Stylus Rust SDK overview

ALPHA RELEASE, PUBLIC PREVIEW DOCS Stylus is currently tagged as analpha release. The code has not been audited, and shouldnot be used in production scenarios . This documentation is currently inpublic preview .

To provide feedback, click theRequest an update button at the top of this documentjoin the Arbitrum Discord , or reach out to our team directly by completingthis form . This document provides an in-depth overview of the features provided by theStylus Rust SDK . For information about deploying Rust smart contracts, see thecargo stylusCLI Tool . For a conceptual introduction to Stylus, seeStylus: A Gentle Introduction . To deploy your first Stylus smart contract using Rust, refer to theQuickstart .

The Stylus Rust SDK is built on top ofAlloy , a collection of crates empowering the Rust Ethereum ecosystem. Because the SDK uses the sameRust primitives for Ethereum types , Stylus is compatible with existing Rust libraries.

info Many of the affordances use macros. Though this document details what each does, it may be helpful to useeargo expand to see what they expand into if you're doing advanced work in Rust. Additionally, the Stylus SDK supports#[no_std] for contracts that wish to opt out of the standard library. In fact, the entire SDK is available from#[no_std] , so no special feature flag is required. This can be helpful for reducing binary size, and may be preferable in pure-compute use cases like cryptography.

Most users will want to use the standard library, which is available since the Stylus VM supportsrustc 'swasm32-unknown-unknown target triple. In the future we may addwasm32-wasi too, along with floating point and SIMD, which the Stylus VM does not yet support.

Storage

Rust smart contracts may use state that persists across transactions. There's two primary ways to define storage, depending on if you want to use Rust or Solidity definitions. Both are equivalent, and are up to the developer depending on their needs.

#[solidity_storage]

The#[solidity_storage] macro allows a Rust struct to be used in persistent storage.

[solidity_storage]

pub

struct

Contract

{ owner :

StorageAddress , active :

StorageBool , sub_struct :

SubStruct , }

[solidity_storage]

pub

struct

SubStruct

{ // types implementing the StorageType trait. } Any type implementing theStorageType trait may be used as a field, including other structs, which will implement the trait automatically when#[solidity_storage] is applied. You can even implementStorageType yourself to define custom storage types. However, we've gone ahead and implemented the common ones.

Type Info StorageBool Stores a bool StorageAddress Stores an AlloyAddress StorageUint Stores an AlloyUint StorageSigned Stores an AlloySigned StorageFixedBytes Stores an AlloyFixedBytes StorageBytes Stores a Solidity bytes

StorageString Stores a Solidity string StorageVec Stores a vector ofStorageType StorageMap Stores a mapping ofStorageKey toStorageType StorageArray Stores a fixed-sized array ofStorageType EveryAlloy primitive has a correspondingStorageType implementation with the wordStorage before it. This includes aliases, likeStorageU256 andStorageB64 .

sol_storage!

The types in#[solidity_storage] are laid out in the EVM state trie exactly as they are inSolidity . This means that the fields of a struct definition will map to the same storage slots as they would in EVM programming languages.

Because of this, it is often nice to define your types using Solidity syntax, which makes that guarantee easier to see. For example, the earlier Rust struct can re-written to:

sol_storage!

{ pub

struct

Contract

{ address owner ;

// becomes a StorageAddress bool active ;

// becomes a StorageBool SubStruct sub_struct , }

pub

struct

SubStruct

{ // other solidity fields, such as mapping (address => uint) balances ;

// becomes a StorageMap Delegate delegates [ ] ;

// becomes a StorageVec } } The above will expand to the equivalent definitions in Rust, each structure implementing theStorageType trait. Many contracts, likeour example ERC 20 , do exactly this.

Because the layout is identical toSolidity's , existing Solidity smart contracts can upgrade to Rust without fear of storage slots not lining up. You simply copy-paste your type definitions.

tip Existing Solidity smart contracts can upgrade to Rust if they use proxy patterns. Consequently, the order of fields will affect the JSON ABIs produced that explorers and tooling might use. Most developers won't need to worry about this though and can freely order their types when working on a Rust contract from scratch.

Reading and writing storage

You can access storage types via getters and setters. For example, theContract struct from earlier might access itsowner address as follows.

impl

Contract

{ /// Gets the owner from storage. pub

fn

owner (& self)

->

Result < Address ,

Vec < u8

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

/// Updates the owner in storage pub

fn

set_owner (& mut

self , new_owner :

Address)

->

Result < ( ) ,

Vec < u8

{ if

msg :: sender ( )

==

self . owner ( ) ?

{

// we'll discuss msg::sender later self . owner . set (new_owner) ; } Ok (( )) } } In Solidity, one has to be very careful about storage access patterns. Getting or setting the same value twice doubles costs, leading developers to avoid storage access at all costs. By contrast, the Stylus SDK employs an optimal storage-caching policy that avoids the underlyingSLOAD orSSTORE operations.

tip Stylus uses storage caching, so multiple accesses of the same variable is virtually free. However it must be said that storage is ultimately more expensive than memory. So if a value doesn’t need to be stored in state, you probably shouldn’t do it.

Collections

Collections likeStorageVec andStorageMap are dynamic and have methods likepush ,insert ,replace , and similar.

impl

SubStruct

{ pub

fn

add_delegate (& mut

self , delegate :

Delegate)

->

Result < ( ) ,

Vec < u8

{ self . delegates . push (delegate) ; }

pub

fn

track_balance (& mut

self , address :

Address)

->

Result < U256 ,

Vec < u8

{ self . balances . insert (address , address . balance ( )) ; } } You may notice that some methods return types likeStorageGuard andStorageGuardMut . This allows us to leverage the Rust borrow checker for storage mistakes, just like it does for memory. Here's an example that will fail to compile.

fn

mistake (vec :

& mut

StorageVec < StorageU64

)

->

U64

{ let value = vec . setter (0) ; let alias = vec . setter (0) ; value . set (32 . into ( )) ; alias . set (48 . into ( )) ; value . get ( )

// uh, oh. what value should be returned? } Under the hoodvec.setter() returns aStorageGuardMut instead of a&mut StorageU64 . Because the guard is bound to a&mut StorageVec lifetime,value andalias cannot be alive simultaneously. This causes the Rust compiler to reject the above code, saving you from entire classes of storage aliasing errors.

In this way the Stylus SDK safeguards storage access the same way Rust ensures memory safety. It should never be possible to alias Storage withoutunsafe Rust.

SimpleStorageType

You may run into scenarios where a collection's methods likepush andinsert aren't available. This is because only primitives, which implement a special trait calledSimpleStorageType , can be added to a collection by value. For nested collections, one instead uses the equivalentgrow andsetter .

fn

nested_vec (vec :

& mut

StorageVec < StorageVec < StorageU8

)

{ let

mut inner = vec . grow ( ) ;

// adds a new element accessible via inner inner . push (0 . into ( )) ;

// inner is a guard to a StorageVec }

fn

nested_map (map :

& mut

StorageMap < u32 ,

StorageVec < U8

)

{ let

mut slot = map . setter (0) ; slot . push (0) ; }

Erase

and#[derive(Erase)]

SomeStorageType values implementErase , which provides anerase() method for clearing state. We've implementedErase for all primitives, and for vectors of primitives, but not maps. This is because a solditymapping does not provide iteration, and so it's generally impossible to know which slots to set to zero.

Structs may also beErase if all of the fields are#[derive(Erase)] lets you do this automatically.

sol_storage!

{

[derive(Erase)]

pub

struct

Contract

{ address owner ;

// can erase primitive uint256 [ ] hashes ;

// can erase vector of primitive }

pub

struct

NotErase

{ mapping (address => uint) balances ;

// can't erase a map mapping (uint => uint) [ ] roots ;

// can't erase vector of maps } } You can also implementErase manually if desired. Note that the reason we care aboutErase at all is that you get storage refunds when clearing state, lowering fees. There's also minor implications for patterns usingunsafe Rust.

The storage cache

The Stylus SDK employs an optimal storage-caching policy that avoids the underlyingSLOAD orSSTORE operations needed to get and set state. For the vast majority of use cases, this happens in the background and requires no input from the user.

However, developers working withunsafe Rust implementing their own customStorageType collections, theStorageCache type enables direct control over this data structure. Included areunsafe methods for manipulating the cache directly, as well as for bypassing it altogether.

Immutables andPhantomData

So that generics are possible insol_interface! ,core::marker::PhantomData implementsStorageType and takes up zero space, ensuring that it won't cause storage slots to change. This can be useful when writing libraries.

pub

trait

Erc20Params

{ const

NAME :

& 'static

str ; const

SYMBOL :

& 'static

str ; const

DECIMALS :

u8 ; }

sol_storage!

{ pub

struct

Erc20 < T

{ mapping (address => uint256) balances ; PhantomData < T

phantom ; } } The above allows consumers ofErc20 to choose immutable constants via specialization. See ourWETH sample contract for a full example of this feature.

Future storage work

The Stylus SDK is currently in alpha and will improve in the coming versions. Something you may notice is that storage access patterns are often a bit verbose. This will change soon when we implementDerefMut for most types.

Methods

Just as with storage, Stylus SDK methods are Solidity ABI equivalent. This means that contracts written in different programming languages are fully interoperable. As detailed in this section, you can even automatically export your Rust contract as a Solidity interface so that others can add it to their Solidity projects.

tip Stylus programs compose regardless of programming language. For example, existing Solidity DEXs can list Rust ERC-20s without modification.

#[external]

This macro makes methods “external” so that other contracts can call them by implementing theRouter trait.

[external]

impl

Contract

{ // our owner method is now callable by other contracts pub

fn

owner (& self)

->

Result < Address ,

Vec < u8

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

impl

Contract

{ // our set_owner method is not pub

fn

set_owner (& mut

self , new_owner :

Address)

->

Result < ( ) ,

Vec < u8

{ ... } } Note that, currently, all external methods must return a Result with the error type Vec . We intend to change this very soon. In the current model, Vec becomes the program's revert data, which we intend to both make optional and richly typed.

#[payable]

As in Solidity, methods may accept ETH as call value.

[external]

impl

Contract

{

[payable]

pub

fn

credit (& mut

self)

->

Result < ( ) ,

Vec < u8

{ self . erc20 . add_balance (msg :: sender ( ) ,

msg :: value ( )) } } In the above, msg::value is the amount of ETH passed to the contract in wei, which may be used to pay for something depending on the contract's business logic. Note that you have to annotate the method with #[payable] , or else calls to it will revert. This is required as a safety measure to prevent users losing funds to methods that didn't intend to accept ether.

#[pure]

, #[view] , and #[write]

For aesthetics, these additional purity attributes exist to clarify that a method is pure , view , or write . They aren't necessary though, since the #[external] macro can figure purity out for you based on the types of the arguments.

For example, if a method includes an &self , it's at least view . If you'd prefer it be write , applying #[write] will make it so. Note however that the reverse is not allowed. An &mut self method cannot be made #[view] , since it might mutate state.

#[entrypoint]

This macro allows you to define the entrypoint, which is where Stylus execution begins. Without it, the contract will fail to pass cargo stylus check . Most commonly, the macro is used to annotate the top level storage struct.

sol_storage!

{

[entrypoint]

pub

struct

Contract

{ ... }

// only one entrypoint is allowed pub

struct

SubStruct

{ ... } } The above will make the external methods ofContract the first to consider during invocation. In a later section we'll discuss inheritance, which will allow the#[external] methods of other types to be invoked as well.

Bytes-in, bytes-out programming

A less common usage of#[entrypoint] is for low-level, bytes-in bytes-out programming. When applied to a free-standing function, a different way of writing smart contracts becomes possible, where the Rust SDK's macros and storage types are entirely optional.

[entrypoint]

fn

entrypoint (calldata :

Vec < u8

)

->

ArbResult

{ // bytes-in, bytes-out programming }

Reentrancy

If a contract calls another that then calls the first, it is said to be reentrant. By default, all Stylus programs revert when this happens. However, you can opt out of this behavior by enabling thereentrant feature flag.

stylus

sdk

{ version =

"0.3.0" , features =

["reentrant"]

} This is dangerous, and should be done only after careful review — ideally by 3rd party auditors. Numerous exploits and hacks have in Web3 are attributable to developers misusing or not fully understanding reentrant patterns.

If enabled, the Stylus SDK will flush the storage cache in between reentrant calls, persisting values to state that might be used by inner calls. Note that preventing storage invalidation is only part of the battle in the fight against exploits. You can tell if a call is reentrant viamsg::reentrant , and condition your business logic accordingly.

TopLevelStorage

The#[entrypoint] macro will automatically implement theTopLevelStorage trait for the annotatedstruct . The single type implementingTopLevelStorage is special in that mutable access to it represents mutable access to the entire program's state. This idea will become important when discussing calls to other programs in later sections.

Inheritance,#[inherit]

, and#[borrow] .

Composition in Rust follows that of Solidity. Types that implementRouter , the trait that#[external] provides, can be connected via inheritance.

[external]

[inherit(Erc20)]

impl

Token

{ pub

fn

mint (& mut

self , amount :

U256)

->

Result < ( ) ,

Vec < u8

{ ... } }

[external]

impl

Erc20

{ pub

fn

balance_of ( )

->

Result < U256

{ ... } } BecauseToken inheritsErc20 in the above, ifToken has the#[entrypoint] , calls to the contract will first check if the requested method exists withinToken . If a matching function is not found, it will then try theErc20 . Only after trying everythingToken inherits will the call revert.

Note that because methods are checked in that order, if both implement the same method, the one inToken will override the one inErc20 , which won't be callable. This allows for patterns where the developer imports a crate implementing a standard, like the ERC 20, and then adds or overrides just the methods they want to without modifying the importedErc20 type.

Inheritance can also be chained.#[inherit(Erc20, Erc721)] will inherit bothErc20 andErc721 , checking for methods in that order.Erc20 andErc721 may also inherit other types themselves. Method resolution finds the first matching method byDepth First Search .

Note that for the above to work,Token must implementBorrow . You can implement this yourself, but for simplicity#

[solidity_storage] andsol_storage! provide a#[borrow] annotation.

sol_storage!

{

[entrypoint]

pub

struct

Token

{

[borrow]

Erc20 erc20 ; ... }

pub

struct

Erc20

{ ... } } In the future we plan to simplify the SDK so thatBorrow isn't needed and so thatRouter composition is more configurable. The motivation for this becomes clearer in complex cases of multi-level inheritance, which we intend to improve.

Exporting a Solidity interface

Recall that Stylus contracts are fully interoperable across all languages, including Solidity. The Stylus SDK provides tools for exporting a Solidity interface for your contract so that others can call it. This is usually done with thecargo stylus CLI tool , but we'll detail how to do it manually here.

The SDK does this automatically for you via a feature flag calledexport-abi that causes the#[external] and#[entrypoint] macros to generate amain function that prints the Solidity ABI to the console.

cargo run

- features export - abi - - target < triple

> Note that because the above actually generates amain function that you need to run, the target can't bewasm32-unknown-unknown like normal. Instead you'll need to pass in your target triple, whichcargo stylus figures out for you. Thismain function is also why the following commonly appears in themain.rs file of Stylus contracts.

![cfg_attr(not(feature =

"export-abi"), no_main)] Here's an example output. Observe that the method names change from Rust'ssnake_case to Solidity'scamelCase . For compatibility reasons, onchain method selectors are alwayscamelCase . We'll provide the ability to customize selectors very soon. Note too that you can use argument names likeaddress without fear. The SDK will prepend an_ when necessary.

interface

Erc20

{ function

name ( )

external

pure

returns

(string

memory) ;

function

balanceOf (address _address)

external

view

returns

(uint256) ; }

interface

Weth

is Erc20 { function

mint ( )

external

payable ;

function

burn (uint256 amount)

external ; }

Calls

Just as with storage and methods, Stylus SDK calls are Solidity ABI equivalent. This means you never have to know the implementation details of other contracts to invoke them. You simply import the Solidity interface of the target contract, which can be auto-generated via thecargo stylus CLI tool .

tip You can call contracts in any programming language with the Stylus SDK.

sol_interface!

This macro defines astruct for each of the Solidity interfaces provided.

sol_interface!

{ interface IService

{ function makePayment (address user) payable returns

(string) ; function getConstant ( ) pure returns

(bytes32) }

interface ITree

{ // other interface methods } } The above will defineIService andITree for calling the methods of the two contracts.

For example,IService will have amake_payment method that accepts anAddress and returns aB256 .

pub

fn

do_call (& mut

self , account :

IService , user :

Address)

->

Result < String ,

Error

{ account . make_payment (self , user)

// note the snake case } Observe the casing changesol_interface! computes the selector based on the exact name passed in, which should almost always beCamelCase . For aesthetics, the rust functions will instead usesnake_case .

Configuring gas and value withCall

Call lets you configure a call via optional configuration methods. This is similar to how one would configure opening a File in Rust.

pub

fn

do_call (account :

IService , user :

Address)

->

Result < String ,

Error

{ let config =

Call :: new ( ) . gas (evm :: gas_left ( )

/

2)

// limit to half the gas left . value (msg :: value ( )) ;

// set the callvalue

account . make_payment (config , user) } By defaultCall supplies all gas remaining and zero value, which often meansCall::new() may be passed to the method directly. Additional configuration options are available in cases of reentrancy.

Reentrant calls

Contracts that opt into reentrancy via thereentrant feature flag require extra care. When thestorage-cache feature is enabled, cross-contract calls mustflush orclear theStorageCache to safeguard state. This happens automatically via the type system.

sol_interface!

{ interface IMethods

{ function pureFoo ( ) pure ; function viewFoo ( ) view ; function writeFoo ( ) ; function payableFoo ( ) payable ; } }

[external]

impl

Contract

{ pub

fn

call_pure (& self , methods :

IMethods)

->

Result < ( ) ,

Vec < u8

{ Ok (methods . pure_foo (self) ?)

// pure methods might lie about not being view }

pub

fn

call_view (& self , methods :

IMethods)

->

Result < ( ) ,

Vec < u8

{ Ok (methods . view_foo (self) ?) }

pub

fn

call_write (& mut

self , methods :

IMethods)

->

Result < ( ) ,

Vec < u8

{ methods . view_foo (self) ? ;

// allows pure and view methods too Ok (methods . write_foo (self) ?) }

[payable]

pub

fn

call_payable (& mut

self , methods :

IMethods)

->

Result < ( ) ,

Vec < u8

{ methods . write_foo (Call :: new_in (self)) ? ;

// these are the same Ok (methods . payable_foo (self) ?)

// ---------- } } In the above, we're able to pass&self and&mut self becauseContract implementsTopLevelStorage , which means that a reference to it entails access to the entirety of the contract's state. This is the reason it is sound to make a call, since it ensures all cached values are invalidated and/or persisted to state at the right time.

When writing Stylus libraries, a type might not beTopLevelStorage and therefore&self or&mut self won't work. Building aCall from a generic parameter vianew_in is the usual solution.

pub

fn

do_call (storage :

& mut

impl

TopLevelStorage ,

// can be generic, but often just &mut self account :

IService ,

// serializes as an Address user :

Address ,)

->

Result < String ,

Error

{

let config =

Call :: new_in (storage)

// take exclusive access to all contract storage . gas (evm :: gas_left ( )

/

2)

// limit to half the gas left . value (msg :: value ( )) ;

// set the callvalue

account . make_payment (config , user)

// note the snake case } Note that in the context of an#[external] call, the&mut impl argument will correctly distinguish the method as beingwrite orpayable . This means you can write library code that will work regardless of whether the reentrant feature flag is enabled.

Note too thatCall::new_in should be used instead ofCall::new since the former provides access to storage. Code that previously compiled with reentrancy disabled may require modification in order to type-check. This is done to ensure storage changes are persisted and that the storage cache is properly managed before calls.

call

,static_call , anddelegate_call

Thoughsol_interface! andCall form the most common idiom to invoke other contracts, their underlyingcall andstatic_call are exposed for direct access.

let return_data =

call (Call :: new ( ) , contract , call_data) ? ; In each case the calldata is supplied as aVec . The return result is either the raw return data on success, or a callError on failure.

delegate_call is also available, though it'sunsafe and doesn't have a richly-typed equivalent. This is because a delegate call must trust the other contract to uphold safety requirements. Though this function clears any cached values, the other

contract may arbitrarily change storage, spend ether, and do other things one should never blindly allow other contracts to do.

transfer_eth

This method provides a convenient shorthand for transferring ether.

Note that this method invokes the other contract, which may in turn call others. All gas is supplied, which the recipient may burn. If this is not desired, thecall function may be used instead.

transfer_eth (recipient , value) ? ;

// these two are equivalent

call (Call :: new ( ) . value (value) , recipient ,

& [ ]) ? ;

// these two are equivalent

RawCall

andunsafe calls

Occasionally, an untyped call to another contract is necessaryRawCall lets you configure anunsafe call by calling optional configuration methods. This is similar to how one would configure opening aFile in Rust.

let data =

RawCall :: new_delegate ( )

// configure a delegate call . gas (2100)

// supply 2100 gas . limit_return_data (0 ,

32)

// only read the first 32 bytes back . flush_storage_cache ( )

// flush the storage cache before the call . call (contract , calldata) ? ;

// do the call Note that thecall method isunsafe when reentrancy is enabled. Seeflush_storage_cache andclear_storage_cache for more information.

RawDeploy

andunsafe deployments

Right now the only way to deploy a contract from inside Rust is to useRawDeploy , similar toRawCall . As withRawCall , this mechanism is inherently unsafe due to reentrancy concerns, and requires manual management of theStorageCache .

Note that the EVM allows init code to make calls to other contracts, which provides a vector for reentrancy. This means that this technique may enable storage aliasing if used in the middle of a storage reference's lifetime and if reentrancy is allowed.

When configured with asalt ,RawDeploy will useCREATE2 instead of the defaultCREATE , facilitating address determinism.

Events

Emitting Solidity-style events is supported out-of-the-box with the Rust SDK. They can be defined in Solidity syntax using Alloy'ssol! macro, and then used as input arguments toevm::log . The function accepts any type that implements Alloy'sSolEvent trait.

sol!

{ event Transfer (address indexed from , address indexed to , uint256 value) ; event Approval (address indexed owner , address indexed spender , uint256 value) ; }

fn

foo ( )

{ ... evm :: log (Transfer

{ from :

Address :: ZERO , to : address , value , }) ; } The SDK also exposes a low-level evm::raw_log that takes in raw bytes and topics:

/// Emits an evm log from combined topics and data. The topics come first. fn

emit_log (bytes :

& [u8] , topics :

usize)

EVM affordances

The SDK contains several modules for interacting with the EVM, which can be imported like so.

use

stylus_sdk :: { block , contract , crypto , evm , msg , tx } ;

let callvalue =

msg :: value ( ) ; Rust SDK Module Description block block info for the number, timestamp, etc. contract contract info, such as its address, balance crypto VM accelerated cryptography evm ink / memory access functions msg sender, value, and reentrancy detection tx gas price, ink price, origin, and other tx-level info Edit this page Last updated on Mar 7, 2024 Previous Opcode and host I/O pricing Next Stylus SDK repositories