/** *Submitted for verification at Etherscan.io on 2020-07-17 */ pragma solidity ^0.4.24; // File: contracts/upgradable/ProxyStorage.sol contract ProxyStorage {   /**   * Current contract to which we are proxing   */   address public currentContract;   address public proxyOwner; } // File: contracts/upgradable/OwnableStorage.sol contract OwnableStorage {   address public owner;   constructor() internal {   owner = msg.sender;   } } // File: erc821/contracts/AssetRegistryStorage.sol contract AssetRegistryStorage {   string internal _name;   string internal _symbol;   string internal _description;   /**   * Stores the total count of assets managed by this registry   */   uint256 internal _count;   /**   * Stores an array of assets owned by a given account   */   mapping(address => uint256[]) internal _assetsOf;   /**   * Stores the current holder of an asset   */   mapping(uint256 => address) internal _holderOf;   /**   * Stores the index of an asset in the `_assetsOf` array of its holder   */   mapping(uint256 => uint256) internal _indexOfAsset;   /**   * Stores the data associated with an asset   */   mapping(uint256 => string) internal _assetData;   /**   * For a given account, for a given operator, store whether that operator is   * allowed to transfer and modify assets on behalf of them.   */   mapping(address => mapping(address => bool)) internal _operators;   /**   * Approval array   */   mapping(uint256 => address) internal _approval; } // File: contracts/estate/IEstateRegistry.sol contract IEstateRegistry {   function mint(address to, string metadata) external returns (uint256);   function ownerOf(uint256 _tokenId) public view returns (address _owner); // from ERC721   // Events   event CreateEstate(address indexed _owner,   uint256 indexed _estateId,   string _data   );   event AddLand(uint256 indexed _estateId,   uint256 indexed _landId   );   event RemoveLand(uint256 indexed _estateId,   uint256 indexed _landId,   address indexed _destinatary   );   event Update(uint256 indexed _assetId,   address indexed _holder,   address indexed _operator,   string _data   );   event UpdateOperator(uint256 indexed _estateId,   address indexed _operator   );   event UpdateManager(address indexed _owner,   address indexed _operator,   address indexed _caller,   bool _approved   );   event SetLANDRegistry(address indexed _registry   );   event SetEstateLandBalanceToken(address indexed _previousEstateLandBalance,   address indexed _newEstateLandBalance   ); } // File: contracts/minimeToken/IMinimeToken.sol interface IMiniMeToken { ////////////////// // Generate and destroy tokens //////////////////   /// @notice Generates `_amount` tokens that are assigned to `_owner`   /// @param _owner The address that will be assigned the new tokens   /// @param _amount The quantity of tokens generated   /// @return True if the tokens are generated correctly   function generateTokens(address _owner, uint _amount) external returns (bool);   /// @notice Burns `_amount` tokens from `_owner`   /// @param _owner The address that will lose the tokens   /// @param _amount The quantity of tokens to burn   /// @return True if the tokens are burned correctly   function destroyTokens(address _owner, uint _amount) external returns (bool);   /// @param _owner The address that's balance is being requested   /// @return The balance of `_owner` at the current block   function balanceOf(address _owner) external view returns (uint256 balance);   event Transfer(address indexed _from, address indexed _to, uint256 _amount); } // File: contracts/land/LANDStorage.sol contract LANDStorage {   mapping (address => uint) public latestPing;   uint256 constant clearLow = 0xffffffffffffffffffffffffffffffff00000000000000000000000000000000;   uint256 constant clearHigh = 0x00000000000000000000000000000000ffffffffffffffffffffffffffffffff;   uint256 constant factor = 0x100000000000000000000000000000000;   mapping (address => bool) internal

_deprecated_authorizedDeploy; mapping (uint256 => address) public updateOperator; IEstateRegistry public estateRegistry; mapping (address => bool) public authorizedDeploy; mapping(address => mapping(address => bool)) public updateManager; // Land balance minime token IMiniMeToken public landBalance; // Registered balance accounts mapping(address => bool) public registeredBalance; } // File: contracts/Storage.sol contract Storage is ProxyStorage, OwnableStorage, AssetRegistryStorage, LANDStorage { } // File: contracts/upgradable/Ownable.sol contract Ownable is Storage { event OwnerUpdate(address _prevOwner, address _newOwner); modifier onlyOwner { assert(msg.sender == owner); _; } function transferOwnership(address _newOwner) public onlyOwner { require(_newOwner != owner, "Cannot transfer to yourself"); owner = _newOwner; } } // File: contracts/upgradable/IApplication.sol contract IApplication { function initialize(bytes data) public; } // File: openzeppelin-solidity/contracts/math/SafeMath.sol /** * @title SafeMath * @dev Math operations with safety checks that throw on error */ library SafeMath { /** * @dev Multiplies two numbers, throws on overflow. */ function mul(uint256 _a, uint256 _b) internal pure returns (uint256 c) { // Gas optimization: this is cheaper than asserting 'a' not being zero, but the // benefit is lost if 'b' is also tested. // See: https://github.com/OpenZeppelin/openzeppelin-solidity/pull/522 if (_a == 0) { return 0; } c = _a * _b; assert(c / _a == _b); return c; } /** * @dev Integer division of two numbers, truncating the quotient. */ function div(uint256 _a, uint256 _b) internal pure returns (uint256) { // assert(_b > 0); // Solidity automatically throws when dividing by 0 // uint256 c = _a / _b; // assert(_a == _b * c + _a % _b); // There is no case in which this doesn't hold return _a / _b; } /** * @dev Subtracts two numbers, throws on overflow (i.e. if subtrahend is greater than minuend). */ function sub(uint256 _a, uint256 _b) internal pure returns (uint256) { assert(_b <= _a); return _a - _b; } /** * @dev Adds two numbers, throws on overflow. */ function add(uint256 _a, uint256 _b) internal pure returns (uint256 c) { c = _a + _b; assert(c >= _a); return c; } } // File: erc821/contracts/IERC721Base.sol interface IERC721Base { function totalSupply() external view returns (uint256); // function exists(uint256 assetId) external view returns (bool); function ownerOf(uint256 assetId) external view returns (address); function balanceOf(address holder) external view returns (uint256); function safeTransferFrom(address from, address to, uint256 assetId) external; function safeTransferFrom(address from, address to, uint256 assetId, bytes userData) external; function transferFrom(address from, address to, uint256 assetId) external; function approve(address operator, uint256 assetId) external; function setApprovalForAll(address operator, bool authorized) external; function getApprovedAddress(uint256 assetId) external view returns (address); function isApprovedForAll(address assetHolder, address operator) external view returns (bool); function isAuthorized(address operator, uint256 assetId) external view returns (bool); /** * @dev Deprecated transfer event. Now we use the standard with three parameters * It is only used in the ABI to get old transfer events. Do not remove */ event Transfer(address indexed from, address indexed to, uint256 indexed assetId, address operator, bytes userData, bytes operatorData); /** * @dev Deprecated transfer event. Now we use the standard with three parameters * It is only used in the ABI to get old transfer events. Do not remove */ event Transfer(address indexed from, address indexed to, uint256 indexed assetId, address operator, bytes userData); event Transfer(address indexed from, address indexed to, uint256 indexed assetId); event ApprovalForAll(address indexed

holder, address indexed operator, bool authorized); event Approval(address indexed owner, address indexed operator, uint256 indexed assetId); } // File: erc821/contracts/IERC721Receiver.sol interface IERC721Receiver { function onERC721Received(address _operator, address _from, uint256 _tokenId, bytes _userData) external returns (bytes4); } // File: erc821/contracts/ERC165.sol interface ERC165 { function supportsInterface(bytes4 interfaceID) external view returns (bool); } // File: erc821/contracts/ERC721Base.sol contract ERC721Base is AssetRegistryStorage, IERC721Base, ERC165 { using SafeMath for uint256; // Equals to `bytes4(keccak256("onERC721Received(address,address,uint256,bytes)"))` bytes4 private constant ERC721_RECEIVED = 0x150b7a02; bytes4 private constant InterfaceId_ERC165 = 0x01ffc9a7; /* * 0x01ffc9a7 === * bytes4(keccak256('supportsInterface(bytes4)')) */ bytes4 private constant Old_InterfaceId_ERC721 = 0x7c0633c6; bytes4 private constant InterfaceId_ERC721 = 0x80ac58cd; /* * 0x80ac58cd === * bytes4(keccak256('balanceOf(address)')) ^ * bytes4(keccak256('ownerOf(uint256)')) ^ * bytes4(keccak256('approve(address,uint256)')) ^ * bytes4(keccak256('getApproved(uint256)')) ^ * bytes4(keccak256('setApprovalForAll(address,bool)')) ^ * bytes4(keccak256('isApprovedForAll(address,address)')) ^ * bytes4(keccak256('transferFrom(address,address,uint256)')) ^ * bytes4(keccak256('safeTransferFrom(address,address,uint256)')) ^ * bytes4(keccak256('safeTransferFrom(address,address,uint256,bytes')) */ // // Global Getters // /** * @dev Gets the total amount of assets stored by the contract * @return uint256 representing the total amount of assets */ function totalSupply() external view returns (uint256) { return _totalSupply(); } function _totalSupply() internal view returns (uint256) { return _count; } // // Asset-centric getter functions // /** * @dev Queries what address owns an asset. This method does not throw. * In order to check if the asset exists, use the `exists` function or check if the * return value of this call is `0`. * @return uint256 the assetId */ function ownerOf(uint256 assetId) external view returns (address) { return _ownerOf(assetId); } function _ownerOf(uint256 assetId) internal view returns (address) { return _holderOf[assetId]; } // // Holder-centric getter functions // /** * @dev Gets the balance of the specified address * @param owner address to query the balance of * @return uint256 representing the amount owned by the passed address */ function balanceOf(address owner) external view returns (uint256) { return _balanceOf(owner); } function _balanceOf(address owner) internal view returns (uint256) { return _assetsOf[owner].length; } // // Authorization getters // /** * @dev Query whether an address has been authorized to move any assets on behalf of someone else * @param operator the address that might be authorized * @param assetHolder the address that provided the authorization * @return bool true if the operator has been authorized to move any assets */ function isApprovedForAll(address assetHolder, address operator) external view returns (bool) { return _isApprovedForAll(assetHolder, operator); } function _isApprovedForAll(address assetHolder, address operator) internal view returns (bool) { return _operators[assetHolder][operator]; } /** * @dev Query what address has been particularly authorized to move an asset * @param assetId the asset to be queried for * @return bool true if the asset has been approved by the holder */ function

```
getApproved(uint256 assetId) external view returns (address) {     return _getApprovedAddress(assetId);   }   function getApprovedAddress(uint256 assetId) external view returns (address) {     return _getApprovedAddress(assetId);   }   function _getApprovedAddress(uint256 assetId) internal view returns (address) {     return _approval[assetId];   }   /**    * @dev Query if an operator can move an asset.    * @param operator the address that might be authorized    * @param assetId the asset that has been `approved` for transfer    * @return bool true if the asset has been approved by the holder    */   function isAuthorized(address operator, uint256 assetId) external view returns (bool) {     return _isAuthorized(operator, assetId);   }   function _isAuthorized(address operator, uint256 assetId) internal view returns (bool)   {     require(operator != 0);     address owner = _ownerOf(assetId);     if (operator == owner) {       return true;     }     return _isApprovedForAll(owner, operator) || _getApprovedAddress(assetId) == operator;   }   //   // Authorization   //   /**    * @dev Authorize a third party operator to manage (send) msg.sender's asset    * @param operator address to be approved    * @param authorized bool set to true to authorize, false to withdraw authorization    */   function setApprovalForAll(address operator, bool authorized) external {     return _setApprovalForAll(operator, authorized);   }   function _setApprovalForAll(address operator, bool authorized) internal {     if (authorized) {       require(!_isApprovedForAll(msg.sender, operator));       _addAuthorization(operator, msg.sender);     } else {       require(_isApprovedForAll(msg.sender, operator));       _clearAuthorization(operator, msg.sender);     }     emit ApprovalForAll(msg.sender, operator, authorized);   }   /**    * @dev Authorize a third party operator to manage one particular asset    * @param operator address to be approved    * @param assetId asset to approve    */   function approve(address operator, uint256 assetId) external {     address holder = _ownerOf(assetId);     require(msg.sender == holder || _isApprovedForAll(msg.sender, holder));     require(operator != holder);     if (_getApprovedAddress(assetId) != operator) {       _approval[assetId] = operator;       emit Approval(holder, operator, assetId);     }   }   function _addAuthorization(address operator, address holder) private {     _operators[holder][operator] = true;   }   function _clearAuthorization(address operator, address holder) private {     _operators[holder][operator] = false;   }   //   // Internal Operations   //   function _addAssetTo(address to, uint256 assetId) internal {     _holderOf[assetId] = to;     uint256 length = _balanceOf(to);     _assetsOf[to].push(assetId);     _indexOfAsset[assetId] = length;     _count = _count.add(1);   }   function _removeAssetFrom(address from, uint256 assetId) internal {     uint256 assetIndex = _indexOfAsset[assetId];     uint256 lastAssetIndex = _balanceOf(from).sub(1);     uint256 lastAssetId = _assetsOf[from][lastAssetIndex];     _holderOf[assetId] = 0;     // Insert the last asset into the position previously occupied by the asset to be removed     _assetsOf[from][assetIndex] = lastAssetId;     // Resize the array     _assetsOf[from][lastAssetIndex] = 0;     _assetsOf[from].length--;     // Remove the array if no more assets are owned to prevent pollution     if (_assetsOf[from].length == 0) {       delete _assetsOf[from];     }     // Update the index of positions for the asset     _indexOfAsset[assetId] = 0;     _indexOfAsset[lastAssetId] = assetIndex;     _count = _count.sub(1);   }   function _clearApproval(address holder, uint256 assetId) internal {     if (_ownerOf(assetId) == holder && _approval[assetId] != 0) {       _approval[assetId] = 0;       emit Approval(holder, 0, assetId);     }   }   //   // Supply-altering functions   //   function _generate(uint256 assetId, address beneficiary) internal {     require(_holderOf[assetId] == 0);     _addAssetTo(beneficiary, assetId);     emit
```

Transfer(0, beneficiary, assetId); } function _destroy(uint256 assetId) internal { address holder = _holderOf[assetId]; require(holder != 0); _removeAssetFrom(holder, assetId); emit Transfer(holder, 0, assetId); } // // Transaction related operations // modifier onlyHolder(uint256 assetId) { require(_ownerOf(assetId) == msg.sender); _; } modifier onlyAuthorized(uint256 assetId) { require(_isAuthorized(msg.sender, assetId)); _; } modifier isCurrentOwner(address from, uint256 assetId) { require(_ownerOf(assetId) == from); _; } modifier isDestinataryDefined(address destinatary) { require(destinatary != 0); _; } modifier destinataryIsNotHolder(uint256 assetId, address to) { require(_ownerOf(assetId) != to); _; } /** * @dev Alias of `safeTransferFrom(from, to, assetId, '')` * * @param from address that currently owns an asset * @param to address to receive the ownership of the asset * @param assetId uint256 ID of the asset to be transferred */ function safeTransferFrom(address from, address to, uint256 assetId) external { return _doTransferFrom(from, to, assetId, '', true); } /** * @dev Securely transfers the ownership of a given asset from one address to * another address, calling the method `onNFTReceived` on the target address if * there's code associated with it * * @param from address that currently owns an asset * @param to address to receive the ownership of the asset * @param assetId uint256 ID of the asset to be transferred * @param userData bytes arbitrary user information to attach to this transfer */ function safeTransferFrom(address from, address to, uint256 assetId, bytes userData) external { return _doTransferFrom(from, to, assetId, userData, true); } /** * @dev Transfers the ownership of a given asset from one address to another address * Warning! This function does not attempt to verify that the target address can send * tokens. * * @param from address sending the asset * @param to address to receive the ownership of the asset * @param assetId uint256 ID of the asset to be transferred */ function transferFrom(address from, address to, uint256 assetId) external { return _doTransferFrom(from, to, assetId, '', false); } function _doTransferFrom(address from, address to, uint256 assetId, bytes userData, bool doCheck) onlyAuthorized(assetId) internal { _moveToken(from, to, assetId, userData, doCheck); } function _moveToken(address from, address to, uint256 assetId, bytes userData, bool doCheck) isDestinataryDefined(to) destinataryIsNotHolder(assetId, to) isCurrentOwner(from, assetId) private { address holder = _holderOf[assetId]; _clearApproval(holder, assetId); _removeAssetFrom(holder, assetId); _addAssetTo(to, assetId); emit Transfer(holder, to, assetId); if (doCheck && _isContract(to)) { // Equals to `bytes4(keccak256("onERC721Received(address,address,uint256,bytes)"))` require(IERC721Receiver(to).onERC721Received(msg.sender, holder, assetId, userData) == ERC721_RECEIVED); } } /** * Internal function that moves an asset from one holder to another */ /** * @dev Returns `true` if the contract implements `interfaceID` and `interfaceID` is not 0xffffffff, `false` otherwise * @param _interfaceID The interface identifier, as specified in ERC-165 */ function supportsInterface(bytes4 _interfaceID) external view returns (bool) { if (_interfaceID == 0xffffffff) { return false; } return _interfaceID == InterfaceId_ERC165 || _interfaceID == Old_InterfaceId_ERC721 || _interfaceID == InterfaceId_ERC721; } // // Utilities // function _isContract(address addr) internal view returns (bool) { uint size; assembly { size := extcodesize(addr) } return size > 0; } } // File: erc821/contracts/IERC721Enumerable.sol contract IERC721Enumerable { /** * @notice Enumerate active tokens * @dev Throws if `index` >= `totalSupply()`, otherwise

SHALL NOT throw. * @param index A counter less than `totalSupply()` * @return The identifier for the `index`th asset, (sort order not * specified) */ // TODO (eordano): Not implemented // function tokenByIndex(uint256 index) public view returns (uint256 _assetId); /** * @notice Count of owners which own at least one asset * Must not throw. * @return A count of the number of owners which own asset */ // TODO (eordano): Not implemented // function countOfOwners() public view returns (uint256 _count); /** * @notice Enumerate owners * @dev Throws if `index` >= `countOfOwners()`, otherwise must not throw. * @param index A counter less than `countOfOwners()` * @return The address of the `index`th owner (sort order not specified) */ // TODO (eordano): Not implemented // function ownerByIndex(uint256 index) public view returns (address owner); /** * @notice Get all tokens of a given address * @dev This is not intended to be used on-chain * @param owner address of the owner to query * @return a list of all assetIds of a user */ function tokensOf(address owner) external view returns (uint256[]); /** * @notice Enumerate tokens assigned to an owner * @dev Throws if `index` >= `balanceOf(owner)` or if * `owner` is the zero address, representing invalid assets. * Otherwise this must not throw. * @param owner An address where we are interested in assets owned by them * @param index A counter less than `balanceOf(owner)` * @return The identifier for the `index`th asset assigned to `owner`, * (sort order not specified) */ function tokenOfOwnerByIndex(address owner, uint256 index) external view returns (uint256 tokenId); } // File: erc821/contracts/ERC721Enumerable.sol contract ERC721Enumerable is AssetRegistryStorage, IERC721Enumerable { /** * @notice Get all tokens of a given address * @dev This is not intended to be used on-chain * @param owner address of the owner to query * @return a list of all assetIds of a user */ function tokensOf(address owner) external view returns (uint256[]) { return _assetsOf[owner]; } /** * @notice Enumerate tokens assigned to an owner * @dev Throws if `index` >= `balanceOf(owner)` or if * `owner` is the zero address, representing invalid assets. * Otherwise this must not throw. * @param owner An address where we are interested in assets owned by them * @param index A counter less than `balanceOf(owner)` * @return The identifier for the `index`th asset assigned to `owner`, * (sort order not specified) */ function tokenOfOwnerByIndex(address owner, uint256 index) external view returns (uint256 assetId) { require(index < _assetsOf[owner].length); require(index < (1<<127)); return _assetsOf[owner][index]; } } // File: erc821/contracts/IERC721Metadata.sol contract IERC721Metadata { /** * @notice A descriptive name for a collection of NFTs in this contract */ function name() external view returns (string); /** * @notice An abbreviated name for NFTs in this contract */ function symbol() external view returns (string); /** * @notice A description of what this DAR is used for */ function description() external view returns (string); /** * Stores arbitrary info about a token */ function tokenMetadata(uint256 assetId) external view returns (string); } // File: erc821/contracts/ERC721Metadata.sol contract ERC721Metadata is AssetRegistryStorage, IERC721Metadata { function name() external view returns (string) { return _name; } function symbol() external view returns (string) { return _symbol; } function description() external view returns (string) { return _description; } function tokenMetadata(uint256 assetId) external view returns (string) { return _assetData[assetId]; } function _update(uint256 assetId, string data) internal { _assetData[assetId] = data; } } // File: erc821/contracts/FullAssetRegistry.sol contract FullAssetRegistry is ERC721Base,

ERC721Enumerable, ERC721Metadata { constructor() public { } /** * @dev Method to check if an asset identified by the given id exists under this DAR. * @return uint256 the assetId */ function exists(uint256 assetId) external view returns (bool) { return _exists(assetId); } function _exists(uint256 assetId) internal view returns (bool) { return _holderOf[assetId] != 0; } function decimals() external pure returns (uint256) { return 0; } } // File: contracts/land/ILANDRegistry.sol interface ILANDRegistry { // LAND can be assigned by the owner function assignNewParcel(int x, int y, address beneficiary) external; function assignMultipleParcels(int[] x, int[] y, address beneficiary) external; // After one year, LAND can be claimed from an inactive public key function ping() external; // LAND-centric getters function encodeTokenId(int x, int y) external pure returns (uint256); function decodeTokenId(uint value) external pure returns (int, int); function exists(int x, int y) external view returns (bool); function ownerOfLand(int x, int y) external view returns (address); function ownerOfLandMany(int[] x, int[] y) external view returns (address[]); function landOf(address owner) external view returns (int[], int[]); function landData(int x, int y) external view returns (string); // Transfer LAND function transferLand(int x, int y, address to) external; function transferManyLand(int[] x, int[] y, address to) external; // Update LAND function updateLandData(int x, int y, string data) external; function updateManyLandData(int[] x, int[] y, string data) external; // Authorize an updateManager to manage parcel data function setUpdateManager(address _owner, address _operator, bool _approved) external; // Events event Update(uint256 indexed assetId, address indexed holder, address indexed operator, string data); event UpdateOperator(uint256 indexed assetId, address indexed operator); event UpdateManager(address indexed _owner, address indexed _operator, address indexed _caller, bool _approved); event DeployAuthorized(address indexed _caller, address indexed _deployer); event DeployForbidden(address indexed _caller, address indexed _deployer); event SetLandBalanceToken(address indexed _previousLandBalance, address indexed _newLandBalance); } // File: contracts/metadata/IMetadataHolder.sol contract IMetadataHolder is ERC165 { function getMetadata(uint256 /* assetId */) external view returns (string); } // File: contracts/land/LANDRegistry.sol /* solium-disable function-order */ contract LANDRegistry is Storage, Ownable, FullAssetRegistry, ILANDRegistry { bytes4 constant public GET_METADATA = bytes4(keccak256("getMetadata(uint256)")); function initialize(bytes) external { _name = "Decentraland LAND"; _symbol = "LAND"; _description = "Contract that stores the Decentraland LAND registry"; } modifier onlyProxyOwner() { require(msg.sender == proxyOwner, "This function can only be called by the proxy owner"); _; } modifier onlyDeployer() { require(msg.sender == proxyOwner || authorizedDeploy[msg.sender], "This function can only be called by an authorized deployer"); _; } modifier onlyOwnerOf(uint256 assetId) { require(msg.sender == _ownerOf(assetId), "This function can only be called by the owner of the asset"); _; } modifier onlyUpdateAuthorized(uint256 tokenId) { require(msg.sender == _ownerOf(tokenId) || _isAuthorized(msg.sender, tokenId) || _isUpdateAuthorized(msg.sender, tokenId), "msg.sender is not authorized to update"); _; } modifier canSetUpdateOperator(uint256 tokenId) { address owner = _ownerOf(tokenId); require(_isAuthorized(msg.sender, tokenId) || updateManager[owner][msg.sender], "unauthorized user"); _; } // // Authorization

```
//   function isUpdateAuthorized(address operator, uint256 assetId) external view returns (bool) {     return _isUpdateAuthorized(operator, assetId);   }   function _isUpdateAuthorized(address operator, uint256 assetId) internal view returns (bool) {     address owner = _ownerOf(assetId);     return owner == operator  ||       updateOperator[assetId] == operator ||       updateManager[owner][operator];   }   function authorizeDeploy(address beneficiary) external onlyProxyOwner {     require(beneficiary != address(0), "invalid address");     require(authorizedDeploy[beneficiary] == false, "address is already authorized");     authorizedDeploy[beneficiary] = true;     emit DeployAuthorized(msg.sender, beneficiary);   }   function forbidDeploy(address beneficiary) external onlyProxyOwner {     require(beneficiary != address(0), "invalid address");     require(authorizedDeploy[beneficiary], "address is already forbidden");     authorizedDeploy[beneficiary] = false;     emit DeployForbidden(msg.sender, beneficiary);   }   //   // LAND Create   //   function assignNewParcel(int x, int y, address beneficiary) external onlyDeployer {     _generate(_encodeTokenId(x, y), beneficiary);     _updateLandBalance(address(0), beneficiary);   }   function assignMultipleParcels(int[] x, int[] y, address beneficiary) external onlyDeployer {     for (uint i = 0; i < x.length; i++) {       _generate(_encodeTokenId(x[i], y[i]), beneficiary);       _updateLandBalance(address(0), beneficiary);     }   }   //   // Inactive keys after 1 year lose ownership   //   function ping() external {     // solium-disable-next-line security/no-block-members     latestPing[msg.sender] = block.timestamp;   }   function setLatestToNow(address user) external {     require(msg.sender == proxyOwner || _isApprovedForAll(msg.sender, user), "Unauthorized user");     // solium-disable-next-line security/no-block-members     latestPing[user] = block.timestamp;   }   // // LAND Getters   //   function encodeTokenId(int x, int y) external pure returns (uint) {     return _encodeTokenId(x, y);   }   function _encodeTokenId(int x, int y) internal pure returns (uint result) {     require(       -1000000 < x && x < 1000000 && -1000000 < y && y < 1000000,       "The coordinates should be inside bounds"     );     return _unsafeEncodeTokenId(x, y);   }   function _unsafeEncodeTokenId(int x, int y) internal pure returns (uint) {     return ((uint(x) * factor) & clearLow) | (uint(y) & clearHigh);   }   function decodeTokenId(uint value) external pure returns (int, int) {     return _decodeTokenId(value);   }   function _unsafeDecodeTokenId(uint value) internal pure returns (int x, int y) {     x = expandNegative128BitCast((value & clearLow) >> 128);     y = expandNegative128BitCast(value & clearHigh);   }   function _decodeTokenId(uint value) internal pure returns (int x, int y) {     (x, y) = _unsafeDecodeTokenId(value);     require(       -1000000 < x && x < 1000000 && -1000000 < y && y < 1000000,       "The coordinates should be inside bounds"     );   }   function expandNegative128BitCast(uint value) internal pure returns (int) {     if (value & (1<<127) != 0) {       return int(value | clearLow);     }     return int(value);   }   function exists(int x, int y) external view returns (bool) {     return _exists(x, y);   }   function _exists(int x, int y) internal view returns (bool) {     return _exists(_encodeTokenId(x, y));   }   function ownerOfLand(int x, int y) external view returns (address) {     return _ownerOfLand(x, y);   }   function _ownerOfLand(int x, int y) internal view returns (address) {     return _ownerOf(_encodeTokenId(x, y));   }   function ownerOfLandMany(int[] x, int[] y) external view returns (address[]) {     require(x.length > 0, "You should supply at least one coordinate");     require(x.length == y.length, "The coordinates should have the same length");     address[] memory addrs = new address[](x.length);     for (uint i = 0; i < x.length; i++) {       addrs[i] = _ownerOfLand(x[i], y[i]);     }     return addrs;   }   function landOf(address owner) external view returns (int[], int[]) {     uint256 len =
```

```
_assetsOf[owner].length;     int[] memory x = new int[](len);     int[] memory y = new int[](len);
int assetX;     int assetY;     for (uint i = 0; i < len; i++) {       (assetX, assetY) =
_decodeTokenId(_assetsOf[owner][i]);       x[i] = assetX;       y[i] = assetY;     }      return (x, y);   }
function tokenMetadata(uint256 assetId) external view returns (string) {     return
_tokenMetadata(assetId);   }    function _tokenMetadata(uint256 assetId) internal view returns
(string) {     address _owner = _ownerOf(assetId);     if (_isContract(_owner) && _owner !=
address(estateRegistry)) {       if ((ERC165(_owner)).supportsInterface(GET_METADATA)) {
return IMetadataHolder(_owner).getMetadata(assetId);       }     }     return _assetData[assetId];
}    function landData(int x, int y) external view returns (string) {     return
_tokenMetadata(_encodeTokenId(x, y));   }    //   // LAND Transfer   //    function
transferFrom(address from, address to, uint256 assetId) external {     require(to !=
address(estateRegistry), "EstateRegistry unsafe transfers are not allowed");     return
_doTransferFrom(       from,       to,       assetId,       "",       false     );   }    function
transferLand(int x, int y, address to) external {     uint256 tokenId = _encodeTokenId(x, y);
_doTransferFrom(       _ownerOf(tokenId),       to,       tokenId,       "",       true     );   }    function
transferManyLand(int[] x, int[] y, address to) external {     require(x.length > 0, "You should
supply at least one coordinate");     require(x.length == y.length, "The coordinates should have
the same length");      for (uint i = 0; i < x.length; i++) {       uint256 tokenId =
_encodeTokenId(x[i], y[i]);       _doTransferFrom(       _ownerOf(tokenId),       to,       tokenId,
"",       true       );     }   }    function transferLandToEstate(int x, int y, uint256 estateId) external {
require(       estateRegistry.ownerOf(estateId) == msg.sender,       "You must own the Estate you
want to transfer to"     );      uint256 tokenId = _encodeTokenId(x, y);     _doTransferFrom(
_ownerOf(tokenId),      address(estateRegistry),      tokenId,      toBytes(estateId),      true     );
}    function transferManyLandToEstate(int[] x, int[] y, uint256 estateId) external {
require(x.length > 0, "You should supply at least one coordinate");     require(x.length ==
y.length, "The coordinates should have the same length");     require(
estateRegistry.ownerOf(estateId) == msg.sender,       "You must own the Estate you want to
transfer to"     );      for (uint i = 0; i < x.length; i++) {       uint256 tokenId = _encodeTokenId(x[i],
y[i]);       _doTransferFrom(       _ownerOf(tokenId),       address(estateRegistry),       tokenId,
toBytes(estateId),       true       );     }   }    /**    * @notice Set LAND updateOperator    *
@param assetId - LAND id    * @param operator - address of the account to be set as the
updateOperator    */   function setUpdateOperator(     uint256 assetId,     address operator   )
public     canSetUpdateOperator(assetId)   {     updateOperator[assetId] = operator;     emit
UpdateOperator(assetId, operator);   }    /**    * @notice Set many LAND updateOperator    *
@param _assetIds - LAND ids    * @param _operator - address of the account to be set as the
updateOperator    */   function setManyUpdateOperator(     uint256[] _assetIds,     address
_operator   )     public   {     for (uint i = 0; i < _assetIds.length; i++) {
setUpdateOperator(_assetIds[i], _operator);     }   }    /**   * @dev Set an updateManager for an
account   * @param _owner - address of the account to set the updateManager   * @param
_operator - address of the account to be set as the updateManager   * @param _approved -
bool whether the address will be approved or not   */   function setUpdateManager(address
_owner, address _operator, bool _approved) external {     require(_operator != msg.sender, "The
operator should be different from owner");     require(       _owner == msg.sender ||
_isApprovedForAll(_owner, msg.sender),       "Unauthorized user"     );
```

```
updateManager[_owner][_operator] = _approved;      emit UpdateManager(       _owner,
_operator,       msg.sender,       _approved     );   }    //   // Estate generation   //    event
EstateRegistrySet(address indexed registry);    function setEstateRegistry(address registry)
external onlyProxyOwner {     estateRegistry = IEstateRegistry(registry);     emit
EstateRegistrySet(registry);   }    function createEstate(int[] x, int[] y, address beneficiary)
external returns (uint256) {     // solium-disable-next-line arg-overflow     return _createEstate(x,
y, beneficiary, "");   }    function createEstateWithMetadata(     int[] x,     int[] y,     address
beneficiary,     string metadata    )    external     returns (uint256)   {     // solium-disable-next-line
arg-overflow     return _createEstate(x, y, beneficiary, metadata);   }    function _createEstate(
int[] x,     int[] y,     address beneficiary,     string metadata    )    internal     returns (uint256)   {
require(x.length > 0, "You should supply at least one coordinate");     require(x.length ==
y.length, "The coordinates should have the same length");     require(address(estateRegistry) !=
0, "The Estate registry should be set");      uint256 estateTokenId =
estateRegistry.mint(beneficiary, metadata);     bytes memory estateTokenIdBytes =
toBytes(estateTokenId);      for (uint i = 0; i < x.length; i++) {       uint256 tokenId =
_encodeTokenId(x[i], y[i]);       _doTransferFrom(         _ownerOf(tokenId),
address(estateRegistry),         tokenId,         estateTokenIdBytes,         true       );     }      return
estateTokenId;   }    function toBytes(uint256 x) internal pure returns (bytes b) {     b = new
bytes(32);     // solium-disable-next-line security/no-inline-assembly     assembly { mstore(add(b,
32), x) }   }    //   // LAND Update   //    function updateLandData(     int x,     int y,     string data    )
external   {     return _updateLandData(x, y, data);   }    function _updateLandData(     int x,     int
y,     string data    )    internal     onlyUpdateAuthorized(_encodeTokenId(x, y))   {     uint256
assetId = _encodeTokenId(x, y);     address owner = _holderOf[assetId];      _update(assetId,
data);      emit Update(       assetId,       owner,       msg.sender,       data     );   }    function
updateManyLandData(int[] x, int[] y, string data) external {     require(x.length > 0, "You should
supply at least one coordinate");     require(x.length == y.length, "The coordinates should have
the same length");     for (uint i = 0; i < x.length; i++) {       _updateLandData(x[i], y[i], data);     }
}    /**    * @dev Set a new land balance minime token    * @notice Set new land balance token:
`_newLandBalance`    * @param _newLandBalance address of the new land balance token    */
function setLandBalanceToken(address _newLandBalance) onlyProxyOwner external {
require(_newLandBalance != address(0), "New landBalance should not be zero address");
emit SetLandBalanceToken(landBalance, _newLandBalance);     landBalance =
IMiniMeToken(_newLandBalance);   }    /**    * @dev Register an account balance    * @notice
Register land Balance    */   function registerBalance() external {
require(!registeredBalance[msg.sender], "Register Balance::The user is already registered");
// Get balance of the sender     uint256 currentBalance = landBalance.balanceOf(msg.sender);
if (currentBalance > 0) {       require(         landBalance.destroyTokens(msg.sender,
currentBalance),         "Register Balance::Could not destroy tokens"       );     }      // Set balance
as registered     registeredBalance[msg.sender] = true;      // Get LAND balance     uint256
newBalance = _balanceOf(msg.sender);      // Generate Tokens     require(
landBalance.generateTokens(msg.sender, newBalance),       "Register Balance::Could not
generate tokens"     );   }    /**    * @dev Unregister an account balance    * @notice Unregister
land Balance    */   function unregisterBalance() external {
require(registeredBalance[msg.sender], "Unregister Balance::The user not registered");      // Set
```

```
balance as unregistered     registeredBalance[msg.sender] = false;      // Get balance     uint256
currentBalance = landBalance.balanceOf(msg.sender);      // Destroy Tokens     require(
landBalance.destroyTokens(msg.sender, currentBalance),       "Unregister Balance::Could not
destroy tokens"     );   }    function _doTransferFrom(     address from,     address to,     uint256
assetId,     bytes userData,     bool doCheck    )    internal   {     updateOperator[assetId] =
address(0);     _updateLandBalance(from, to);     super._doTransferFrom(       from,       to,
assetId,       userData,       doCheck     );   }    function _isContract(address addr) internal view
returns (bool) {     uint size;     // solium-disable-next-line security/no-inline-assembly     assembly
{ size := extcodesize(addr) }     return size > 0;   }    /**    * @dev Update account balances    *
@param _from account    * @param _to account    */   function _updateLandBalance(address
_from, address _to) internal {     if (registeredBalance[_from]) {
landBalance.destroyTokens(_from, 1);     }      if (registeredBalance[_to]) {
landBalance.generateTokens(_to, 1);     }   } }
```