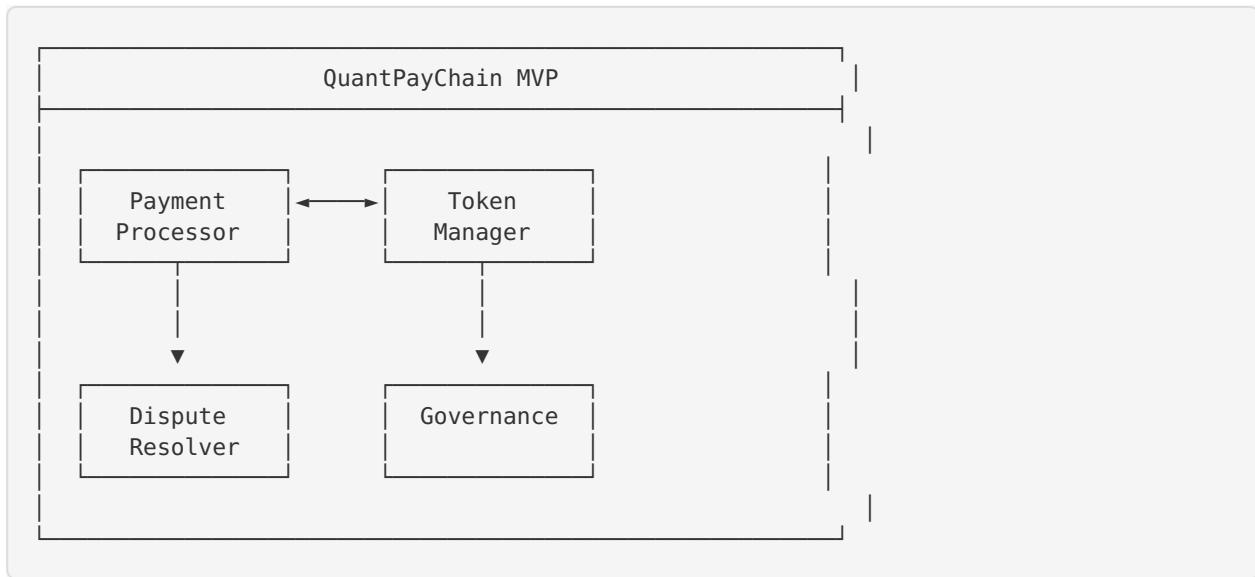
This document has been automatically translated from Spanish. Technical review is recommended for accuracy.

Table of Contents

1. General Architecture
2. PaymentProcessor
3. TokenManager
4. DisputeResolver
5. Governance
6. Security and Audit
7. Gas Optimization

General Architecture

Component Diagram



Design Principles

- **Modularity:** Each contract has well-defined responsibilities
- **Upgradability:** Design prepared for future updates
- **Security First:** Implementation of OpenZeppelin best practices
- **Gas Efficiency:** Optimization of expensive operations
- **PQC Ready:** Architecture prepared for post-quantum cryptography

PaymentProcessor

Description

Main contract for payment management with escrow and dispute resolution functionality.

Key Features

- Payment creation with automatic escrow
- Fund release with confirmation
- Refund system
- Integration with DisputeResolver
- Detailed events for tracking

Data Structures

Payment Struct

```
struct Payment {
    address payer;          // Payer address
    address payee;          // Receiver address
    uint256 amount;          // Amount in wei
    uint256 timestamp;        // Creation timestamp
    PaymentStatus status;      // Payment status
    string description;      // Payment description
}

enum PaymentStatus {
    PENDING,      // Payment created, funds in escrow
    COMPLETED,    // Payment completed, funds released
    REFUNDED,     // Payment refunded
    DISPUTED      // Payment in dispute
}
```

Main Functions

createPayment

```
function createPayment(
    address _payee,
    string memory _description
) external payable returns (uint256 paymentId)
```

Description: Creates a new payment and locks funds in escrow.

Parameters:

- `_payee` : Payment receiver address
- `_description` : Payment description

Requirements:

- `msg.value > 0` : Must send ETH
- `_payee != address(0)` : Valid address

Events: `PaymentCreated(paymentId, payer, payee, amount)`

completePayment

```
function completePayment(uint256 _paymentId) external
```

Description: Completes a payment and releases funds to receiver.

Parameters:

- `_paymentId` : Payment ID to complete

Requirements:

- Only payer can complete
- Status must be PENDING
- Must not be in dispute

Events: `PaymentCompleted(paymentId, amount)`

refundPayment

```
function refundPayment(uint256 _paymentId) external
```

Description: Refunds a payment to payer.

Parameters:

- `_paymentId` : Payment ID to refund

Requirements:

- Only receiver can initiate refund
- Status must be PENDING
- Must not be in dispute

Events: `PaymentRefunded(paymentId, amount)`

raiseDispute

```
function raiseDispute(
    uint256 _paymentId,
    string memory _reason
) external
```

Description: Initiates a dispute about a payment.

Parameters:

- `_paymentId` : Disputed payment ID
- `_reason` : Dispute reason

Requirements:

- Only payer or receiver can initiate
- Status must be PENDING
- Must not have active dispute

Events: `DisputeRaised(paymentId, initiator, reason)`

Security

Reentrancy Protection

```
// Using OpenZeppelin's ReentrancyGuard
modifier nonReentrant() {
    require(_status != _ENTERED, "ReentrancyGuard: reentrant call");
    _status = _ENTERED;
    ;
    _status = _NOT_ENTERED;
}
```

Access Control

```
// Role verification
modifier onlyPayer(uint256 _paymentId) {
    require(payments[_paymentId].payer == msg.sender, "Not payer");
    ;
}

modifier onlyPayee(uint256 _paymentId) {
    require(payments[_paymentId].payee == msg.sender, "Not payee");
    ;
}
```

TokenManager

Description

ERC-20 token management with advanced transfer and allowance features.

Key Features

- Complete ERC-20 implementation
- Secure allowance system
- Controlled minting and burning
- Pausable in emergencies
- Integration with PaymentProcessor

Main Functions

mint

```
function mint(address _to, uint256 _amount) external onlyOwner
```

Description: Creates new tokens.

Parameters:

- `_to` : Receiver address
- `_amount` : Token quantity

Requirements:

- Only owner can execute
- Must not exceed maximum supply

burn

```
function burn(uint256 _amount) external
```

Description: Destroys caller's tokens.

Parameters:

- `_amount` : Amount to burn

Requirements:

- Caller must have sufficient balance

transferWithFee

```
function transferWithFee(
    address _to,
    uint256 _amount,
    uint256 _feePercentage
) external returns (bool)
```

Description: Transfer with automatic fee.

Parameters:

- `_to` : Destination address
- `_amount` : Amount to transfer
- `_feePercentage` : Fee percentage (basis points)

Requirements:

- Sufficient balance
- Fee must not exceed 10%

DisputeResolver

Description

Decentralized arbitration system for resolving payment disputes.

Key Features

- Arbitrator voting
- Evidence system
- Automatic resolution
- Penalties for misuse
- Dispute history

Data Structures

Dispute Struct

```
struct Dispute {
    uint256 paymentId;
    address initiator;
    string reason;
    DisputeStatus status;
    uint256 votesForPayer;
    uint256 votesForPayee;
    mapping(address => bool) hasVoted;
    uint256 createdAt;
    uint256 resolvedAt;
}

enum DisputeStatus {
    OPEN,           // Open dispute
    VOTING,         // Voting in progress
    RESOLVED_PAYER, // Resolved in favor of payer
    RESOLVED_PAYEE, // Resolved in favor of receiver
    CANCELLED       // Cancelled
}
```

Main Functions

submitEvidence

```
function submitEvidence(
    uint256 _disputeId,
    string memory _evidence
) external
```

Description: Submits evidence for a dispute.

Parameters:

- `_disputeId` : Dispute ID
- `_evidence` : Evidence in string/IPFS hash format

Requirements:

- Only involved parties
- Dispute must be OPEN

voteOnDispute

```
function voteOnDispute(
    uint256 _disputeId,
    bool _voteForPayer
) external
```

Description: Votes on a dispute (arbitrators only).

Parameters:

- `_disputeId` : Dispute ID
- `_voteForPayer` : true for payer, false for receiver

Requirements:

- Must be authorized arbitrator
- Must not have voted previously
- Dispute in VOTING status

resolveDispute

```
function resolveDispute(uint256 _disputeId) external
```

Description: Resolves a dispute based on votes.

Parameters:

- `_disputeId` : Dispute ID

Requirements:

- Voting period finished
- Quorum reached

Governance

Description

On-chain governance system for protocol decisions.

Key Features

- Proposal creation
- Token-weighted voting
- Timelock for execution
- Vote delegation
- Dynamic quorum

Data Structures

Proposal Struct

```
struct Proposal {
    uint256 id;
    address proposer;
    string description;
    uint256 forVotes;
    uint256 againstVotes;
    uint256 startBlock;
    uint256 endBlock;
    bool executed;
    bool cancelled;
    mapping(address => bool) hasVoted;
}
```

Main Functions

propose

```
function propose(
    address[] memory _targets,
    uint256[] memory _values,
    bytes[] memory _calldatas,
    string memory _description
) external returns (uint256 proposalId)
```

Description: Creates a new proposal.

Parameters:

- `_targets` : Contracts to call
- `_values` : ETH values to send
- `_calldatas` : Call data
- `_description` : Proposal description

Requirements:

- Have sufficient governance tokens
- Must not have active proposal

castVote

```
function castVote(
    uint256 _proposalId,
    bool _support
) external
```

Description: Votes on a proposal.

Parameters:

- `_proposalId` : Proposal ID
- `_support` : true for in favor, false against

Requirements:

- Active proposal
- Must not have voted previously
- Have governance tokens

execute

```
function execute(uint256 _proposalId) external
```

Description: Executes an approved proposal.

Parameters:

- `_proposalId` : Proposal ID

Requirements:

- Approved proposal
- Timelock expired
- Not previously executed

Security and Audit

Implemented Best Practices

1. Checks-Effects-Interactions Pattern

```
function completePayment(uint256 _paymentId) external {
    // Checks
    require(payments[_paymentId].status == PaymentStatus.PENDING);
    require(msg.sender == payments[_paymentId].payer);

    // Effects
    payments[_paymentId].status = PaymentStatus.COMPLETED;

    // Interactions
    (bool success, ) = payments[_paymentId].payee.call{
        value: payments[_paymentId].amount
    }("");
    require(success, "Transfer failed");
}
```

2. Reentrancy Guards

- Use of OpenZeppelin's `ReentrancyGuard`
- Applied to all functions with transfers

3. Access Control

- Roles defined with OpenZeppelin's `AccessControl`
- Permission verification in critical functions

4. Input Validation

```
require(_amount > 0, "Amount must be positive");
require(_address != address(0), "Invalid address");
require(_percentage <= 10000, "Percentage too high");
```

5. Emergency Pause

```
// Pausable in case of emergency
function pause() external onlyOwner {
    _pause();
}

function unpause() external onlyOwner {
    _unpause();
}
```

Mitigated Vulnerabilities

| Vulnerability | Mitigation |
|------------------|---------------------------------|
| Reentrancy | ReentrancyGuard + CEI pattern |
| Integer Overflow | Solidity 0.8.x built-in checks |
| Front-running | Commit-reveal in voting |
| Access Control | OpenZeppelin AccessControl |
| DoS | Gas limits and circuit breakers |

Test Suite

Coverage

- **Total tests:** 59
- **Line coverage:** >95%
- **Branch coverage:** >90%

Test Categories

1. **Unit Tests:** Individual functions
2. **Integration Tests:** Contract interaction
3. **Edge Cases:** Boundary cases and errors
4. **Gas Tests:** Gas optimization
5. **Security Tests:** Attack vectors

Gas Optimization

Implemented Techniques

1. Storage Packing

```
// Before (3 slots)
uint256 amount;
address user;
bool active;

// After (2 slots)
address user;          // 20 bytes
uint96 amount;         // 12 bytes
bool active;           // 1 byte
```

2. Calldata vs Memory

```
// Use calldata for read-only parameters
function process(string calldata _data) external {
    // Cheaper than memory
}
```

3. Unchecked Arithmetic

```
// When overflow is impossible
unchecked {
    counter++;
}
```

4. Short-circuit Evaluation

```
// Cheaper conditions first
require(cheapCheck() && expensiveCheck());
```

Gas Benchmarks

| Operation | Gas Used | Optimization |
|-----------------|----------|------------------|
| createPayment | ~50,000 | Storage packing |
| completePayment | ~35,000 | Calldata usage |
| voteOnDispute | ~45,000 | Bitmap voting |
| propose | ~80,000 | Batch operations |

PQC Preparation

Future Integration Points

1. Hybrid Signature

```
// Placeholder for PQC signatures
struct HybridSignature {
    bytes ecdsaSignature;
    bytes pqcSignature; // Dilithium
}
```

2. Key Exchange

```
// Placeholder for Kyber KEM
struct KeyExchange {
    bytes publicKey;
    bytes encapsulatedKey;
}
```

3. Key Rotation

```
// Prepared rotation system
mapping(address => uint256) public keyVersion;
mapping(address => mapping(uint256 => bytes)) public publicKeys;
```

See [SECURITY-PQC.md](#) ([..//SECURITY-PQC.md](#)) for complete details.

Last updated: October 2025

Solidity version: 0.8.20

Framework: Foundry