How to add a new programming language to Stylus

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 . Arbitrum Stylus is a new technology developed for Arbitrum chains which gives smart contract developers superpowers. With Stylus, developers can write EVM-compatible smart contracts in many different programming languages, and reap massive performance gains. Stylus slashes fees, with performance gains ranging from 10-70x, and memory efficiency gains as high as 100-500x.

This is possible thanks toWebAssembly technology, which all Stylus programs compile to. Stylus smart contracts live under thesame Ethereum state trie in Arbitrum nodes, and can fully interoperate with Solidity or Vyper EVM smart contracts. With Stylus, developers can write smart contracts in Rust that talk to Solidity and vice versa without any limitations.

Today, the Stylus testnet also comes with 2 officially supportedSDKs for developers to write contracts in theRust orC programming languages.

However,anyone can add support for new languages in Stylus.As long as a programming language can compile to WebAssembly , Stylus will let you use it to write EVM-compatible smart contracts. Note that in order to be deployed onchain, your compiled program must fit under the 24Kb brotli-compressed limit, and should meet Stylus gas metering requirements.

In this document, we go over how we added support for the up-and-comingZig programming language, which is meant to be a spiritual successor to C that comes with great performance and memory safetywithin 20 lines of code .

Why Zig?

1. Zig containsmemory safety guardrails
2. , requiring developers to think hard about manual memory allocation in a prudent manner
3. Zig is aC equivalent
4. language, and its tooling is also a C compiler. This means C projects can incrementally adopt Zig when refactoring
5. Zig islightning fast
6. and producessmall binaries
7. , making it suitable for blockchain applications

Programs written in Zig and deployed to Stylus have a tiny footprint and will have gas costs comparable, if not equal to, C programs.

Requirements

- Download and installZig 0.11.0
- InstallRust
- , which we'll need for theStylus CLI tool
- to deploy our program to the Stylus testnet

We'll also be using Rust to run an example script that can call our Zig contract on the Stylus testnet using the popularethers-rs library.

Once Rust is installed, also install the Stylus CLI tool with

RUSTFLAGS

"-C link-args=-rdynamic" cargo install --force cargo-stylus

Using Zig with Stylus

First, let's clone the repository:

git clone https://github.com/offchainlabs/zig-on-stylus &&

cd zig-on-stylus then delete everything inside ofmain.zig . We'll be filling it out ourselves in this tutorial.

To support Stylus, your Zig programs need to define a special entrypoint function, which takes in the length of its input args,len , and returns a status codei32 , which is either 0 or 1. We won't need the Zig standard library for this.

One more thing it needs is to use a special function, calledmemory_grow which can allocate memory for your program. This function isinjected into all Stylus contracts as an external import. Internally, we call thesevm_hooks , and also refer to them

ashost-io's , because they give you access to the host, EVM environment.

Go ahead and replace everything in yourmain.zig function with:

pub extern

"vm_hooks" fn memory_grow (len : u32)

void ;

export fn mark_unused ( )

void

{ memory_grow (0) ; @ panic ("") ; }

// The main entrypoint to use for execution of the Stylus WASM program. export fn user_entrypoint (len : usize) i32 { _ = len ; return

0 ; } At the top, we declare thememory_grow external function for use.

Next, we can build our Zig library to a freestanding WASM file for our onchain deployment:

zig build-lib ./src/main.zig -target wasm32-freestanding -dynamic --export

user_entrypoint -OReleaseSmall --export

mark_unused This is enough for us to deploy on the Stylus testnet! We'll use theStylus CLI tool , which you installed earlier usingcargo install :

cargo stylus deploy --private-key=--wasm-file-path=main.wasm The tool will send two transactions: one to deploy your Zig contract's code onchain, and the other to activate it for usage.

Uncompressed WASM size: 112 B Compressed WASM size to be deployed onchain: 103 B You can see that our Zig program istiny when compiled to WASM. Next, we can call our contract to make sure it works using any of your favorite Ethereum tooling. In this example below, we use thecast CLI tool provided byfoundry . The contract above has been deployed to the Stylus testnet at address0xe0CD04EA8c148C9a5A58Fee1C895bc2cf6896799 .

export ADDR=0xe0CD04EA8c148C9a5A58Fee1C895bc2cf6896799 cast call --rpc-url 'https://stylus-testnet.arbitrum.io/rpc' ADDR '0x' Calling the contract via RPC should simply return the value0 as we programmed it to.

0x

Reading input and writing output data

Smart contracts on Ethereum, at the bare minimum, can take in data and output data as bytes. Stylus programs are no different, and to do anything useful, we need to be able to read from user input also write our output to the caller. To do this, the Stylus runtime provides all Stylus programs with two additional, useful host-ios:

pub extern

"vm_hooks" fn read_args (dest :

* u8)

void ; pub extern

"vm_hooks" fn write_result (data :

* const u8 , len : usize)

void ; Add these near the top of yourmain.zig file.

The first,read_args takes in a pointer to a byte slice where the input arguments will be written to. The length of this byte slice must equal the length of the program args received in theuser_entrypoint . We can write a helper function that uses this vm hook and gives us a byte slice in Zig we can then operate on.

// Allocates a Zig byte slice of length=len reads a Stylus contract's calldata // using the read_args hostio function. pub fn input

(len : usize)

! [ ] u8 { var input = try allocator . alloc (u8 , len) ; read_args (@ ptrCast (* u8 , input)) ; return input ; } Next, we implement a helper function that outputs the data bytes to the Stylus contract's caller:

// Outputs data as bytes via the write_result hostio to the Stylus contract's caller. pub fn output (data :

[ ] u8)

void

{ write_result (@ ptrCast (* u8 , data) , data . len) ; } Let's put these together:

// The main entrypoint to use for execution of the Stylus WASM program. // It echoes the input arguments to the caller. export fn user_entrypoint (len : usize) i32 { var in =

input (len) catch return

1 ; output (in) ; return

0 ; } We're almost good to go, let's try to compile to WASM and deploy to the Stylus testnet. Let's run our build command again:

src/main.zig:21:20: error: use of undeclared identifier 'allocator' var data = try allocator.alloc(u8, len); ^~~~~~~~~ Oops! Looks like we need an allocator to do our job here. Zig, as a language, requires programmers to think carefully about memory allocation and it's a typical pattern to require them to manually provide an allocator. There are many to choose from, but the Zig standard library already has one built specifically for WASM programs. Memory in WASM programs grows in increments of 64Kb, and the allocator from the stdlib has us covered here.

Let's try to use it by adding the following to the top of ourmain.zig

const std = @ import ("std") ; const allocator = std . heap . WasmAllocator ; Our code compiles, but will it deploy onchain? Runcargo stylus check --wasm-file-path=main.wasm and see:

Caused by: missing import memory_grow What's wrong? This means that the WasmAllocator from the Zig standard library should actually be using our specialmemory_grow hostio function underneath the hood. We can fix this by copying over the WasmAllocator.zig file from the standard library, and modifying a single line to usememory_grow .

You can find this file underWasmAllocator.zig in the OffchainLabs/zig-on-stylus repository. We can now use it:

const std = @ import ("std") ; const WasmAllocator = @ import ("WasmAllocator.zig") ;

// Uses our custom WasmAllocator which is a simple modification over the wasm allocator // from the Zig standard library as of Zig 0.11.0. pub const allocator = std . mem . Allocator { . ptr = undefined , . vtable =

& WasmAllocator . vtable , } ; Building again and runningcargo stylus check should now succeed:

Uncompressed WASM size: 514 B Compressed WASM size to be deployed onchain: 341 B Connecting to Stylus RPC endpoint: https://stylus-testnet.arbitrum.io/rpc Stylus program with same WASM code is already activated onchain Let's deploy it:

cargo stylus deploy --private-key=--wasm-file-path=main.wasm Now if we try to call it, it will output whatever input we send it, like an echo. Let's send it the input 0x123456:

export ADDR=0x20Aa65a9D3F077293993150C0345f62B50CCb549 cast call --rpc-url 'https://stylus-testnet.arbitrum.io/rpc' ADDR '0x123456'

0x123456 Works!

Prime number checker implementation

Let's build something a little bit fancier: this time we'll implement a primality checker in Zig using an ancient algorithm called thesieve of erathosthenes . Given a number, our contract will output 1 if it is prime, or 0 otherwise. We'll implement in a pretty naive way, but leverage one of Zig's awesome features:comptime .

Thecomptime keyword tells the Zig compiler to evaluate the code involved at compile time, allowing you to define computation that would normally make runtime more expensive and do it while your binary is being compiled! Comptime in Zig is extremely flexible. In this example, we use it to define a slice of booleans up to a certain limit at compile time, which we'll use to mark which numbers are prime or not.

fn sieve_of_erathosthenes (comptime limit : usize , nth : u16) bool { var prime =

[_] bool { true }

* * limit ; prime [0]

= false ; prime [1]

= false ; var i : usize =

2 ; while

(i * i < limit)

:

(i +=

1)

{ if

(prime [i])

{ var j = i * i ; while

(j < limit)

:

(j += i) prime [j]

= false ; } } return prime [nth] ; } Checking if a number N is prime would involve just checking if the value at index N in thisprime boolean slice is true. We can then integrate this function into ouruser_entrypoint :

// The main entrypoint to use for execution of the Stylus WASM program. export fn user_entrypoint (len : usize) i32 { // Expects the input is a u16 encoded as little endian bytes. var input =

args (len) catch return

1 ; var check_nth_prime = std . mem . readIntSliceLittle (u16 , input) ; const limit : u16 =

10 _000 ; if

(check_nth_prime

limit)

{ @ panic ("input is greater than limit of 10,000 primes") ; } // Checks if the number is prime and returns a boolean using the output function. var is_prime =

sieve_of_erathosthenes (limit , check_nth_prime) ; var out = input [0. .1] ; if

(is_prime)

{ out [0]

=

1 ; }

else

{ out [0]

=

0 ; } output (out) ; return

0 ; } Let's check and deploy it:

Uncompressed WASM size: 10.8 KB Compressed WASM size to be deployed onchain: 525 B Our uncompressed size is big because of that giant array of booleans, but the program is highly compressible because all of them are zeros!

An instance of this program has been deployed to the Stylus testnet at address0x0c503Bb757b1CaaD0140e8a2700333C0C9962FE4

Interacting With Stylus Programs Using Ethers-rs

An example is included in this repo underrust-example which uses the popularethers-rs library to interact with our prime sieve contract on the Stylus testnet. To run it, do:

export STYLUS_PROGRAM_ADDRESS=0x0c503Bb757b1CaaD0140e8a2700333C0C9962FE4 cargo run ...and see:

Checking if 2 is_prime = true, took: 404.146917ms Checking if 3 is_prime = true, took: 154.802083ms Checking if 4 is_prime = false, took: 123.239583ms Checking if 5 is_prime = true, took: 109.248709ms Checking if 6 is_prime = false, took: 113.086625ms Checking if 32 is_prime = false, took: 280.19975ms Checking if 53 is_prime = true, took: 123.667958ms

Next steps

The hostios defined in this walkthrough are not the only ones! Check out ourstylus-sdk-c to see all the hostios you can use underhostio.h . These include affordances for the EVM, utilities to access storage, and utilities to call other Arbitrum smart contracts. Edit this page Last updatedonMar 19, 2024 Previous Troubleshooting Stylus Next How to reduce the size of Stylus WASM binaries