Developing with Rust

Solana supports writing on-chain programs using theRust programming language.

Project Layout#

Solana Rust programs follow the typicalRust project layout :

/inc/ /src/ /Cargo.toml Solana Rust programs may depend directly on each other in order to gain access to instruction helpers when makingcross-program invocations . When doing so it's important to not pull in the dependent program's entrypoint symbols because they may conflict with the program's own. To avoid this, programs should define anno-entrypoint feature inCargo.toml and use to exclude the entrypoint.

- Define the feature
- Exclude the entrypoint

Then when other programs include this program as a dependency, they should do so using theno-entrypoint feature.

- Include without entrypoint

Project Dependencies#

At a minimum, Solana Rust programs must pull in thesolana-program crate.

Solana SBF programs have somerestrictions that may prevent the inclusion of some crates as dependencies or require special handling.

For example:

- Crates that require the architecture be a subset of the ones supported by the
- official toolchain. There is no workaround for this unless that crate is
- forked and SBF added to that those architecture checks.
- Crates may depend onrand
- which is not supported in Solana's deterministic
- program environment. To include arand
- dependent crate refer toDepending on Rand
- .
- Crates may overflow the stack even if the stack overflowing code isn't
- included in the program itself. For more information refer toStack
- .

How to Build#

First setup the environment:

- Install the latest Rust stable fromhttps://rustup.rs/
- Install the latestSolana command-line tools

The normal cargo build is available for building programs against your host machine which can be used for unit testing:

cargo build To build a specific program, such as SPL Token, for the Solana SBF target which can be deployed to the cluster:

cd cargo build-bpf

How to Test#

Solana programs can be unit tested via the traditionalcargo test mechanism by exercising program functions directly.

To help facilitate testing in an environment that more closely matches a live cluster, developers can use theprogram-test crate. Theprogram-test crate starts up a local instance of the runtime and allows tests to send multiple transactions while keeping state for the duration of the test.

For more information thetest in sysvar example shows how an instruction containing sysvar account is sent and processed by the program.

Program Entrypoint#

Programs export a known entrypoint symbol which the Solana runtime looks up and calls when invoking a program. Solana supports multiple versions of the BPF loader and the entrypoints may vary between them. Programs must be written for and deployed to the same loader. For more details see theFAQ section on Loaders .

Currently there are two supported loadersBPF Loader andBPF loader deprecated

They both have the same raw entrypoint definition, the following is the raw symbol that the runtime looks up and calls:

[no_mangle]

pub unsafe extern "C" fn entrypoint(input: *mut u8) -> u64; This entrypoint takes a generic byte array which contains the serialized program parameters (program id, accounts, instruction data, etc...). To deserialize the parameters each loader contains its own wrapper macro that exports the raw entrypoint, deserializes the parameters, calls a user defined instruction processing function, and returns the results.

You can find the entrypoint macros here:

- BPF Loader's entrypoint macro
- BPF Loader deprecated's entrypoint macro

The program defined instruction processing function that the entrypoint macros call must be of this form:

pub type ProcessInstruction = fn(program_id: &Pubkey, accounts: &[AccountInfo], instruction_data: &[u8]) -> ProgramResult;

Parameter Deserialization#

Each loader provides a helper function that deserializes the program's input parameters into Rust types. The entrypoint macros automatically calls the deserialization helper:

- BPF Loader deserialization
- BPF Loader deprecated deserialization

Some programs may want to perform deserialization themselves and they can by providing their own implementation of theraw entrypoint . Take note that the provided deserialization functions retain references back to the serialized byte array for variables that the program is allowed to modify (lamports, account data). The reason for this is that upon return the loader will read those modifications so they may be committed. If a program implements their own deserialization function they need to ensure that any modifications the program wishes to commit be written back into the input byte array.

Details on how the loader serializes the program inputs can be found in theInput Parameter Serialization docs.

Data Types#

The loader's entrypoint macros call the program defined instruction processor function with the following parameters:

program_id: &Pubkey, accounts: &[AccountInfo], instruction_data: &[u8] The program id is the public key of the currently executing program.

The accounts is an ordered slice of the accounts referenced by the instruction and represented as anAccountInfo structures. An account's place in the array signifies its meaning, for example, when transferring lamports an instruction may define the first account as the source and the second as the destination.

The members of theAccountInfo structure are read-only except forlamports anddata . Both may be modified by the program in accordance with theruntime enforcement policy . Both of these members are protected by the RustRefCell construct, so they must be borrowed to read or write to them. The reason for this is they both point back to the original input byte array, but there may be multiple entries in the accounts slice that point to the same account. UsingRefCell ensures that the program does not accidentally perform overlapping read/writes to the same underlying data via multipleAccountInfo structures. If a program implements their own deserialization function care should be taken to handle duplicate accounts appropriately.

The instruction data is the general purpose byte array from theinstruction's instruction data being processed.

Heap#

Rust programs implement the heap directly by defining a customglobal_allocator

Programs may implement their ownglobal_allocator based on its specific needs. Refer to thecustom heap example for more information.

Restrictions#

On-chain Rust programs support most of Rust's libstd, libcore, and liballoc, as well as many 3rd party crates.

There are some limitations since these programs run in a resource-constrained, single-threaded environment, as well as being deterministic:

- No access to* rand
- std::fs
- std::net
- std::future
- std::process
- std::sync
- std::task
- std::thread
- std::time
- Limited access to:* std::hash
- std::os
- Bincode is extremely computationally expensive in both cycles and call depth
- and should be avoided
- String formatting should be avoided since it is also computationally
- expensive.
- No support forprintln!
- ,print!
- , the Solanalogging helpers
- should be used instead.
- The runtime enforces a limit on the number of instructions a program can
- execute during the processing of one instruction. Seecomputation budget
- for more
- information.

Depending on Rand#

Programs are constrained to run deterministically, so random numbers are not available. Sometimes a program may depend on a crate that depends itself onrand even if the program does not use any of the random number functionality. If a program depends onrand , the compilation will fail because there is noget-random support for Solana. The error will typically look like this:

error: target is not supported, for more information see: https://docs.rs/getrandom/#unsupported-targets --> /Users/jack/.cargo/registry/src/github.com-1ecc6299db9ec823/getrandom-0.1.14/src/lib.rs:257:9 | 257 | / compile_error!("\ 258 | | target is not supported, for more information see: \ 259 | | https://docs.rs/getrandom/#unsupported-targets\ 260 | | "); | |_____^ To work around this dependency issue, add the following dependency to the program'sCargo.toml :

getrandom = { version = "0.1.14", features = ["dummy"] } or if the dependency is on getrandom v0.2 add:

getrandom = { version = "0.2.2", features = ["custom"] }

Logging#

Rust'sprintln! macro is computationally expensive and not supported. Instead the helper macromsg! is provided.

msg! has two forms:

msg!("A string"); or

msg!(0_64, 1_64, 2_64, 3_64, 4_64); Both forms output the results to the program logs. If a program so wishes they can emulateprintln! by usingformat! :

msg!("Some variable: {:?}", variable); Thedebugging section has more information about working with program logs theRust

examples contains a logging example.

Panicking#

Rust'spanic! ,assert! , and internal panic results are printed to theprogram logs by default.

INFO solana_runtime::message_processor] Finalized account CGLhHSuWsp1gT4B7MY2KACqp9RUwQRhcUFfVSuxpSajZ INFO solana_runtime::message_processor] Call SBF program CGLhHSuWsp1gT4B7MY2KACqp9RUwQRhcUFfVSuxpSajZ INFO solana_runtime::message_processor] Program log: Panicked at: 'assertion failed: (left == right) left: 1, right: 2', rust/panic/src/lib.rs:22:5 INFO solana_runtime::message_processor] SBF program consumed 5453 of 200000 units INFO solana_runtime::message_processor] SBF program CGLhHSuWsp1gT4B7MY2KACqp9RUwQRhcUFfVSuxpSajZ failed: BPF program panicked

Custom Panic Handler#

Programs can override the default panic handler by providing their own implementation.

First define thecustom-panic feature in the program'sCargo.toml

[features] default = ["custom-panic"] custom-panic = [] Then provide a custom implementation of the panic handler:

[cfg(all(feature = "custom-panic", target_os = "solana"))]

[no_mangle]

fn custom_panic(info: &core::panic::PanicInfo<'_>) { solana_program::msg!("program custom panic enabled"); solana_program::msg!("{}", info); } In the above snippit, the default implementation is shown, but developers may replace that with something that better suits their needs.

One of the side effects of supporting full panic messages by default is that programs incur the cost of pulling in more of Rust'slibstd implementation into program's shared object. Typical programs will already be pulling in a fair amount oflibstd and may not notice much of an increase in the shared object size. But programs that explicitly attempt to be very small by avoidinglibstd may take a significant impact (~25kb). To eliminate that impact, programs can provide their own custom panic handler with an empty implementation.

[cfg(all(feature = "custom-panic", target_os = "solana"))]

[no_mangle]

fn custom_panic(info: &core::panic::PanicInfo<'_>) { // Do nothing to save space }

Compute Budget#

Use the system callsol_remaining_compute_units() to return au64 indicating the number of compute units remaining for this transaction.

Use the system callsol_log_compute_units() to log a message containing the remaining number of compute units the program may consume before execution is halted

Seecompute budget for more information.

ELF Dump#

The SBF shared object internals can be dumped to a text file to gain more insight into a program's composition and what it may be doing at runtime. The dump will contain both the ELF information as well as a list of all the symbols and the instructions that implement them. Some of the BPF loader's error log messages will reference specific instruction numbers where the error occurred. These references can be looked up in the ELF dump to identify the offending instruction and its context.

To create a dump file:

cd cargo build-bpf --dump

Examples#

TheSolana Program Library GitHub repo contains a collection of Rust examples.

TheSolana Developers Program Examples GitHub repo also contains a collection of beginner to intermediate Rust program examples.