```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.29;

/// @title Xolaxis TRX Escrow
/// @author Xolaxis
/// @notice Milestone escrow for native TRX (accounted in sun) on the TRON network.
/// @dev
///  - The contract is intentionally non-upgradeable.
///  - Every agreement snapshots its fee rate and fee recipient.
///  - Administrative recovery is limited to unaccounted surplus and cannot use
///    locked escrow or user withdrawal credits.
///  - Pausing blocks new obligations, but never blocks disputes, settlements,
///    refunds, or withdrawals.
contract XolaxisTRXEscrow {
    uint256 public constant BPS = 10_000;
    uint16 public constant MAX_PLATFORM_FEE_BPS = 500;
    uint16 public constant MAX_MILESTONES = 32;
    uint64 public constant MIN_REVIEW_PERIOD = 1 hours;
    uint64 public constant MAX_REVIEW_PERIOD = 30 days;
    uint64 public constant MAX_ACCEPTANCE_PERIOD = 30 days;
    uint64 public constant MAX_MILESTONE_HORIZON = 730 days;

    enum AgreementState {
        None,
        PendingAcceptance,
        Active,
        Completed,
        Cancelled
    }

    enum MilestoneState {
        Funded,
        Submitted,
        Disputed,
        Released,
        Refunded,
        Resolved
    }

    struct Agreement {
        address payable buyer;
        address payable seller;
        address arbitrator;
        address payable feeRecipient;
        uint256 totalValue;
        uint256 buyerRefunded;
        uint256 sellerGrossReleased;
        uint256 feePaid;
        uint64 createdAt;
        uint64 acceptanceDeadline;
        uint64 reviewPeriod;
        uint16 feeBps;
        uint16 milestoneCount;
        uint16 settledMilestones;
        AgreementState state;
    }

    struct Milestone {
        uint256 amount;
```

```
    uint256 buyerAward;
    uint256 sellerGrossAward;
    uint256 feePaid;
    uint64 dueAt;
    uint64 submittedAt;
    uint64 reviewEndsAt;
    uint64 disputedAt;
    MilestoneState state;
    bytes32 submissionHash;
    bytes32 disputeHash;
    bytes32 resolutionHash;
}

struct ResolutionProposal {
    uint256 buyerAmount;
    uint256 sellerAmount;
    address proposer;
    bytes32 termsHash;
}

mapping(bytes32 id => Agreement) private _agreements;
mapping(bytes32 id => Milestone[]) private _milestones;
mapping(bytes32 id => mapping(uint256 index => ResolutionProposal)) private _resolutionProposals;
mapping(bytes32 id => bool) public usedAgreementIds;
mapping(address account => uint256 amount) public withdrawable;
mapping(address account => bool enabled) public pausers;

address public admin;
address public pendingAdmin;
address payable public platformFeeRecipient;
uint16 public platformFeeBps;
uint256 public totalLockedEscrow;
uint256 public totalWithdrawable;
bool public paused;

uint256 private _reentrancyState = 1;

error Unauthorized();
error InvalidAddress();
error InvalidValue();
error InvalidState();
error InvalidDeadline();
error InvalidMilestone();
error AgreementIdAlreadyUsed();
error ContractPaused();
error ContractNotPaused();
error ReentrantCall();
error NothingToWithdraw();
error TransferFailed();
error DirectPaymentRejected();
error AccountingInvariantViolated();

event AdminTransferStarted(address indexed currentAdmin, address indexed pendingAdmin);
event AdminTransferred(address indexed previousAdmin, address indexed newAdmin);
event PauserUpdated(address indexed account, bool enabled);
event PlatformFeeUpdated(uint16 previousFeeBps, uint16 newFeeBps);
event PlatformFeeRecipientUpdated(
    address indexed previousRecipient,
    address indexed newRecipient
```

```
);
event Paused(address indexed by);
event Unpaused(address indexed by);

event AgreementCreated(
    bytes32 indexed id,
    address indexed buyer,
    address indexed seller,
    address arbitrator,
    uint256 totalValue,
    uint16 feeBps,
    address feeRecipient,
    uint64 acceptanceDeadline,
    uint64 reviewPeriod,
    uint16 milestoneCount
);
event AgreementAccepted(bytes32 indexed id, address indexed seller);
event UnacceptedAgreementCancelled(
    bytes32 indexed id,
    address indexed buyer,
    uint256 refund
);
event AgreementCompleted(
    bytes32 indexed id,
    uint256 buyerRefunded,
    uint256 sellerGrossReleased,
    uint256 feePaid
);

event MilestoneConfigured(
    bytes32 indexed id,
    uint256 indexed index,
    uint256 amount,
    uint64 dueAt
);
event MilestoneSubmitted(
    bytes32 indexed id,
    uint256 indexed index,
    bytes32 submissionHash,
    uint64 reviewEndsAt
);
event DisputeOpened(
    bytes32 indexed id,
    uint256 indexed index,
    address indexed opener,
    bytes32 disputeHash
);
event ResolutionProposed(
    bytes32 indexed id,
    uint256 indexed index,
    address indexed proposer,
    uint256 buyerAmount,
    uint256 sellerAmount,
    bytes32 termsHash
);
event MilestoneSettled(
    bytes32 indexed id,
    uint256 indexed index,
    MilestoneState finalState,
```

```
    uint256 buyerAmount,
    uint256 sellerGrossAmount,
    uint256 feeAmount,
    bytes32 resolutionHash
);

event WithdrawalCredited(address indexed account, uint256 amount);
event Withdrawn(
    address indexed account,
    address indexed recipient,
    uint256 amount
);
event SurplusRecovered(address indexed recipient, uint256 amount);

modifier onlyAdmin() {
    if (msg.sender != admin) revert Unauthorized();
    _;
}

modifier onlyPauser() {
    if (msg.sender != admin && !pausers[msg.sender]) revert Unauthorized();
    _;
}

modifier whenNotPaused() {
    if (paused) revert ContractPaused();
    _;
}

modifier nonReentrant() {
    if (_reentrancyState != 1) revert ReentrantCall();
    _reentrancyState = 2;
    _;
    _reentrancyState = 1;
}

constructor(
    address initialAdmin,
    address payable initialFeeRecipient,
    uint16 initialFeeBps
) {
    if (initialAdmin == address(0) || initialFeeRecipient == address(0)) {
        revert InvalidAddress();
    }
    if (initialFeeBps > MAX_PLATFORM_FEE_BPS) revert InvalidValue();

    admin = initialAdmin;
    pausers[initialAdmin] = true;
    platformFeeRecipient = initialFeeRecipient;
    platformFeeBps = initialFeeBps;

    emit AdminTransferred(address(0), initialAdmin);
    emit PauserUpdated(initialAdmin, true);
    emit PlatformFeeRecipientUpdated(address(0), initialFeeRecipient);
    emit PlatformFeeUpdated(0, initialFeeBps);
}

/// @notice Creates and fully funds an agreement. The seller must accept it.
/// @param id A nonzero, globally unique application-generated identifier.
```

```
/// @param seller The service provider receiving approved milestone proceeds.
/// @param arbitrator The address exclusively authorized to resolve disputes.
/// @param acceptanceDeadline Latest timestamp at which the seller may accept.
/// @param reviewPeriod Time allowed to dispute a submitted milestone.
/// @param amounts Gross milestone amounts in sun.
/// @param dueAts Strictly increasing delivery deadlines in Unix seconds.
function createAgreement(
    bytes32 id,
    address payable seller,
    address arbitrator,
    uint64 acceptanceDeadline,
    uint64 reviewPeriod,
    uint256[] calldata amounts,
    uint64[] calldata dueAts
) external payable whenNotPaused {
    _validateParticipants(id, seller, arbitrator);
    _validateTiming(acceptanceDeadline, reviewPeriod);
    (uint256 total, uint16 milestoneCount) = _validateMilestones(
        acceptanceDeadline,
        amounts,
        dueAts
    );

    if (total == 0 || total != msg.value) revert InvalidValue();

    usedAgreementIds[id] = true;
    Agreement storage agreement = _agreements[id];
    agreement.buyer = payable(msg.sender);
    agreement.seller = seller;
    agreement.arbitrator = arbitrator;
    agreement.feeRecipient = platformFeeRecipient;
    agreement.totalValue = total;
    agreement.createdAt = uint64(block.timestamp);
    agreement.acceptanceDeadline = acceptanceDeadline;
    agreement.reviewPeriod = reviewPeriod;
    agreement.feeBps = platformFeeBps;
    agreement.milestoneCount = milestoneCount;
    agreement.state = AgreementState.PendingAcceptance;

    _storeMilestones(id, amounts, dueAts);

    totalLockedEscrow += total;
    _assertSolvent();
    _emitAgreementCreated(id, agreement);
}

/// @notice Accepts an agreement. Only its named seller may call this.
function acceptAgreement(bytes32 id) external whenNotPaused {
    Agreement storage agreement = _agreement(id);
    if (
        agreement.state != AgreementState.PendingAcceptance ||
        msg.sender != agreement.seller
    ) {
        revert Unauthorized();
    }
    if (block.timestamp > agreement.acceptanceDeadline) {
        revert InvalidDeadline();
    }
```

```
    agreement.state = AgreementState.Active;
    emit AgreementAccepted(id, msg.sender);
}

/// @notice Refunds an agreement that the seller has not accepted.
/// @dev The buyer may cancel before or after the acceptance deadline.
function cancelUnacceptedAgreement(bytes32 id) external {
    Agreement storage agreement = _agreement(id);
    if (
        agreement.state != AgreementState.PendingAcceptance ||
        msg.sender != agreement.buyer
    ) {
        revert Unauthorized();
    }

    agreement.state = AgreementState.Cancelled;
    agreement.buyerRefunded = agreement.totalValue;
    agreement.settledMilestones = agreement.milestoneCount;
    totalLockedEscrow -= agreement.totalValue;
    _credit(agreement.buyer, agreement.totalValue);
    _assertSolvent();

    emit UnacceptedAgreementCancelled(
        id,
        agreement.buyer,
        agreement.totalValue
    );
}

/// @notice Records a seller's off-chain deliverable commitment.
function submitMilestone(
    bytes32 id,
    uint256 index,
    bytes32 submissionHash
) external whenNotPaused {
    Agreement storage agreement = _activeAgreement(id);
    if (msg.sender != agreement.seller) revert Unauthorized();
    if (submissionHash == bytes32(0)) revert InvalidValue();

    Milestone storage milestone = _milestone(id, index);
    if (milestone.state != MilestoneState.Funded) revert InvalidState();
    if (block.timestamp > milestone.dueAt) revert InvalidDeadline();

    uint64 submittedAt = uint64(block.timestamp);
    uint64 reviewEndsAt = submittedAt + agreement.reviewPeriod;
    milestone.state = MilestoneState.Submitted;
    milestone.submittedAt = submittedAt;
    milestone.reviewEndsAt = reviewEndsAt;
    milestone.submissionHash = submissionHash;

    emit MilestoneSubmitted(id, index, submissionHash, reviewEndsAt);
}

/// @notice Releases a submitted milestone early at the buyer's approval.
function releaseMilestone(bytes32 id, uint256 index) external {
    Agreement storage agreement = _activeAgreement(id);
    if (msg.sender != agreement.buyer) revert Unauthorized();

    Milestone storage milestone = _milestone(id, index);
```

```
    if (milestone.state != MilestoneState.Submitted) revert InvalidState();

    _settleMilestone(
        id,
        index,
        0,
        milestone.amount,
        MilestoneState.Released,
        bytes32(0)
    );
}

/// @notice Allows the seller to release an undisputed submission after review.
function claimMilestoneAfterReview(bytes32 id, uint256 index) external {
    Agreement storage agreement = _activeAgreement(id);
    if (msg.sender != agreement.seller) revert Unauthorized();

    Milestone storage milestone = _milestone(id, index);
    if (milestone.state != MilestoneState.Submitted) revert InvalidState();
    if (block.timestamp <= milestone.reviewEndsAt) {
        revert InvalidDeadline();
    }

    _settleMilestone(
        id,
        index,
        0,
        milestone.amount,
        MilestoneState.Released,
        bytes32(0)
    );
}

/// @notice Refunds an unsubmitted milestone after its delivery deadline.
function refundOverdueMilestone(bytes32 id, uint256 index) external {
    Agreement storage agreement = _activeAgreement(id);
    if (msg.sender != agreement.buyer) revert Unauthorized();

    Milestone storage milestone = _milestone(id, index);
    if (milestone.state != MilestoneState.Funded) revert InvalidState();
    if (block.timestamp <= milestone.dueAt) revert InvalidDeadline();

    _settleMilestone(
        id,
        index,
        milestone.amount,
        0,
        MilestoneState.Refunded,
        bytes32(0)
    );
}

/// @notice Opens a dispute during a submitted milestone's review period.
/// @dev Either party may escalate. The evidence itself remains off-chain.
function openDispute(
    bytes32 id,
    uint256 index,
    bytes32 disputeHash
) external {
```

```
    Agreement storage agreement = _activeAgreement(id);
    if (msg.sender != agreement.buyer && msg.sender != agreement.seller) {
        revert Unauthorized();
    }
    if (disputeHash == bytes32(0)) revert InvalidValue();

    Milestone storage milestone = _milestone(id, index);
    if (milestone.state != MilestoneState.Submitted) revert InvalidState();
    if (block.timestamp > milestone.reviewEndsAt) {
        revert InvalidDeadline();
    }

    milestone.state = MilestoneState.Disputed;
    milestone.disputedAt = uint64(block.timestamp);
    milestone.disputeHash = disputeHash;

    emit DisputeOpened(id, index, msg.sender, disputeHash);
}

/// @notice Resolves a dispute. Only the agreement's named arbitrator may call.
function resolveDispute(
    bytes32 id,
    uint256 index,
    uint256 buyerAmount,
    uint256 sellerAmount,
    bytes32 resolutionHash
) external {
    Agreement storage agreement = _activeAgreement(id);
    if (msg.sender != agreement.arbitrator) revert Unauthorized();
    if (resolutionHash == bytes32(0)) revert InvalidValue();

    Milestone storage milestone = _milestone(id, index);
    if (milestone.state != MilestoneState.Disputed) revert InvalidState();
    if (
        buyerAmount > milestone.amount ||
        sellerAmount != milestone.amount - buyerAmount
    ) {
        revert InvalidValue();
    }

    _settleMilestone(
        id,
        index,
        buyerAmount,
        sellerAmount,
        MilestoneState.Resolved,
        resolutionHash
    );
}

/// @notice Proposes a party-agreed split if arbitration is unnecessary or unavailable.
function proposeMutualResolution(
    bytes32 id,
    uint256 index,
    uint256 buyerAmount,
    uint256 sellerAmount,
    bytes32 termsHash
) external {
    Agreement storage agreement = _activeAgreement(id);
```

```
    if (msg.sender != agreement.buyer && msg.sender != agreement.seller) {
        revert Unauthorized();
    }
    if (termsHash == bytes32(0)) revert InvalidValue();

    Milestone storage milestone = _milestone(id, index);
    if (milestone.state != MilestoneState.Disputed) revert InvalidState();
    if (
        buyerAmount > milestone.amount ||
        sellerAmount != milestone.amount - buyerAmount
    ) {
        revert InvalidValue();
    }

    _resolutionProposals[id][index] = ResolutionProposal({
        buyerAmount: buyerAmount,
        sellerAmount: sellerAmount,
        proposer: msg.sender,
        termsHash: termsHash
    });

    emit ResolutionProposed(
        id,
        index,
        msg.sender,
        buyerAmount,
        sellerAmount,
        termsHash
    );
}

/// @notice Accepts the counterparty's exact on-chain resolution proposal.
function acceptMutualResolution(bytes32 id, uint256 index) external {
    Agreement storage agreement = _activeAgreement(id);
    if (msg.sender != agreement.buyer && msg.sender != agreement.seller) {
        revert Unauthorized();
    }

    Milestone storage milestone = _milestone(id, index);
    if (milestone.state != MilestoneState.Disputed) revert InvalidState();

    ResolutionProposal memory proposal = _resolutionProposals[id][index];
    if (
        proposal.proposer == address(0) ||
        proposal.proposer == msg.sender
    ) {
        revert Unauthorized();
    }

    _settleMilestone(
        id,
        index,
        proposal.buyerAmount,
        proposal.sellerAmount,
        MilestoneState.Resolved,
        proposal.termsHash
    );
}
```

```
/// @notice Withdraws the caller's full credit to the caller.
function withdraw() external nonReentrant {
    _withdraw(msg.sender, payable(msg.sender));
}

/// @notice Withdraws the caller's full credit to an alternative recipient.
function withdrawTo(address payable recipient) external nonReentrant {
    if (recipient == address(0)) revert InvalidAddress();
    _withdraw(msg.sender, recipient);
}

function transferAdmin(address newAdmin) external onlyAdmin {
    if (newAdmin == address(0) || newAdmin == admin) {
        revert InvalidAddress();
    }
    pendingAdmin = newAdmin;
    emit AdminTransferStarted(admin, newAdmin);
}

function acceptAdmin() external {
    if (msg.sender != pendingAdmin) revert Unauthorized();
    address previousAdmin = admin;
    admin = msg.sender;
    pendingAdmin = address(0);
    emit AdminTransferred(previousAdmin, msg.sender);
}

function setPauser(address account, bool enabled) external onlyAdmin {
    if (account == address(0)) revert InvalidAddress();
    pausers[account] = enabled;
    emit PauserUpdated(account, enabled);
}

/// @notice Sets the fee for future agreements only.
function setPlatformFee(uint16 newFeeBps) external onlyAdmin {
    if (newFeeBps > MAX_PLATFORM_FEE_BPS) revert InvalidValue();
    uint16 previousFeeBps = platformFeeBps;
    platformFeeBps = newFeeBps;
    emit PlatformFeeUpdated(previousFeeBps, newFeeBps);
}

/// @notice Sets the fee recipient for future agreements only.
function setPlatformFeeRecipient(
    address payable newRecipient
) external onlyAdmin {
    if (newRecipient == address(0)) revert InvalidAddress();
    address previousRecipient = platformFeeRecipient;
    platformFeeRecipient = newRecipient;
    emit PlatformFeeRecipientUpdated(previousRecipient, newRecipient);
}

function pause() external onlyPauser {
    if (paused) revert ContractPaused();
    paused = true;
    emit Paused(msg.sender);
}

function unpause() external onlyAdmin {
    if (!paused) revert ContractNotPaused();
```

```
    paused = false;
    emit Unpaused(msg.sender);
}

/// @notice Recovers only TRX not represented by escrow or withdrawal credits.
/// @dev This function cannot make the contract insolvent.
function recoverSurplus(
    uint256 amount,
    address payable recipient
) external onlyAdmin nonReentrant {
    if (!paused) revert ContractNotPaused();
    if (recipient == address(0)) revert InvalidAddress();
    if (amount == 0 || amount > surplusBalance()) revert InvalidValue();

    (bool ok, ) = recipient.call{value: amount}("");
    if (!ok) revert TransferFailed();
    _assertSolvent();
    emit SurplusRecovered(recipient, amount);
}

function getAgreement(
    bytes32 id
) external view returns (Agreement memory) {
    return _agreement(id);
}

function getMilestone(
    bytes32 id,
    uint256 index
) external view returns (Milestone memory) {
    _agreement(id);
    return _milestone(id, index);
}

function getResolutionProposal(
    bytes32 id,
    uint256 index
) external view returns (ResolutionProposal memory) {
    _agreement(id);
    _milestone(id, index);
    return _resolutionProposals[id][index];
}

function accountedBalance() public view returns (uint256) {
    return totalLockedEscrow + totalWithdrawable;
}

function surplusBalance() public view returns (uint256) {
    uint256 balance = address(this).balance;
    uint256 accounted = accountedBalance();
    return balance > accounted ? balance - accounted : 0;
}

function isSolvent() external view returns (bool) {
    return address(this).balance >= accountedBalance();
}

function _withdraw(address account, address payable recipient) private {
    uint256 amount = withdrawable[account];
```

```
    if (amount == 0) revert NothingToWithdraw();

    withdrawable[account] = 0;
    totalWithdrawable -= amount;

    (bool ok, ) = recipient.call{value: amount}("");
    if (!ok) revert TransferFailed();

    _assertSolvent();
    emit Withdrawn(account, recipient, amount);
}

function _validateParticipants(
    bytes32 id,
    address seller,
    address arbitrator
) private view {
    if (
        id == bytes32(0) ||
        seller == address(0) ||
        arbitrator == address(0)
    ) {
        revert InvalidAddress();
    }
    if (
        seller == msg.sender ||
        arbitrator == msg.sender ||
        arbitrator == seller
    ) {
        revert InvalidAddress();
    }
    if (usedAgreementIds[id]) revert AgreementIdAlreadyUsed();
}

function _validateTiming(
    uint64 acceptanceDeadline,
    uint64 reviewPeriod
) private view {
    if (
        reviewPeriod < MIN_REVIEW_PERIOD ||
        reviewPeriod > MAX_REVIEW_PERIOD
    ) {
        revert InvalidValue();
    }
    if (
        acceptanceDeadline <= block.timestamp ||
        acceptanceDeadline >
        block.timestamp + MAX_ACCEPTANCE_PERIOD
    ) {
        revert InvalidDeadline();
    }
}

function _validateMilestones(
    uint64 acceptanceDeadline,
    uint256[] calldata amounts,
    uint64[] calldata dueAts
) private view returns (uint256 total, uint16 count) {
    uint256 milestoneCount = amounts.length;
```

```
    if (
        milestoneCount == 0 ||
        milestoneCount > MAX_MILESTONES ||
        milestoneCount != dueAts.length
    ) {
        revert InvalidMilestone();
    }

    uint64 previousDueAt;
    uint256 latestAllowedDueAt =
        block.timestamp +
        MAX_MILESTONE_HORIZON;

    for (uint256 i; i < milestoneCount; ++i) {
        uint256 amount = amounts[i];
        uint64 dueAt = dueAts[i];
        if (
            amount == 0 ||
            dueAt <= acceptanceDeadline ||
            dueAt <= previousDueAt ||
            dueAt > latestAllowedDueAt
        ) {
            revert InvalidMilestone();
        }
        total += amount;
        previousDueAt = dueAt;
    }

    count = uint16(milestoneCount);
}

function _storeMilestones(
    bytes32 id,
    uint256[] calldata amounts,
    uint64[] calldata dueAts
) private {
    for (uint256 i; i < amounts.length; ++i) {
        _milestones[id].push(
            Milestone({
                amount: amounts[i],
                buyerAward: 0,
                sellerGrossAward: 0,
                feePaid: 0,
                dueAt: dueAts[i],
                submittedAt: 0,
                reviewEndsAt: 0,
                disputedAt: 0,
                state: MilestoneState.Funded,
                submissionHash: bytes32(0),
                disputeHash: bytes32(0),
                resolutionHash: bytes32(0)
            })
        );
        emit MilestoneConfigured(id, i, amounts[i], dueAts[i]);
    }
}

function _emitAgreementCreated(
    bytes32 id,
```

```
    Agreement storage agreement
) private {
    emit AgreementCreated(
        id,
        agreement.buyer,
        agreement.seller,
        agreement.arbitrator,
        agreement.totalValue,
        agreement.feeBps,
        agreement.feeRecipient,
        agreement.acceptanceDeadline,
        agreement.reviewPeriod,
        agreement.milestoneCount
    );
}

function _settleMilestone(
    bytes32 id,
    uint256 index,
    uint256 buyerAmount,
    uint256 sellerGrossAmount,
    MilestoneState finalState,
    bytes32 resolutionHash
) private {
    Agreement storage agreement = _activeAgreement(id);
    Milestone storage milestone = _milestone(id, index);

    if (
        finalState != MilestoneState.Released &&
        finalState != MilestoneState.Refunded &&
        finalState != MilestoneState.Resolved
    ) {
        revert InvalidState();
    }
    if (
        buyerAmount > milestone.amount ||
        sellerGrossAmount != milestone.amount - buyerAmount
    ) {
        revert InvalidValue();
    }

    uint256 feeAmount =
        (sellerGrossAmount * agreement.feeBps) /
        BPS;
    uint256 sellerNetAmount = sellerGrossAmount - feeAmount;

    milestone.state = finalState;
    milestone.buyerAward = buyerAmount;
    milestone.sellerGrossAward = sellerGrossAmount;
    milestone.feePaid = feeAmount;
    milestone.resolutionHash = resolutionHash;

    delete _resolutionProposals[id][index];

    agreement.buyerRefunded += buyerAmount;
    agreement.sellerGrossReleased += sellerGrossAmount;
    agreement.feePaid += feeAmount;
    agreement.settledMilestones += 1;
```

```
        totalLockedEscrow -= milestone.amount;
        _credit(agreement.buyer, buyerAmount);
        _credit(agreement.seller, sellerNetAmount);
        _credit(agreement.feeRecipient, feeAmount);

        emit MilestoneSettled(
            id,
            index,
            finalState,
            buyerAmount,
            sellerGrossAmount,
            feeAmount,
            resolutionHash
        );

        if (agreement.settledMilestones == agreement.milestoneCount) {
            agreement.state = AgreementState.Completed;
            emit AgreementCompleted(
                id,
                agreement.buyerRefunded,
                agreement.sellerGrossReleased,
                agreement.feePaid
            );
        }

        _assertSolvent();
    }

    function _credit(address account, uint256 amount) private {
        if (amount == 0) return;
        withdrawable[account] += amount;
        totalWithdrawable += amount;
        emit WithdrawalCredited(account, amount);
    }

    function _agreement(
        bytes32 id
    ) private view returns (Agreement storage agreement) {
        agreement = _agreements[id];
        if (agreement.state == AgreementState.None) revert InvalidState();
    }

    function _activeAgreement(
        bytes32 id
    ) private view returns (Agreement storage agreement) {
        agreement = _agreement(id);
        if (agreement.state != AgreementState.Active) revert InvalidState();
    }

    function _milestone(
        bytes32 id,
        uint256 index
    ) private view returns (Milestone storage milestone) {
        if (index >= _milestones[id].length) revert InvalidMilestone();
        milestone = _milestones[id][index];
    }

    function _assertSolvent() private view {
        if (address(this).balance < accountedBalance()) {
```

```
            revert AccountingInvariantViolated();
        }
    }

    receive() external payable {
        revert DirectPaymentRejected();
    }

    fallback() external payable {
        revert DirectPaymentRejected();
    }
}
```