ERC-721

Any contract that follows theERC-721 standard is an ERC-721 token.

Here is the interface for ERC-721.

interface

ERC721

{ event

Transfer (address

indexed _from ,

address

indexed _to ,

uint256

indexed _tokenId) ; event

Approval (address

indexed _owner ,

address

indexed _approved ,

uint256

indexed _tokenId) ; event

ApprovalForAll (address

indexed _owner ,

address

indexed _operator ,

bool _approved) ;

function

balanceOf (address _owner)

external

view

returns

(uint256) ; function

ownerOf (uint256 _tokenId)

external

view

returns

(address) ; function

safeTransferFrom (address _from ,

address _to ,

uint256 _tokenId ,

bytes data)

external

payable ; function

safeTransferFrom (address _from ,

address _to ,

uint256 _tokenId)

external

payable ; function

transferFrom (address _from ,

address _to ,

uint256 _tokenId)

external

payable ; function

approve (address _approved ,

uint256 _tokenId)

external

payable ; function

setApprovalForAll (address _operator ,

bool _approved)

external ; function

getApproved (uint256 _tokenId)

external

view

returns

(address) ; function

isApprovedForAll (address _owner ,

address _operator)

external

view

returns

(bool) ; } Example implementation of an ERC-721 token contract written in Rust.

src/erc721.rs

note This code has yet to be audited. Please use at your own risk. //! Implementation of the ERC-721 standard //! //! The eponymous [Erc721] type provides all the standard methods, //! and is intended to be inherited by other contract types. //! //! You can configure the behavior of [Erc721] via the [Erc721Params] trait, //! which allows specifying the name, symbol, and token uri. //! //! Note that this code is unaudited and not fit for production use.

use

alloc :: { string :: String , vec ,

vec :: Vec } ; use

```
alloy_primitives :: { Address ,
U256 ,
FixedBytes } ; use
alloy_sol_types :: sol ; use
core :: { borrow :: BorrowMut ,
marker :: PhantomData } ; use
stylus_sdk :: { abi :: Bytes , evm , msg , prelude :: * } ;
pub
trait
Erc721Params
{ /// Immutable NFT name. const
NAME :
& 'static
str ;
/// Immutable NFT symbol. const
SYMBOL :
& 'static
str ;
/// The NFT's Uniform Resource Identifier. fn
token_uri ( token_id :
U256 )
->
String ; }
sol_storage!
{ /// Erc721 implements all ERC-721 methods pub
struct
Erc721 < T :
Erc721Params
{ /// Token id to owner map mapping ( uint256 => address ) owners ; /// User to balance map mapping ( address => uint256 ) balances ; /// Token id to approved user map mapping ( uint256 => address ) token_approvals ; /// User to operator map (the operator can manage all NFTs of the owner) mapping ( address =>
mapping ( address =>
bool ) ) operator_approvals ; /// Total supply uint256 total_supply ; /// Used to allow [Erc721Params] PhantomData < T
        phantom ; } }
// Declare events and Solidity error types sol!
{ event Transfer ( address indexed from , address indexed to , uint256 indexed token_id ) ; event Approval ( address indexed owner , address indexed approved , uint256 indexed token_id ) ; event ApprovalForAll ( address indexed owner , address indexed operator ,
bool approved ) ;
```

// Token id has not been minted, or it has been burned error InvalidTokenId (uint256 token_id) ; // The specified address is not the owner of the specified token id error NotOwner (address from , uint256 token_id , address real_owner) ; // The specified address does not have allowance to spend the specified token id error NotApproved (address owner , address spender , uint256 token_id) ; // Attempt to transfer token id to the Zero address error TransferToZero (uint256 token_id) ; // The receiver address refused to receive the specified token id error ReceiverRefused (address receiver , uint256 token_id , bytes4 returned) ; }

/// Represents the ways methods may fail.

[derive(SolidityError)]

pub

enum

Erc721Error

{ InvalidTokenId (InvalidTokenId) , NotOwner (NotOwner) , NotApproved (NotApproved) , TransferToZero (TransferToZero) , ReceiverRefused (ReceiverRefused) , }

// External interfaces sol_interface!

{ /// Allows calls to the onERC721Received method of other contracts implementing IERC721TokenReceiver. interface IERC721TokenReceiver

{ function onERC721Received (address operator , address from , uint256 token_id , bytes data) external returns (bytes4) ; } }

/// Selector for onERC721Received, which is returned by contracts implementing IERC721TokenReceiver. const

ERC721_TOKEN_RECEIVER_ID :

u32

=

0x150b7a02 ;

// These methods aren't external, but are helpers used by external methods. // Methods marked as "pub" here are usable outside of the erc721 module (i.e. they're callable from lib.rs). impl < T :

Erc721Params

Erc721 < T

{ /// Requires that msg::sender() is authorized to spend a given token fn

require_authorized_to_spend (& self , from :

Address , token_id :

U256)

->

Result < ( ) ,

Erc721Error

{ // from must be the owner of the token_id let owner =

self . owner_of (token_id) ? ; if from != owner { return

Err (Erc721Error :: NotOwner (NotOwner

{ from , token_id , real_owner : owner , })) ; }

// caller is the owner if

msg :: sender ( )

== owner { return

Ok (( )) ; }

// caller is an operator for the owner (can manage their tokens) if

self . operator_approvals . getter (owner) . get (msg :: sender ( ))

{ return

Ok (( )) ; }

// caller is approved to manage this token_id if

msg :: sender ( )

==

self . token_approvals . get (token_id)

{ return

Ok (( )) ; }

// otherwise, caller is not allowed to manage this token_id Err (Erc721Error :: NotApproved (NotApproved

{ owner , spender :

msg :: sender ( ) , token_id , })) }

/// Transfers token_id from from to to. /// This function does check that from is the owner of the token, but it does not check /// that to is not the zero address, as this function is usable for burning. pub

fn

transfer (& mut

self , token_id :

U256 , from :

Address , to :

Address)

->

Result < ( ) ,

Erc721Error

{ let

mut owner =

self . owners . setter (token_id) ; let previous_owner = owner . get ( ) ; if previous_owner != from { return

Err (Erc721Error :: NotOwner (NotOwner

{ from , token_id , real_owner : previous_owner , })) ; } owner . set (to) ;

// right now working with storage can be verbose, but this will change upcoming version of the Stylus SDK let

mut from_balance =

self . balances . setter (from) ; let balance = from_balance . get ( )

-

U256 :: from (1) ; from_balance . set (balance) ;

let

mut to_balance =

self . balances . setter (to) ; let balance = to_balance . get ( )

+

U256 :: from (1) ; to_balance . set (balance) ;

// cleaning app the approved mapping for this token self . token_approvals . delete (token_id) ;

evm :: log (Transfer

{ from , to , token_id }) ; Ok (( )) }

/// Calls onERC721Received on the to address if it is a contract. /// Otherwise it does nothing fn

call_receiver < S :

TopLevelStorage

(storage :

& mut

S , token_id :

U256 , from :

Address , to :

Address , data :

Vec < u8

,)

->

Result < ( ) ,

Erc721Error

{ if to . has_code ( )

{ let receiver =

IERC721TokenReceiver :: new (to) ; let received = receiver . on_erc_721_received (& mut

* storage ,

msg :: sender ( ) , from , token_id , data . into ( )) . map_err (| _e |

Erc721Error :: ReceiverRefused (ReceiverRefused

{ receiver : receiver . address , token_id , returned :

alloy_primitives :: FixedBytes (0_u32 . to_be_bytes ( )) , })) ? . 0 ;

if

u32 :: from_be_bytes (received)

!=

ERC721_TOKEN_RECEIVER_ID

{ return

Err (Erc721Error :: ReceiverRefused (ReceiverRefused

{ receiver : receiver . address , token_id , returned :

alloy_primitives :: FixedBytes (received) , })) ; } } Ok (( )) }

/// Transfers and calls onERC721Received pub

fn

safe_transfer < S :

TopLevelStorage

+

BorrowMut < Self

(storage :

& mut

S , token_id :

U256 , from :

Address , to :

Address , data :

Vec < u8

,)

->

Result < ( ) ,

Erc721Error

{ storage . borrow_mut ( ) . transfer (token_id , from , to) ? ; Self :: call_receiver (storage , token_id , from , to , data) }

/// Mints a new token and transfers it toto pub

fn

mint (& mut

self , to :

Address)

->

Result < ( ) ,

Erc721Error

{ let new_token_id =

self . total_supply . get ( ) ; self . total_supply . set (new_token_id +

U256 :: from (1u8)) ; self . transfer (new_token_id ,

Address :: default ( ) , to) ? ; Ok (( )) }

/// Burns the token token_id from from /// Note that total_supply is not reduced since it's used to calculate the next token_id to mint pub

fn

burn (& mut

self , from :

Address , token_id :

U256)

->

Result < ( ) ,

Erc721Error

{ self . transfer (token_id , from ,

Address :: default ( )) ? ; Ok (( )) } }

// these methods are external to other contracts

[public]

impl < T :

Erc721Params

Erc721 < T

{ /// Immutable NFT name. pub

fn

name ( )

->

Result < String ,

Erc721Error

{ Ok (T :: NAME . into ( )) }

/// Immutable NFT symbol. pub

fn

symbol ( )

->

Result < String ,

Erc721Error

{ Ok (T :: SYMBOL . into ( )) }

/// The NFT's Uniform Resource Identifier.

[selector(name =

"tokenURI")] pub

fn

token_uri (& self , token_id :

U256)

->

Result < String ,

Erc721Error

{ self . owner_of (token_id) ? ;

// require NFT exist Ok (T :: token_uri (token_id)) }

/// Gets the number of NFTs owned by an account. pub

fn

balance_of (& self , owner :

Address)

->

Result < U256 ,

Erc721Error

{ Ok (self . balances . get (owner)) }

/// Gets the owner of the NFT, if it exists. pub

fn

owner_of (& self , token_id :

U256)

->

Result < Address ,

Erc721Error

{ let owner =

self . owners . get (token_id) ; if owner . is_zero ( )

{ return

Err (Erc721Error :: InvalidTokenId (InvalidTokenId

{ token_id })) ; } Ok (owner) }

/// Transfers an NFT, but only after checking theto address can receive the NFT. /// It includes additional data for the receiver.

[selector(name =

"safeTransferFrom")] pub

fn

safe_transfer_from_with_data < S :

TopLevelStorage

+

BorrowMut < Self

(storage :

& mut

S , from :

Address , to :

Address , token_id :

U256 , data :

Bytes ,)

->

Result < ( ) ,

Erc721Error

{ if to . is_zero ( )

{ return

Err (Erc721Error :: TransferToZero (TransferToZero

{ token_id })) ; } storage . borrow_mut ( ) . require_authorized_to_spend (from , token_id) ? ;

Self :: safe_transfer (storage , token_id , from , to , data .0) }

/// Equivalent to [safe_transfer_from_with_data], but without the additional data. /// /// Note: because Rust doesn't allow multiple methods with the same name, /// we use the #[selector] macro attribute to simulate solidity overloading.

[selector(name =

"safeTransferFrom")] pub

fn

safe_transfer_from < S :

TopLevelStorage

+

BorrowMut < Self

(storage :

& mut

S , from :

Address , to :

Address , token_id :

U256 ,)

->

Result < ( ) ,

Erc721Error

{ Self :: safe_transfer_from_with_data (storage , from , to , token_id ,

Bytes (vec! [ ])) }

/// Transfers the NFT. pub

fn

transfer_from (& mut

self , from :

Address , to :

Address , token_id :

U256)

->

Result < ( ) ,

Erc721Error

{ if to . is_zero ( )

{ return

Err (Erc721Error :: TransferToZero (TransferToZero

{ token_id })) ; } self . require_authorized_to_spend (from , token_id) ? ; self . transfer (token_id , from , to) ? ; Ok (( )) }

```
/// Grants an account the ability to manage the sender's NFT. pub
fn
approve ( & mut
self , approved :
Address , token_id :
U256 )
->
Result < ( ) ,
Erc721Error
{ let owner =
self . owner_of ( token_id ) ? ;
// require authorization if
msg :: sender ( )
!= owner &&
! self . operator_approvals . getter ( owner ) . get ( msg :: sender ( ) )
{ return
Err ( Erc721Error :: NotApproved ( NotApproved
{ owner , spender :
msg :: sender ( ) , token_id , } ) ) ; } self . token_approvals . insert ( token_id , approved ) ;
evm :: log ( Approval
{ approved , owner , token_id , } ) ; Ok ( ( ) ) }
/// Grants an account the ability to manage all of the sender's NFTs. pub
fn
set_approval_for_all ( & mut
self , operator :
Address , approved :
bool )
->
Result < ( ) ,
Erc721Error
{ let owner =
msg :: sender ( ) ; self . operator_approvals . setter ( owner ) . insert ( operator , approved ) ;
evm :: log ( ApprovalForAll
{ owner , operator , approved , } ) ; Ok ( ( ) ) }
/// Gets the account managing an NFT, or zero if unmanaged. pub
fn
get_approved ( & mut
```

```
self , token_id :
U256 )
->
Result < Address ,
Erc721Error
{ Ok ( self . token_approvals . get ( token_id ) ) }
/// Determines if an account has been authorized to managing all of a user's NFTs. pub
fn
is_approved_for_all ( & mut
self , owner :
Address , operator :
Address )
->
Result < bool ,
Erc721Error
{ Ok ( self . operator_approvals . getter ( owner ) . get ( operator ) ) }
/// Whether the NFT supports a given standard. pub
fn
supports_interface ( interface :
FixedBytes < 4
        )
->
Result < bool ,
Erc721Error
{ let interface_slice_array :
[ u8 ;
4 ]
= interface . as_slice ( ) . try_into ( ) . unwrap ( ) ;
if
u32 :: from_be_bytes ( interface_slice_array )
==
0xffffffff
{ // special cased in the ERC165 standard return
Ok ( false ) ; }
const
IERC165 :
u32
```

=

0x01ffc9a7 ; const

IERC721 :

u32

=

0x80ac58cd ; const

IERC721_METADATA :

u32

=

0x5b5e139f ;

Ok (matches! (u32 :: from_be_bytes (interface_slice_array) ,

IERC165

|

IERC721

|

IERC721_METADATA)) } }

lib.rs

// Only run this as a WASM if the export-abi feature is not set.

![cfg_attr(not(any(feature =

"export-abi" , test)), no_main)] extern

crate

alloc ;

// Modules and imports mod

erc721 ;

use

alloy_primitives :: { U256 ,

Address } ; /// Import the Stylus SDK along with alloy primitive types for use in our program. use

stylus_sdk :: { msg ,

prelude :: * } ; use

crate :: erc721 :: { Erc721 ,

Erc721Params ,

Erc721Error } ;

/// Immutable definitions struct

StylusNFTParams ; impl

Erc721Params

for

StylusNFTParams

{ const

NAME :

& 'static

str

=

"StylusNFT" ; const

SYMBOL :

& 'static

str

=

"SNFT" ;

fn

token_uri (token_id :

U256)

->

String

{ format! ("{}{}{}" ,

"https://my-nft-metadata.com/" , token_id ,

".json") } }

// Define the entrypoint as a Solidity storage object. The sol_storage! macro // will generate Rust-equivalent structs with all fields mapped to Solidity-equivalent // storage slots and types. sol_storage!

{

[entrypoint]

struct

StylusNFT

{

[borrow]

// Allows erc721 to access StylusNFT's storage and make calls Erc721 < StylusNFTParams

erc721 ; } }

[public]

[inherit(Erc721)]

impl

StylusNFT

{ /// Mints an NFT pub

fn

mint (& mut

self)

->

Result < ( ) ,

Erc721Error

{ let minter =

msg :: sender ( ) ; self . erc721 . mint (minter) ? ; Ok (( )) }

/// Mints an NFT to another address pub

fn

mint_to (& mut

self , to :

Address)

->

Result < ( ) ,

Erc721Error

{ self . erc721 . mint (to) ? ; Ok (( )) }

/// Burns an NFT pub

fn

burn (& mut

self , token_id :

U256)

->

Result < ( ) ,

Erc721Error

{ // This function checks that msg::sender() owns the specified token_id self . erc721 . burn (msg :: sender ( ) , token_id) ? ; Ok (( )) }

/// Total supply pub

fn

total_supply (& mut

self)

->

Result < U256 ,

Erc721Error

{ Ok (self . erc721 . total_supply . get ( )) } }

Cargo.toml

[package] name

=

"stylus_erc721_example" version

=

"0.1.7" edition

=

"2021" license

=

"MIT OR Apache-2.0" keywords

=

["arbitrum" ,

"ethereum" ,

"stylus" ,

"alloy"]

[dependencies] alloy-primitives

=

"=0.7.6" alloy-sol-types

=

"=0.7.6" mini-alloc

=

"0.4.2" stylus-sdk

=

"0.6.0" hex

=

"0.4.3"

[dev-dependencies] tokio

=

{

version

=

"1.12.0" ,

features

=

["full"]

} ethers

=

"2.0" eyre

=

"0.6.8"

[features] export-abi

=

["stylus-sdk/export-abi"]

[lib] crate-type

=

["lib" ,

"cdylib"]

[profile.release] codegen-units

=

1 strip

=

true lto

=

true panic

=

"abort" opt-level

=

"s" Edit this page Previous Erc20 Next Multi Call