Solana Smart Contracts: Waste Management DePIN

Contract Architecture Overview

Core Contracts

```
WasteManagement Program
├── waste_token.rs          # WASTE SPL token management
├── node_registry.rs        # Hardware node registration & validation
├── user_rewards.rs         # Community participation rewards
├── municipal_contracts.rs  # Government partnership management
├── data_verification.rs    # Sensor data validation & storage
├── governance.rs           # DAO voting and proposals
└── staking.rs             # Token staking and yield farming
```

Program Architecture

```
// Program ID (hypothetical)
declare_id!("WasteDP1N7xK8mQ2vR5nF3jL9wE6tY4uI8oP7aS6dF5gH4jK3");

// Main program entry point
#[program]
pub mod waste_management {
    use super::*;

    // Initialize the waste management program
    pub fn initialize_program(ctx: Context<InitializeProgram>) -> Result<()>

    // Register a new waste collection node
    pub fn register_node(ctx: Context<RegisterNode>, node_data: NodeData) -> Result<()>

    // Submit sensor data from nodes
    pub fn submit_sensor_data(ctx: Context<SubmitData>, data: SensorReading) -> Result<()>

    // Reward users for proper waste disposal
    pub fn reward_user_action(ctx: Context<RewardUser>, action: WasteAction) -> Result<()>

    // Stake WASTE tokens for governance
    pub fn stake_tokens(ctx: Context<StakeTokens>, amount: u64) -> Result<()>

    // Create governance proposal
    pub fn create_proposal(ctx: Context<CreateProposal>, proposal: ProposalData) -> Result<()>
}
```

1. WASTE Token Contract

Token Specifications

```rust
// waste_token.rs
use anchor_lang::prelude::*;
use anchor_spl::token::{self, Token, TokenAccount, Mint};

#[derive(Accounts)]
pub struct InitializeToken<'info> {
    #[account(
        init,
        payer = authority,
        mint::decimals = 6,
        mint::authority = authority,
    )]
    pub waste_token_mint: Account<'info, Mint>,

    #[account(mut)]
    pub authority: Signer<'info>,

    pub system_program: Program<'info, System>,
    pub token_program: Program<'info, Token>,
    pub rent: Sysvar<'info, Rent>,
}

// Token distribution structure
#[account]
pub struct TokenDistribution {
    pub total_supply: u64,           // 1,000,000,000 WASTE (with 6 decimals)
    pub community_rewards: u64,      // 40% - 400M tokens
    pub development_fund: u64,       // 20% - 200M tokens
    pub ecosystem_growth: u64,       // 15% - 150M tokens
    pub team_advisors: u64,          // 10% - 100M tokens
    pub treasury_reserve: u64,       // 10% - 100M tokens
    pub liquidity_exchange: u64,     // 5% - 50M tokens
    pub distributed_amount: u64,     // Track distributed tokens
    pub burn_amount: u64,            // Track burned tokens
}

// Mint tokens for rewards
pub fn mint_reward_tokens(
    ctx: Context<MintRewardTokens>,
    recipient: Pubkey,
    amount: u64,
) -> Result<()> {
    let distribution = &mut ctx.accounts.token_distribution;

    require!(
        distribution.distributed_amount + amount <=
distribution.community_rewards,
        WasteError::ExceedsRewardAllocation
    );

    // Mint tokens to recipient
    token::mint_to(
        CpiContext::new(
            ctx.accounts.token_program.to_account_info(),
            token::MintTo {
                mint: ctx.accounts.waste_token_mint.to_account_info(),
                to: ctx.accounts.recipient_token_account.to_account_info(),
                authority: ctx.accounts.mint_authority.to_account_info(),
            },
        ),
        amount,
    )?;

    distribution.distributed_amount += amount;
    Ok(())
}
```

2. Node Registry Contract

Hardware Node Management

```
// node_registry.rs
#[account]
pub struct WasteNode {
    pub node_id: String,             // Unique identifier
    pub node_type: NodeType,         // Basic or Advanced
    pub location: Location,          // GPS coordinates
    pub owner: Pubkey,               // Node operator wallet
    pub status: NodeStatus,          // Active, Inactive, Maintenance
    pub registration_time: i64,      // Unix timestamp
    pub last_heartbeat: i64,         // Last communication
    pub total_uptime: u64,           // Cumulative uptime seconds
    pub data_quality_score: u8,      // 0-100 quality rating
    pub rewards_earned: u64,         // Total WASTE tokens earned
    pub stake_amount: u64,           // Staked tokens for operation
}

#[derive(AnchorSerialize, AnchorDeserialize, Clone, PartialEq)]
pub enum NodeType {
    BasicFillMonitoring,    // Type A - $50 nodes
    AdvancedAI,            // Type B - $200 nodes
    VehicleTracker,        // Collection vehicle GPS
}

#[derive(AnchorSerialize, AnchorDeserialize, Clone)]
pub struct Location {
    pub latitude: f64,
    pub longitude: f64,
    pub address: String,
    pub municipality: String,
}

// Register new waste collection node
pub fn register_node(
    ctx: Context<RegisterNode>,
    node_id: String,
    node_type: NodeType,
    location: Location,
    stake_amount: u64,
) -> Result<()> {
    let node = &mut ctx.accounts.waste_node;
    let clock = Clock::get()?;

    // Validate minimum stake requirement
    let min_stake = match node_type {
        NodeType::BasicFillMonitoring => 1000 * 10_u64.pow(6), // 1000 WASTE
        NodeType::AdvancedAI => 5000 * 10_u64.pow(6),          // 5000 WASTE
        NodeType::VehicleTracker => 500 * 10_u64.pow(6),       // 500 WASTE
    };

    require!(stake_amount >= min_stake, WasteError::InsufficientStake);

    // Initialize node data
    node.node_id = node_id;
    node.node_type = node_type;
    node.location = location;
    node.owner = ctx.accounts.owner.key();
    node.status = NodeStatus::Active;
    node.registration_time = clock.unix_timestamp;
    node.last_heartbeat = clock.unix_timestamp;
    node.stake_amount = stake_amount;

    // Transfer stake tokens to escrow
    token::transfer(
        CpiContext::new(
            ctx.accounts.token_program.to_account_info(),
            token::Transfer {
                from: ctx.accounts.owner_token_account.to_account_info(),
                to: ctx.accounts.stake_escrow.to_account_info(),
                authority: ctx.accounts.owner.to_account_info(),
            },
        ),
        stake_amount,
    )?;

    Ok(())
}
```

3. User Rewards Contract

Community Participation Rewards

```
// user_rewards.rs
#[account]
pub struct UserProfile {
    pub wallet: Pubkey,
    pub registration_date: i64,
    pub total_disposals: u64,
    pub proper_sorting_count: u64,
    pub contamination_reports: u64,
    pub community_actions: u64,
    pub total_rewards_earned: u64,
    pub current_streak: u32,        // Days of consistent participation
    pub longest_streak: u32,
    pub reputation_score: u16,      // 0-1000 reputation points
    pub household_id: Option<String>, // Link to household group
}

#[derive(AnchorSerialize, AnchorDeserialize, Clone)]
pub struct WasteAction {
    pub action_type: ActionType,
    pub weight_kg: Option<f32>,     // Weight of waste disposed
    pub bin_location: String,       // Node ID where action occurred
    pub photo_hash: Option<String>, // IPFS hash of verification photo
    pub timestamp: i64,
    pub verification_score: u8,     // AI confidence 0-100
}

#[derive(AnchorSerialize, AnchorDeserialize, Clone, PartialEq)]
pub enum ActionType {
    GeneralWasteDisposal,
    RecyclingDisposal,
    OrganicWasteDisposal,
    HazardousDisposal,
    ContaminationReport,
    BinStatusUpdate,
    CommunityEducation,
    WasteAudit,
}

// Calculate reward amount based on action
pub fn calculate_reward_amount(
    action: &WasteAction,
    user_profile: &UserProfile,
    node_data: &WasteNode,
) -> u64 {
    let base_reward = match action.action_type {
        ActionType::GeneralWasteDisposal => {
            let weight = action.weight_kg.unwrap_or(1.0);
            (weight * 0.1 * 10_u64.pow(6) as f32) as u64 // 0.1 WASTE per kg
        },
        ActionType::RecyclingDisposal => {
            let weight = action.weight_kg.unwrap_or(1.0);
            (weight * 0.5 * 10_u64.pow(6) as f32) as u64 // 0.5 WASTE per kg
        },
        ActionType::OrganicWasteDisposal => {
            let weight = action.weight_kg.unwrap_or(1.0);
            (weight * 0.3 * 10_u64.pow(6) as f32) as u64 // 0.3 WASTE per kg
        },
        ActionType::HazardousDisposal => 2 * 10_u64.pow(6), // 2 WASTE per item
        ActionType::ContaminationReport => {
            if action.verification_score > 80 {
                20 * 10_u64.pow(6) // 20 WASTE for verified report
            } else {
                10 * 10_u64.pow(6) // 10 WASTE for unverified
            }
        },
        ActionType::BinStatusUpdate => 2 * 10_u64.pow(6), // 2 WASTE per update
        ActionType::CommunityEducation => 25 * 10_u64.pow(6), // 25 WASTE per session
        ActionType::WasteAudit => 75 * 10_u64.pow(6), // 75 WASTE per audit
    };

    // Apply multipliers
    let mut final_reward = base_reward;

    // Streak bonus (up to 2x)
    if user_profile.current_streak > 7 {
        final_reward = (final_reward as f64 * 1.5) as u64;
    }
    if user_profile.current_streak > 30 {
        final_reward = (final_reward as f64 * 2.0) as u64;
    }

    // Reputation bonus (up to 1.5x)
    let reputation_multiplier = 1.0 + (user_profile.reputation_score as f64 / 2000.0);
    final_reward = (final_reward as f64 * reputation_multiplier) as u64;

    // Quality bonus for high verification scores
    if action.verification_score > 95 {
        final_reward = (final_reward as f64 * 1.5) as u64;
    }

    final_reward
}

// Process user waste disposal action
pub fn reward_user_action(
    ctx: Context<RewardUserAction>,
    action: WasteAction,
) -> Result<()> {
    let user_profile = &mut ctx.accounts.user_profile;
    let node = &ctx.accounts.waste_node;

    // Validate action timestamp (within last 24 hours)
    let clock = Clock::get()?;
    require!(
        clock.unix_timestamp - action.timestamp < 86400,
        WasteError::ActionTooOld
    );

    // Calculate reward amount
    let reward_amount = calculate_reward_amount(&action, user_profile, node);

    // Update user profile
    user_profile.total_disposals += 1;
    if action.verification_score > 90 {
        user_profile.proper_sorting_count += 1;
    }
    user_profile.total_rewards_earned += reward_amount;

    // Update streak
    let days_since_last = (clock.unix_timestamp - user_profile.registration_date)
/ 86400;
    if days_since_last == 1 {
        user_profile.current_streak += 1;
        if user_profile.current_streak > user_profile.longest_streak {
            user_profile.longest_streak = user_profile.current_streak;
        }
    } else if days_since_last > 1 {
        user_profile.current_streak = 1;
    }

    // Mint reward tokens
    mint_reward_tokens(
        CpiContext::new(
            ctx.accounts.token_program.to_account_info(),
            MintRewardTokens {
                waste_token_mint: ctx.accounts.waste_token_mint.to_account_info(),
                recipient_token_account:
ctx.accounts.user_token_account.to_account_info(),
                mint_authority: ctx.accounts.mint_authority.to_account_info(),
                token_distribution:
ctx.accounts.token_distribution.to_account_info(),
                token_program: ctx.accounts.token_program.to_account_info(),
            },
        ),
        user_profile.wallet,
        reward_amount,
    )?;

    Ok(())
}
```

4. Data Verification Contract

Sensor Data Validation

```
// data_verification.rs
#[account]
pub struct SensorReading {
    pub node_id: String,
    pub timestamp: i64,
    pub reading_type: ReadingType,
    pub value: f64,
    pub unit: String,
    pub confidence_score: u8,       // AI confidence 0-100
    pub validation_status: ValidationStatus,
    pub validator_nodes: Vec<Pubkey>, // Nodes that validated this reading
    pub merkle_proof: Option<String>, // Proof for batch verification
}

#[derive(AnchorSerialize, AnchorDeserialize, Clone, PartialEq)]
pub enum ReadingType {
    FillLevel,          // Percentage full (0-100)
    Weight,             // Kilograms
    WasteClassification, // AI-detected waste type
    Temperature,        // Celsius
    Contamination,      // Boolean contamination detected
    BinStatus,          // Operational status
}

#[derive(AnchorSerialize, AnchorDeserialize, Clone, PartialEq)]
pub enum ValidationStatus {
    Pending,
    Validated,
    Rejected,
    Disputed,
}

// Submit sensor data with validation
pub fn submit_sensor_data(
    ctx: Context<SubmitSensorData>,
    readings: Vec<SensorReading>,
) -> Result<()> {
    let node = &mut ctx.accounts.waste_node;
    let clock = Clock::get()?;

    // Update node heartbeat
    node.last_heartbeat = clock.unix_timestamp;

    // Validate readings
    for reading in readings.iter() {
        // Check timestamp is recent (within 1 hour)
        require!(
            clock.unix_timestamp - reading.timestamp < 3600,
            WasteError::StaleData
        );

        // Validate reading ranges
        match reading.reading_type {
            ReadingType::FillLevel => {
                require!(
                    reading.value >= 0.0 && reading.value <= 100.0,
                    WasteError::InvalidFillLevel
                );
            },
            ReadingType::Weight => {
                require!(
                    reading.value >= 0.0 && reading.value <= 1000.0, // Max 1000kg
                    WasteError::InvalidWeight
                );
            },
            ReadingType::Temperature => {
                require!(
                    reading.value >= -40.0 && reading.value <= 80.0,
                    WasteError::InvalidTemperature
                );
            },
            _ => {} // Other validations as needed
        }
    }

    // Calculate data quality score
    let avg_confidence: u8 = readings.iter()
        .map(|r| r.confidence_score)
        .sum::<u8>() / readings.len() as u8;

    // Update node data quality
    node.data_quality_score = (node.data_quality_score + avg_confidence) / 2;

    // Reward node operator for data submission
    let data_reward = calculate_data_reward(&readings, node);
    if data_reward > 0 {
        // Mint tokens to node operator
        mint_reward_tokens(
            CpiContext::new(
                ctx.accounts.token_program.to_account_info(),
                MintRewardTokens {
                    waste_token_mint: ctx.accounts.waste_token_mint.to_account_info(),
                    recipient_token_account: ctx.accounts.node_operator_token_account.to_account_info(),
                    mint_authority: ctx.accounts.mint_authority.to_account_info(),
                    token_distribution: ctx.accounts.token_distribution.to_account_info(),
                    token_program: ctx.accounts.token_program.to_account_info(),
                },
            ),
            node.owner,
            data_reward,
        )?;

        node.rewards_earned += data_reward;
    }

    Ok(())
}

// Calculate reward for data submission
fn calculate_data_reward(readings: &[SensorReading], node: &WasteNode) -> u64 {
    let base_reward = match node.node_type {
        NodeType::BasicFillMonitoring => 15 * 10_u64.pow(6), // 15 WASTE per day
        NodeType::AdvancedAI => 35 * 10_u64.pow(6),          // 35 WASTE per day
        NodeType::VehicleTracker => 25 * 10_u64.pow(6),      // 25 WASTE per day
    };

    // Quality multiplier based on data accuracy
    let quality_multiplier = if node.data_quality_score > 95 {
        1.4
    } else if node.data_quality_score > 90 {
        1.2
    } else if node.data_quality_score > 80 {
        1.0
    } else {
        0.8
    };

    // Uptime multiplier
    let uptime_percentage = node.total_uptime as f64 / (86400.0 * 30.0); // 30 days
    let uptime_multiplier = if uptime_percentage > 0.99 {
        1.5
    } else if uptime_percentage > 0.95 {
        1.2
    } else {
        1.0
    };

    (base_reward as f64 * quality_multiplier * uptime_multiplier) as u64
}
```

5. Governance Contract

DAO Voting System

```
// governance.rs
#[account]
pub struct Proposal {
    pub id: u64,
    pub title: String,
    pub description: String,
    pub proposer: Pubkey,
    pub proposal_type: ProposalType,
    pub voting_start: i64,
    pub voting_end: i64,
    pub votes_for: u64,
    pub votes_against: u64,
    pub votes_abstain: u64,
    pub status: ProposalStatus,
    pub execution_data: Option<Vec<u8>>, // Serialized instruction data
    pub minimum_threshold: u64,          // Minimum votes needed
}

#[derive(AnchorSerialize, AnchorDeserialize, Clone, PartialEq)]
pub enum ProposalType {
    TokenEconomics,      // Change reward rates, burn mechanisms
    FeatureUpgrade,      // New functionality proposals
    PartnershipApproval, // Municipal contract approvals
    TreasuryManagement,  // Fund allocation decisions
    ProtocolUpgrade,     // Smart contract upgrades
}

#[derive(AnchorSerialize, AnchorDeserialize, Clone, PartialEq)]
pub enum ProposalStatus {
    Active,
    Passed,
    Rejected,
    Executed,
    Expired,
}

#[account]
pub struct Vote {
    pub proposal_id: u64,
    pub voter: Pubkey,
    pub vote_choice: VoteChoice,
    pub voting_power: u64,
    pub timestamp: i64,
}

#[derive(AnchorSerialize, AnchorDeserialize, Clone, PartialEq)]
pub enum VoteChoice {
    For,
    Against,
    Abstain,
}

// Create new governance proposal
pub fn create_proposal(
    ctx: Context<CreateProposal>,
    title: String,
    description: String,
    proposal_type: ProposalType,
    voting_duration: i64, // Duration in seconds
) -> Result<()> {
    let proposal = &mut ctx.accounts.proposal;
    let clock = Clock::get()?;
    // Validate proposer has minimum stake (10,000 WASTE)
    let min_proposal_stake = 10000 * 10_u64.pow(6);
    require!(
        ctx.accounts.proposer_stake.amount >= min_proposal_stake,
        WasteError::InsufficientProposalStake
    );

    // Initialize proposal
    proposal.id = ctx.accounts.governance_state.next_proposal_id;
    proposal.title = title;
    proposal.description = description;
    proposal.proposer = ctx.accounts.proposer.key();
    proposal.proposal_type = proposal_type;
    proposal.voting_start = clock.unix_timestamp;
    proposal.voting_end = clock.unix_timestamp + voting_duration;
    proposal.status = ProposalStatus::Active;

    // Set minimum threshold based on proposal type
    proposal.minimum_threshold = match proposal.proposal_type {
        ProposalType::TokenEconomics => ctx.accounts.governance_state.total_staked / 4, // 25%
        ProposalType::ProtocolUpgrade => ctx.accounts.governance_state.total_staked / 3, // 33%
        _ => ctx.accounts.governance_state.total_staked / 10, // 10%
    };

    // Increment proposal counter
    ctx.accounts.governance_state.next_proposal_id += 1;

    Ok(())
}

// Cast vote on proposal
pub fn cast_vote(
    ctx: Context<CastVote>,
    proposal_id: u64,
    vote_choice: VoteChoice,
) -> Result<()> {
    let proposal = &mut ctx.accounts.proposal;
    let vote = &mut ctx.accounts.vote;
    let clock = Clock::get()?;

    // Validate voting period
    require!(
        clock.unix_timestamp >= proposal.voting_start &&
        clock.unix_timestamp <= proposal.voting_end,
        WasteError::VotingPeriodClosed
    );

    // Calculate voting power (staked tokens + node operator bonus)
    let base_voting_power = ctx.accounts.voter_stake.amount;
    let node_bonus = if ctx.accounts.voter_node.is_some() {
        base_voting_power / 2 // 50% bonus for node operators
    } else {
        0
    };
    let total_voting_power = base_voting_power + node_bonus;

    // Record vote
    vote.proposal_id = proposal_id;
    vote.voter = ctx.accounts.voter.key();
    vote.vote_choice = vote_choice.clone();
    vote.voting_power = total_voting_power;
    vote.timestamp = clock.unix_timestamp;

    // Update proposal vote counts
    match vote_choice {
        VoteChoice::For => proposal.votes_for += total_voting_power,
        VoteChoice::Against => proposal.votes_against += total_voting_power,
        VoteChoice::Abstain => proposal.votes_abstain += total_voting_power,
    }

    Ok(())
}
```

6. Error Definitions

Custom Error Types

```
// errors.rs
#[error_code]
pub enum WasteError {
    #[msg("Exceeds reward token allocation")]
    ExceedsRewardAllocation,

    #[msg("Insufficient stake amount for node type")]
    InsufficientStake,

    #[msg("Action timestamp too old")]
    ActionTooOld,

    #[msg("Stale sensor data")]
    StaleData,

    #[msg("Invalid fill level reading")]
    InvalidFillLevel,

    #[msg("Invalid weight reading")]
    InvalidWeight,

    #[msg("Invalid temperature reading")]
    InvalidTemperature,

    #[msg("Insufficient stake for proposal creation")]
    InsufficientProposalStake,
    #[msg("Voting period is closed")]
    VotingPeriodClosed,

    #[msg("Node not found or inactive")]
    NodeNotFound,

    #[msg("Unauthorized action")]
    Unauthorized,

    #[msg("Invalid verification score")]
    InvalidVerificationScore,
}
```

Deployment Configuration

Program Deployment

```
# Build the program
anchor build

# Deploy to devnet
solana config set --url https://api.devnet.solana.com
anchor deploy --provider.cluster devnet

# Initialize program state
anchor run initialize-program

# Verify deployment
solana program show WasteDP1N7xK8mQ2vR5nF3jL9wE6tY4uI8oP7aS6dF5gH4jK3
```

Integration Points

```
// Client SDK integration example
import { WasteManagementProgram } from '@waste-depin/solana-sdk';

const program = new WasteManagementProgram(connection, wallet);

// Register a new node
await program.registerNode({
  nodeId: "NYC_001_BASIC",
  nodeType: "BasicFillMonitoring",
  location: { lat: 40.7128, lng: -74.0060 },
  stakeAmount: 1000 * 10**6 // 1000 WASTE tokens
});

// Submit sensor data
await program.submitSensorData([{
  nodeId: "NYC_001_BASIC",
  readingType: "FillLevel",
  value: 75.5,
  timestamp: Date.now() / 1000,
  confidenceScore: 95
}]);

// Reward user action
await program.rewardUserAction({
  actionType: "RecyclingDisposal",
  weightKg: 2.5,
  binLocation: "NYC_001_BASIC",
  verificationScore: 92
});
```

Smart contracts enabling decentralized waste management through blockchain incentives and community participation.