BUILD WITH STYLUS

A PRACTICAL GUIDE TO GETTING STARTED

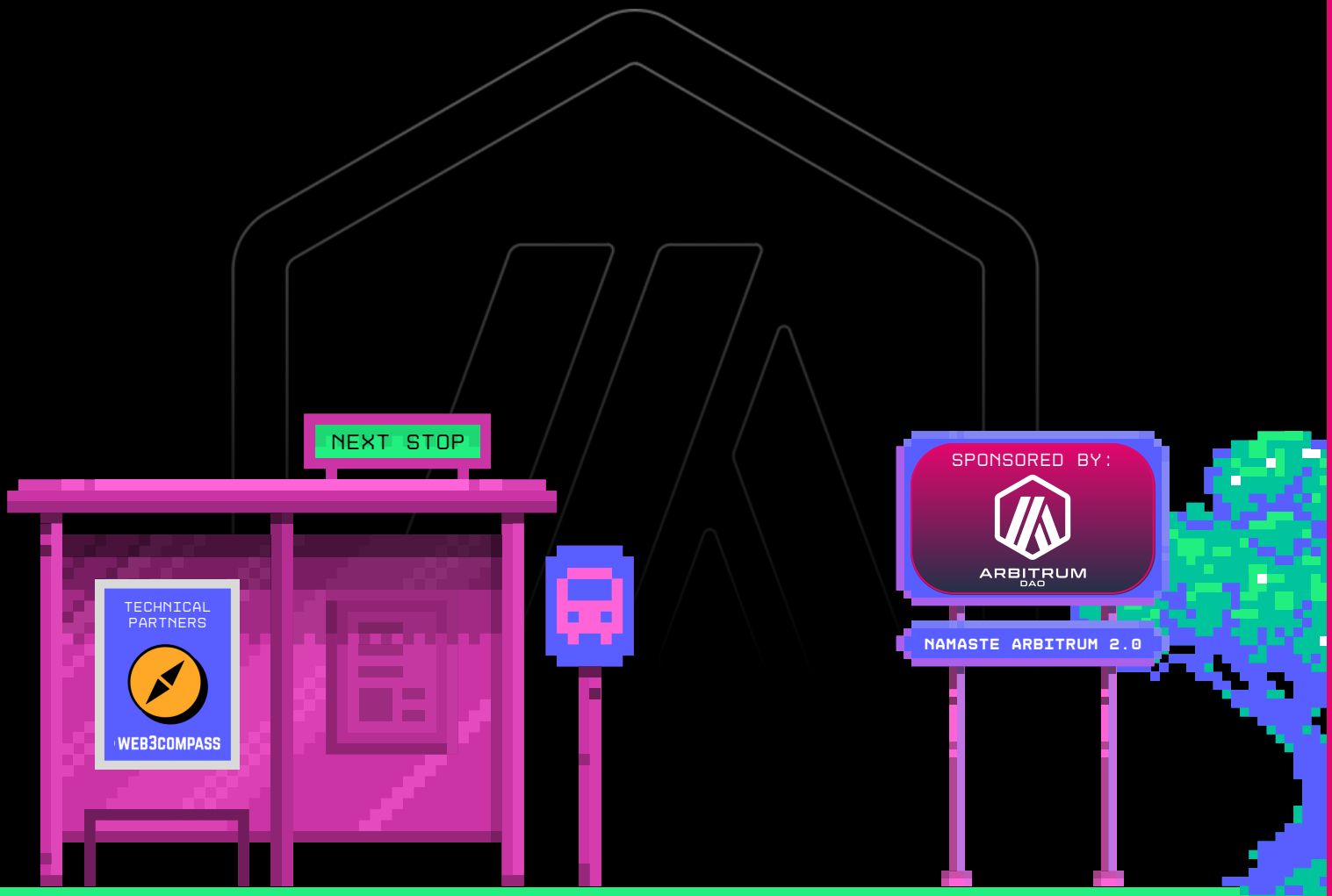


Table of Contents

Introduction

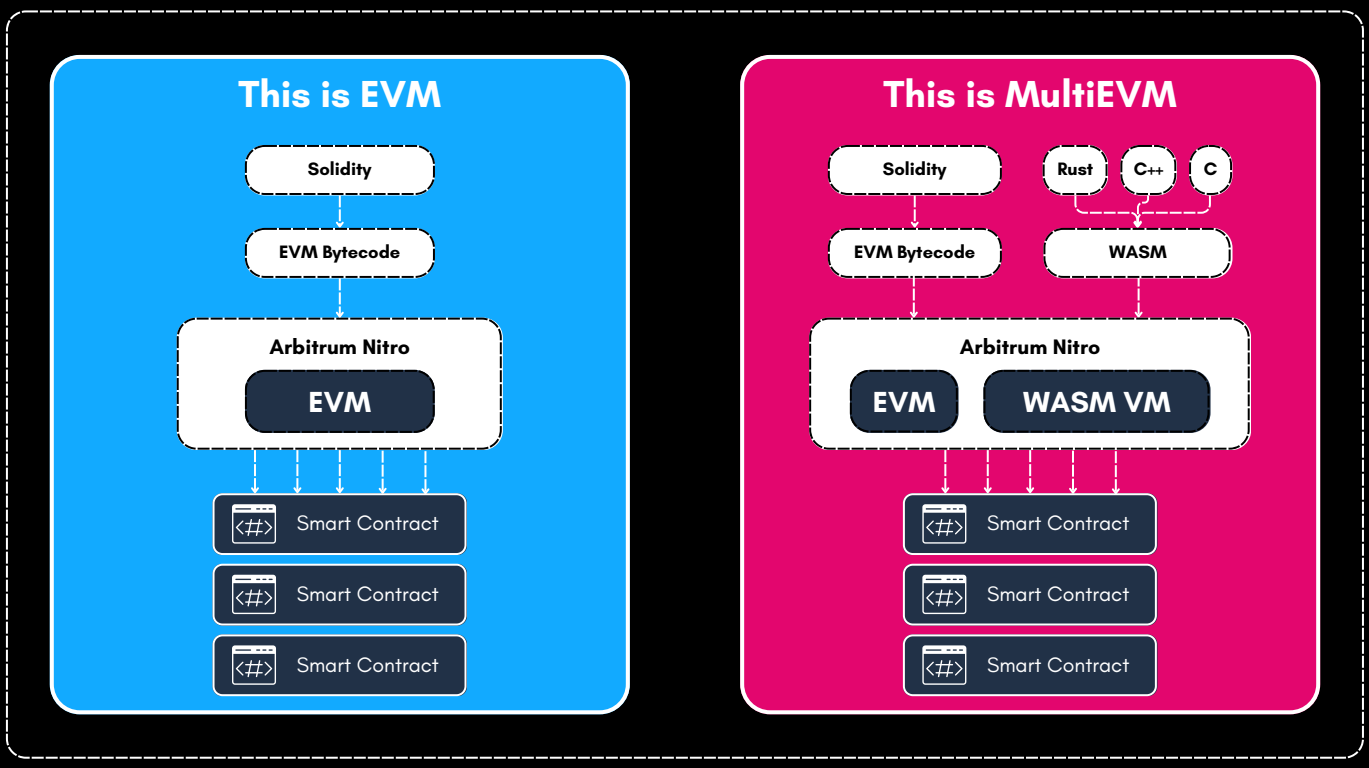
Stylus lets you write smart contracts in programming languages that compile to WASM, such as Rust, C, C++, and many others allowing you to tap into their ecosystem of libraries and tools. Rich language and tooling support already exist for Rust.

Solidity contracts and Stylus contracts are fully interoperable. In Solidity, you can call a Rust program and vice versa, thanks to a second, coequal WASM virtual machine.

Stylus contracts offer significantly faster execution and lower gas fees for memory- and compute-intensive operations, thanks to the superior efficiency of WASM programs.

What's Stylus?

Stylus is an upgrade to Arbitrum Nitro (ArbOS 32), the tech stack powering Arbitrum One, Arbitrum Nova, and Arbitrum chains. This upgrade adds a second, coequal virtual machine to the EVM, where EVM contracts continue to behave exactly as they would in Ethereum. We call this paradigm MultiVM, since everything is entirely additive.



Stylus gives you MultiVM

This second virtual machine executes WebAssembly (WASM) rather than EVM bytecode. WASM is a modern binary format popularized by its use in major web standards, browsers, and companies to speed up computation. WASM is built to be fast, portable, and human-readable. It has sandboxed execution environments for security and simplicity.

Working with WASM is nothing new for Arbitrum chains. Ever since the Nitro upgrade, WASM has been a fundamental component of Arbitrum's fully functioning fraud proofs.

With a WASM VM, any programming language compilable to WASM is within Stylus's scope. While many popular programming languages can compile into WASM, some compilers are more suitable for smart contract development than others — like Rust, C, and C++. Other languages like Go, Sway, Move, and Cairo are also supported.

Languages that include their own runtimes, like Python and Javascript, are more complex for Stylus to support, although not impossible.

Compared to Solidity, WASM programs are much more efficient for memory-intensive applications. There are many reasons for this, including the decades of compiler development for Rust and C. WASM also has a faster runtime than the EVM, resulting in faster execution.

Use Cases

While many developers will be drawn to new use cases, rebuilding existing applications in Stylus will also open the door to innovation and optimization. dApps have never been faster, cheaper, or safer.

Stylus can integrate easily into existing Solidity projects by calling a Stylus contract to optimize specific parts of your dApp or building the entire dApp with Stylus.

It’s impossible to list all of the use cases Stylus enables; think about the properties of all WASM-compatible languages! That said, here are some particularly exciting ideas:

Use Cases

1. Efficient Onchain Verification with ZK-Proofs:	Enable cost-effective onchain verification using zero-knowledge proving systems for privacy, interoperability, and more (see case study).
2. Advanced DeFi Instruments:	Power complex financial instruments and processes like custom pricing curves for AMMs, synthetic assets, options, and futures with onchain computation — via extending current protocols (i.e., Uniswap V4 hooks) or building your own.
3. High-Performance Onchain Logic:	Support memory- and compute-intensive applications like onchain games and generative art — either by writing all of the application in Stylus or enhancing performance of existing Solidity contracts by optimizing specific parts.
4. Endless Possibilities:	Enable innovative use cases such as generative art, compute-heavy AI models, onchain games, and projects utilizing advanced cryptography — unlocking the full potential of resource-intensive applications onchain.

Build With Stylus : Go SDK

Welcome to another installment of Build with Stylus! And today we're stepping into uncharted territory: writing smart contracts in Go on Arbitrum Stylus.
Yes, Go — the language of Kubernetes, Docker, and modern backend infra — is making its first steps into the smart contract world, thanks to Stylus and an experimental SDK.
Let's dive in.

Stylus: A Recap

With the Nitro upgrade, Arbitrum switched to a WebAssembly (WASM) execution environment. That paved the way for Stylus, which allows smart contracts to be written in:

- Rust
- C / C++
- Go (via TinyGo)

This unlocks powerful new workflows and lets builders use familiar tools, better memory control, and massive performance improvements over Solidity.

Why Go?

Go wasn't originally built for WebAssembly. It has:

- A garbage collector
- A full runtime
- Rich abstractions like goroutines and schedulers

Great for web servers, not so great for deterministic WASM execution.
But thanks to TinyGo — a slimmed-down Go compiler for microcontrollers and WASM — we now have a path.

Enter TinyGo

TinyGo compiles Go to WASM by stripping away much of the runtime. This makes it small and predictable enough to run in Stylus.

- TinyGo GitHub: https://github.com/tinygo-org/tinygo
- Installation: https://tinygo.org/getting-started/

```
# Example install on macOS
brew tap tinygo-org/tools
brew install tinygo
```

The Go SDK (Proof of Concept)

A community-built SDK gives us a glimpse of what Go-based smart contracts might look like:

- Repo: https://github.com/af-afk/stylus
- Codegen Tool: https://github.com/af-afk/stylus/blob/trunk/cmd/stylus-go/main.go

Sample Contract (Counter)

```
package main

import (
    "math/big"
    "github.com/af-afk/stylus"
)

//stylus entrypoint
type Storage struct {
    Counter stylus.StorageUint256
}

func (s Storage) Add(x *big.Int) ([]byte, error) {
    y := s.Counter.Get()
    y.Add(y, x)
    s.Counter.Set(y)
    var b [32]byte
    x.FillBytes(b[:])
    return b[:], nil
}
```

Codegen Strategy

- Uses comments like //stylus entrypoint to identify the main struct
- Uses //stylus uint256 to define argument types
- Generates an entrypoint from Go AST

> *Similar in spirit to gqlgen, the codegen tool parses Go files and emits WASM-compatible function wrappers.*

Gotchas and Limitations

- Not ready for production (no deployment path yet)
- Lacks ABI support and official docs
- Go runtime support is limited (e.g. no goroutines)
- Reflection doesn't work in TinyGo

But the proof-of-concept works, and it shows Go contracts can eventually:

- Manage storage
- Encode data
- Return bytes + handle errors

A Glimpse of the Future

The Stylus community is already buzzing:

- Three major audited dApps are in progress using this model
- A gas optimization competition is planned
- A Stylus Developer DAO is forming

> *Join the unofficial Stylus Dev Discord: https://discord.gg/eTRt3r3F*

This whole Go journey is still experimental. But Stylus is proving that:

- WebAssembly opens doors to multiple languages
- Go developers are not locked out of Web3
- Smart contracts don't need to be Solidity-only anymore

Resources

- Stylus Go SDK (POC): https://github.com/af-afk/stylus
- TinyGo Compiler: https://tinygo.org/
- Arbitrum Stylus Docs: https://docs.arbitrum.io/stylus
- Stylus Community Blog: https://0xys.substack.com/

Go on Stylus is still rough around the edges, but it's a spark.
It shows us that the future of onchain programming isn't limited to Solidity, or even Rust. With the right tooling, any modern language can find its way to the blockchain.
Let's build.

Build with Stylus: Go Edition

Ep#1 Intro to Ethereum & Arbitrum

In this episode, we set the stage by diving deep into the journey of Ethereum, the limitations it faced, and how Layer2 solutions like Arbitrum stepped in to scale the ecosystem without compromising on decentralization.

Ep#2 The Birth of Stylus: Arbitrum's Leap Beyond EVM Limits

In this episode, we go beyond Ethereum's limits and explore what makes Stylus on Arbitrum such a game-changer for C, Rust and other developers.

Ep#3 How Stylus Works Under the Hood on Arbitrum

In this episode, we unpack how Stylus compiles, executes, and seamlessly interoperates with Solidity on Arbitrum. Dive into the engine room of Stylus and see how it runs C, Go, and Rust natively on Arbitrum.

Ep#4 Stylus Go SDK: Exploring Smart Contracts in Go on Stylus

In the final episode of Build with Stylus (Go Edition), we explore the experimental Go SDK for Arbitrum Stylus, a unique blend of theory and hands-on tinkering.

Build with Stylus: C/C++ SDK

Welcome to the Build with Stylus series — where we explore how to take the languages you already know, like C and C++ and bring them onchain using Arbitrum Stylus.
This guide is your companion to the video walkthrough. We'll cover everything from:

- What Stylus is and why it matters
- How C/C++ compiles to WebAssembly (WASM)
- What's inside the SDK
- How to build real contracts using familiar toolchains
- And how to get started locally — today

Let's go.

Why Stylus?

Let's start with the big picture.
Traditionally, if you wanted to build on Ethereum, you had to use Solidity — a domain-specific language purpose-built for the Ethereum Virtual Machine (EVM). It works — but it's also limited. Performance bottlenecks, gas inefficiency, weird syntax quirks — you know the drill.
Then came Arbitrum Nitro — a major upgrade that rebuilt Arbitrum's execution layer around WebAssembly (WASM), a modern binary format used across browsers, blockchains, and beyond.
This upgrade unlocked a new environment called Stylus.

What is Stylus?

Stylus is a WASM-based smart contract runtime on Arbitrum. It allows developers to write smart contracts in popular languages like:

- Rust
- C and C++
- (and eventually more)

These contracts are compiled into WASM binaries, which are then executed side by side with Solidity contracts — sharing the same storage, the same chain, but using a faster and cheaper engine.

> *Stylus doesn't replace Solidity — it extends what's possible. You can mix and match: write high-performance logic in C, and your interface in Solidity. It's all interoperable.*

Why C and C++?

If you're a systems programmer, embedded dev, or game developer, you probably already speak C or C++. But even beyond familiarity, these languages bring unique strengths:

- Low-level memory control
- High execution speed
- Tons of audited libraries you can repurpose
- Ideal for compute-heavy use cases like cryptography, compression, and encoding

Stylus makes it possible to bring that power onchain and that's huge.

The Stylus C SDK

- **Repo: github.com/OffchainLabs/stylus-sdk-c**

This SDK is your sandbox. It's not polished or production-ready yet — but it works, and it proves what's possible.

Here's what you'll find inside:

- A minimal set of C headers for interacting with Stylus
- Utilities for:
  - Defining entry points
  - Reading/writing storage
  - Managing memory
- A few replacements for standard libraries like stdlib.h and string.h
- Two working examples:
  - A SipHash hasher
  - A bare-metal ERC-20 token

A Quick Peek at the Examples

1. SipHash Contract

This is a pure compute contract. It takes two inputs, runs a SipHash algorithm, and returns a digest. That's it.

- No storage
- No logs
- No Solidity ABI required
- Just raw math at blazing speed

Why it's cool:

The `.wasm` output is only 609 bytes — and hashing a 32-byte input costs just 22 gas. That's unheard of in Solidity.

```
cd examples/siphash
make
```

This gives you `siphash.wasm` ready to deploy.

2. ERC-20 in C

This is where things get wild.

This contract manually implements an ERC-20 token — no OpenZeppelin, no shortcuts.

- Calculates its own storage slots
- Tracks balances and allowances
- Does all memory and byte handling from scratch

This is "bare-metal" smart contract development. Not recommended for your production launch... but incredibly educational.

```
cd ../erc20
make
```

You'll end up with `erc20.wasm` — a fully functioning token written in C.

Setting Up Your Environment

Now, let's talk about the tools you need to make this all work.

1. LLVM — The Compiler Backbone

LLVM (Low Level Virtual Machine) is an open-source compiler infrastructure that helps you convert C/C++ code into all kinds of formats — including WebAssembly.

You'll be using the `clang` compiler and `wasm-ld` linker from the LLVM suite.

You need:

- LLVM version 14
- `clang` must support `-target=wasm32`
- wasm-ld must be available for linking .wasm

- macOS:

```
brew install llvm@14
```

- Ubuntu (recommended via PPA or source):

```
sudo apt install llvm-14 clang-14 lld-14
```

> *Run clang --version to confirm you're on version 14.*

2. WebAssembly Binary Toolkit (WABT)

This is a set of tools to inspect, debug, and convert WASM files. Super useful for seeing what's going on under the hood.

Key tools:

- `wasm-objdump` : Inspect your compiled `.wasm` for functions and memory layout
- `wasm2wat` : Convert WASM binary to human-readable WAT format
- `wat2wasm` : Convert back from WAT to WASM

Example:

```
wasm-objdump -x siphash.wasm wasm2wat
siphash.wasm -o siphash.wat
```

3. cargo stylus — Your Deployment Companion

Even though we're writing C/C++, Stylus tooling is still part of the Rust ecosystem.

You'll use cargo-stylus to:

- Generate Stylus project scaffolds
- Deploy .wasm contracts
- Simulate and test them locally

Install with:

```
cargo install --git <https://github.com/OffchainLabs/stylus> stylus
```

This tool will handle deployment to local or test networks when you're ready.

> *Full docs: Stylus GitHub*

What to Expect (and What Not To)

What Works:

- Writing real smart contracts in C
- Interacting with blockchain state
- Blazing-fast compute logic
- Full WASM support

What's Missing:

- No automatic ABI decoding — you'll do this manually via pointer math
- No logging or events yet
- Documentation is minimal — treat the examples as your guide
- No polished tooling — but that's improving fast

This is early-stage territory. But if you're comfortable with memory layouts, bytes, and stack traces — you'll feel right at home.

Final Thoughts

If you're a C/C++ dev who's been curious about Web3 but didn't want to mess with Solidity — Stylus is your door in.

And if you're a Solidity dev? This isn't a replacement. It's an upgrade. Stylus lets you move your heavy compute logic into lean, hyper-efficient WASM programs — while keeping everything interoperable.

So yes, this is still experimental.

But it's real. It works. And it's going to keep getting better.

Useful Links

- Stylus C SDK Repo: github.com/OffchainLabs/stylus-sdk-c
- Stylus CLI / cargo-stylus: github.com/OffchainLabs/stylus
- Stylus Official Docs: docs.arbitrum.io/stylus
- WebAssembly Binary Toolkit (WABT): github.com/WebAssembly/wabt
- LLVM: https://llvm.org/

Build with Stylus: C/C++ Edition

Ep#1 Intro to Ethereum & Arbitrum

In this episode, we set the stage by diving deep into the journey of Ethereum, the limitations it faced, and how Layer2 solutions like Arbitrum stepped in to scale the ecosystem without compromising on decentralization.

Ep#2 The Birth of Stylus: Arbitrum's Leap Beyond EVM Limits

In this episode, we go beyond Ethereum's limits and explore what makes Stylus on Arbitrum such a game-changer for C, Rust and other developers.

Ep#3 How Stylus Works Under the Hood on Arbitrum

In this episode, we unpack how Stylus compiles, executes, and seamlessly interoperates with Solidity on Arbitrum. Dive into the engine room of Stylus and see how it runs C, Go, and Rust natively on Arbitrum.

Ep#4 Running C & C++ on Arbitrum Stylus: What's Possible Today

In this episode, we explore how C and C++ smart contracts are becoming possible on Arbitrum through Stylus and why that matters.

Ep#5 Getting Real with Stylus C SDK: A Builder's Walkthrough

In the final episode of Build with Stylus (C/C++ Edition), we roll up our sleeves and dive into the actual Stylus C/C++ SDK created by Offchain Labs to see what it can (and can't) do today.

Build with Stylus: Rust Edition

Github Repo of the project: ArbitrumStylus_RUST

Step 1: Set Up Your Development Environment

Prerequisites:

- Install the Rust toolchain (v1.81 or newer)
  Go to: https://www.rust-lang.org/tools/install
  After installing, verify:

```
rustup --version
rustc --version
cargo --version
```

- Install Docker
  Required for running the Nitro devnode and Stylus checks.
  Download from: https://www.docker.comInstall Foundry's Cast CLI

```
curl -L <https://foundry.paradigm.xyz> | bash foundryup
```

- Recommended IDE: VS Code
  Helpful extensions for Rust development:
  rust-analyzer - smart completion, diagnostics
  Error Lens - highlights errors inline
  Even Better TOML - better syntax for
  Cargo.toml - Manage Rust crates in the editor

Step 2: Run the Nitro Devnode (Local Arbitrum Chain)

Stylus contracts run on Arbitrum — you'll use a local devnode with a pre-funded test wallet.

```
git clone <https://github.com/OffchainLabs/nitro-devnode.git>
cd nitro-devnode
./run-dev-node.sh
```

Ensure Docker is running before executing the above.
This will start a local chain at <http://localhost:8547>.

Step 3: Install and Set Up Cargo Stylus

Install the CLI:

```
cargo install --force cargo-stylus
```

Set Rust toolchain and WASM target:

```
rustup default 1.80
rustup target add wasm32-unknown-unknown --toolchain 1.80
rustup target add wasm32-unknown-unknown (in case there is an error)
```

Confirm installation:

```
cargo stylus --help
```

This shows available commands like:
new , check , deploy, verify, trace , etc.

Step 4: Create a Stylus Project

```
cargo stylus new my-counter
cd my-counter

#This generates a Rust-based implementation of a basic Solidity Counter contract:
#The Rust version will be inside your new project folder — ready to customize.

Step 5: Check If the Contract Is Valid
#Ensure Docker is running, then validate the contract with:
cargo stylus check
```

```
Step 6: Estimate Gas for Deployment

Use the pre-funded dev wallet provided by the Nitro devnode:
cargo stylus deploy --endpoint='<http://localhost:8547>' \\
--private-key="0xb6b15c8cb491557369f3c7d2c287b053eb229daa9c22138887752191c9520659" \\
--estimate-gas

#Output
You should see something like:
deployment tx gas: 7123737
gas price: "0.100000000" gwei
deployment tx total cost: "0.000712373700000000" ETH
```

Step 7: Deploy the Contract

Run the deployment (this includes deployment + activation):
*cargo stylus deploy --endpoint='<http://localhost:8547>' *
*--private-key="0xb6b15c8cb491557369f3c7d2c2c87b053eb229daa9c22138887752191c9520659" *

#On success:
deployed code at address:
0x33f54de59419570a9442e788f5dd5cf635b3c7ac
deployment tx hash:
0xa55efc05c45efc63647dff5cc37ad328a47ba5555009d92ad4e297bf4864de36

#wasm already activated!Save that address — you'll need it to interact with your contract.

This guide walks you through building a complete, production-grade ERC-20 token smart contract in Rust using Arbitrum Stylus, a WASM-based contract engine that lets you write smart contracts in languages beyond Solidity.
You'll learn:

- How to scaffold a Stylus Rust project
- How to write a modular ERC-20 smart contract in Rust
- How to define your own token (CandyToken)
- How to use Stylus primitives and tooling
- How to prepare for deployment & frontend integration

Tools & Setup

Prerequisites

Make sure you have the following set up:

Tool	Purpose
rustup	Manages the Rust toolchain
cargo	Rust's package manager & build tool
Node.js + Yarn/NPM	For frontend setup (optional later)
Stylus CLI	Scaffolding & compiling Stylus apps

Install the Stylus CLI

```
cargo install stylus-cli --locked
```

Stylus CLI helps you quickly spin up new Stylus-compatible smart contract projects.

Create a New Stylus Project

```
cargo stylus new mini-token-dapp
cd mini-token-dapp
```

This generates:

```
css
CopyEdit
mini-token-dapp/
├── Cargo.toml
├── rust-toolchain.toml
├── src/
│   ├── main.rs
│   └── lib.rs
├── examples/
│   └── counter.rs
├── README.md
├── header.png
```

- `Cargo.toml` — your project manifest
- `main.rs` — mostly boilerplate; used for testing or deployments
- `lib.rs` — your contract entry point
- `erc20.rs` — we'll add this to house reusable logic
- `counter.rs` — a simple example you can delete

Project Structure Overview

We'll be splitting logic into two parts:

File	Purpose
`erc20.rs`	Reusable, generic ERC-20 implementation (mint, burn, transfer)
`lib.rs`	Your token config + entrypoint (CandyToken)

Contract Breakdown

`src/erc20.rs` : Generic ERC-20 Logic

This file is a reusable, type-safe ERC-20 engine in Rust.

Imports

```
extern crate alloc;
use alloc::string::String;
use alloy_primitives::{Address, U256};
use alloy_sol_types::sol;
use core::marker::PhantomData;
use stylus_sdk::{prelude::*, stylus_core::log};
```

- `alloc` is required for WASM (no std)
- `Address` and `U256` come from Alloy (Ethereum-native types)
- `PhantomData` Let us use type-based configuration safely
- `Stylus SDK` powers contract interaction and bindings

ERC-20 Config Trait

```
pub trait Erc20Params {
    const NAME: &'static str;
    const SYMBOL: &'static str;
    const DECIMALS: u8;
}
```

Tokens implement this trait to define their identity (name, symbol, decimals). This lets us reuse the contract logic for any ERC-20.

Storage Layout

```
sol_storage! {
    pub struct Erc20<T> {
        mapping(address => uint256) balances;
        mapping(address => mapping(address => uint256)) allowances;
        uint256 total_supply;
        PhantomData<T> phantom;
    }
}
```

- Modeled after Solidity's `mapping`
- Fully type-safe, with the `sol_storage!` macro provided by Stylus SDK
- `PhantomData<T>` connects the storage type to the trait config

Events & Errors

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

    error InsufficientBalance(address from, uint256 have, uint256 want);
    error InsufficientAllowance(address owner, address spender, uint256 have, uint256 want);
}
```

This mirrors the standard ERC-20 interface, along with useful custom errors that provide more visibility than generic reverts.

Internal Logic

```
impl<T: Erc20Params> Erc20<T> {
    pub fn _transfer(...) -> Result<(), Erc20Error> { ... }
    pub fn mint(...) -> Result<(), Erc20Error> { ... }
    pub fn burn(...) -> Result<(), Erc20Error> { ... }
}
```

Safe, auditable core logic for state changes — no `require` , just `Result`

Public Methods (ERC-20 Interface)

```
#[public]
impl<T: Erc20Params> Erc20<T> {
    pub fn name() -> String { ... }
    pub fn symbol() -> String { ... }
    pub fn decimals() -> u8 { ... }

    pub fn total_supply(&self) -> U256 { ... }
    pub fn balance_of(&self, owner: Address) -> U256 { ... }

    pub fn transfer(...) -> Result<bool, Erc20Error> { ... }
    pub fn transfer_from(...) -> Result<bool, Erc20Error> { ... }
    pub fn approve(...) -> bool { ... }
    pub fn allowance(...) -> U256 { ... }
}
```

These are the familiar ERC-20 functions exposed for external calls.

`src/lib.rs` : CandyToken Definition

This is where we create a real token using the generic logic.

Imports & Module Linking

```
extern crate alloc;
mod erc20;
use alloy_primitives::{Address, U256};
use stylus_sdk::prelude::*;
use crate::erc20::{Erc20, Erc20Params, Erc20Error};
```

Define Token Metadata

```
struct CandyTokenParams;
impl Erc20Params for CandyTokenParams {
    const NAME: &'static str = "CandyToken";
    const SYMBOL: &'static str = "CANDY";
    const DECIMALS: u8 = 18;
}
```

Contract Storage & Entrypoint

```
sol_storage! {
    #[entrypoint]
    struct CandyToken {
        #[borrow]
        Erc20<CandyTokenParams> erc20;
    }
}
```

- `#[entrypoint]` : Stylus will compile this struct into the contract entry point
- `#[borrow]` : Enables `erc20` to modify the contract's storage directly

Public Functions

```
#[public]
#[inherit(Erc20<CandyTokenParams>)]
impl CandyToken {
    pub fn mint(...) -> Result<(), Erc20Error> { ... }
    pub fn mint_to(...) -> Result<(), Erc20Error> { ... }
    pub fn burn(...) -> Result<(), Erc20Error> { ... }
}
```

- `#[inherit(...)]` : Automatically exposes all ERC-20 methods
- These methods add mint/burn access for wallets or admins

Helpful References

Resource	Link
Arbitrum Stylus Docs	https://docs.arbitrum.io/stylus
Stylus Example Contracts	https://github.com/OffchainLabs/stylus-examples
Stylus SDK Crate	https://crates.io/crates/stylus-sdk
Stylus CLI	https://github.com/OffchainLabs/stylus-cli
Alloy Primitives	https://docs.rs/alloy_primitives/latest/alloy_primitives/
Arbitrum Dev Discord	https://discord.gg/arbitrum

Next Steps

After writing your contract:

1. Build your project:

```
cargo stylus build
```

1. Deploy on a local devnet or testnet using Stylus deploy tools
2. Write a frontend to call methods like `mint` , `transfer` , `burn`
3. Extend functionality:
   - Access control (onlyOwner)
   - Pausable transfers
   - Metadata extensions

Recap: Why Stylus + Rust?

- **Performance**: WASM execution is fast, scalable, and cheaper than EVM
- **Tooling**: Use familiar Rust tools like `cargo` , `clippy` , `rust-analyzer`
- **Safety**: Strong type system, no silent `reverts`
- **Interoperability**: Direct access to EVM-compatible types via Alloy

Helpful Resources to Learn Rust Language

- https://doc.rust-lang.org/beta/book/
- https://github.com/rust-lang/rustlings
- https://www.rust-lang.org/learn

Build with Stylus: Rust Edition

Ep#1 Intro to Ethereum & Arbitrum

In this episode, we set the stage by diving deep into the journey of Ethereum, the limitations it faced, and how Layer2 solutions like Arbitrum stepped in to scale the ecosystem without compromising on decentralization.

Ep#2 The Birth of Stylus: Arbitrum's Leap Beyond EVM Limits

In this episode, we go beyond Ethereum's limits and explore what makes Stylus on Arbitrum such a game-changer for C, Rust and other developers.

Ep#3 How Stylus Works Under the Hood on Arbitrum

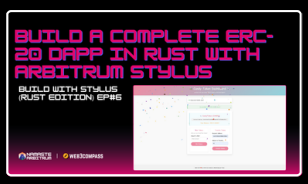
In this episode, we unpack how Stylus compiles, executes, and seamlessly interoperates with Solidity on Arbitrum. Dive into the engine room of Stylus and see how it runs C, Go, and Rust natively on Arbitrum.

Ep#4 Deploy Your First Stylus Contract in Rust: Full Setup Tutorial

In this episode of Build with Stylus (Rust Edition), we go hands-on and walk you through setting up your full development environment to start building with Stylus on Arbitrum.

Ep#5 Build an ERC-20 Token in Rust with Arbitrum Stylus

Complete backend implementation of a Candy Token, from setting up your development structure to understanding Rust-based ERC20 logic like minting, transferring, allowances and more.



Ep#6 Build a Complete ERC-20 DApp in Rust with Arbitrum Stylus

In the final episode of Build with Stylus (Rust Edition), we bring everything together from smart contract to candy-themed frontend!

Arbitrum + Stylus Resources

1. Awesome Stylus:

- https://github.com/OffchainLabs/awesome-stylus
- Repository for various community-contributed Stylus projects and tools

2. Quikstart

- https://docs.arbitrum.io/stylus/quickstart

3. Rust SDK

- https://docs.arbitrum.io/stylus/reference/overview

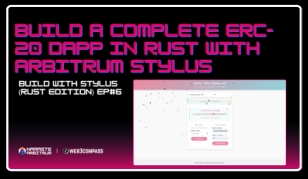
4. Stylus Saturdays

- https://stylus-saturdays.com/
- Stay updated with the latest from the Stylus community through tutorials, builder interviews, technical deep dives, and more with the Stylus Saturdays newsletter.

5. Arbitrum Docs

- https://docs.arbitrum.io/welcome/arbitrum-gentle-introduction

Build with Stylus Tutorials



Build with Stylus - Rust Edition

In this 6-part series, you'll go end-to-end on building a DApp with Arbitrum Stylus + Rust.
Write an ERC-20 contract, deploy it, build a React frontend, and bring your Candy token to life—mint, transfer, and celebrate in style.

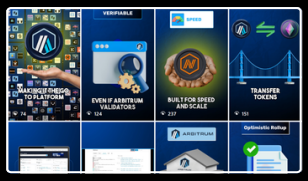
Build with Stylus - C & C++ Edition

A hands-on dive into the Stylus C/C++ SDK by Offchain Labs. Learn how to set up your environment, compile C++ to WebAssembly, explore example contracts, and debug your way through the wild world of Stylus dev. This 5-part series is for builders who want to push the limits of Stylus using C/C++.

Build with Stylus - Go Edition

Explore what it takes to write smart contracts in Go for Arbitrum Stylus. From TinyGo setup to sample contracts, we walk through the quirks, workarounds, and current limitations of building with Go on Stylus. This 4-part series explores the experimental frontier of writing smart contracts in Go for Arbitrum Stylus.

Arbitrum in 60 Seconds



Arbitrum in 60 Seconds

Everything you need to know about Arbitrum, in under a minute. Quick, clear, and to the point, Arbitrum in 60 Seconds breaks down key concepts, tools, and tech (including Stylus) in under a minute. Perfect for builders, learners, and anyone curious about Ethereum's leading L2.

Namaste Arbitrum: A 10-Part Regional Series

Namaste Arbitrum: 10-Part Series (English Edition)

A beginner-friendly 10-part video series in English covering everything from Ethereum and Arbitrum basics to scaling solutions, wallet setup, and governance participation.

Namaste Arbitrum: 10-Part Series (Hindi Edition)

A beginner-friendly 10-part video series in Hindi covering everything from Ethereum and Arbitrum basics to scaling solutions, wallet setup, and governance participation.

Namaste Arbitrum: 10-Part Series (Bengali Edition)

A beginner-friendly 10-part video series in Bengali covering everything from Ethereum and Arbitrum basics to scaling solutions, wallet setup, and governance participation.

Namaste Arbitrum: 10-Part Series (Tamil Edition)

A beginner-friendly 10-part video series in Tamil covering everything from Ethereum and Arbitrum basics to scaling solutions, wallet setup, and governance participation.

Namaste Arbitrum: 10-Part Series (Telugu Edition)

A beginner-friendly 10-part video series in Telugu covering everything from Ethereum and Arbitrum basics to scaling solutions, wallet setup, and governance participation.

Namaste Arbitrum: 10-Part Series (Gujarati Edition)

A beginner-friendly 10-part video series in Gujarati covering everything from Ethereum and Arbitrum basics to scaling solutions, wallet setup, and governance participation.

About Us

About Us

This Practical Guide is brought to you by **Namaste Arbitrum**, a grassroots initiative dedicated to growing the Arbitrum builder ecosystem in India through hands-on education, technical content, and community-driven projects.

Created in collaboration with **Web3 Compass**, a platform designed by builders for builders, this guide is part of our shared mission to help developers explore and master Web3 and AI through practical, challenge-based learning.

For more tutorials, ecosystem updates, and dev resources, follow Namaste Arbitrum and Web3 Compass.

Let's build with Stylus.

Follow Namaste Arbitrum

- X (Twitter): https://x.com/NamasteArbitrum
- Instagram: https://www.instagram.com/pyorxyz/
- LinkedIn: https://www.linkedin.com/company/pyorxyz
- YouTube: https://www.youtube.com/@pyorxyz

Follow Web3 Compass

X (Twitter): https://x.com/the_web3compass
LinkedIn: https://www.linkedin.com/company/the-web3compass
Telegram: https://t.me/+Bmec234RB3M3YTll
YouTube: https://www.youtube.com/@TheWeb3Compass

Join the Build with Stylus WhatsApp Group for More Dev Content