Developing with C

Solana supports writing on-chain programs using the C and C++ programming languages.

Project Layout#

C projects are laid out as follows:

/src/ /makefile Themakefile should contain the following:

OUT_DIR := include ~/.local/share/solana/install/active_release/bin/sdk/sbf/c/sbf.mk The sbf-sdk may not be in the exact place specified above but if you setup your environment perHow to Build then it should be.

How to Build#

First setup the environment:

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

Then build using make:

make -C

How to Test#

Solana uses theCriterion test framework and tests are executed each time the program is builtHow to Build .

To add tests, create a new file next to your source file namedtest_c and populate it with criterion test cases. See theCriterion docs for information on how to write a test case.

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 SBF 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 loadersSBF Loader andSBF loader deprecated .

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

extern uint64_t entrypoint(const uint8_t *input) 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 ownhelper function .

Serialization#

Each loader provides a helper function that deserializes the program's input parameters into C types:

- SBF Loader deserialization
- SBF 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 must be written back into the input byte array.

Details on how the loader serializes the program inputs can be found in the [Input Parameter Serialization/docs/programs/faq#input-parameter-serialization) docs.

Data Types#

The loader's deserialization helper function populates theSolParameters structure:

/ * **Structure that the program's entrypoint input data is deserialized into.** / ***typedef struct { SolAccountInfo*** **ka; /**

Pointer to an array of SolAccountInfo, must already point to an array of SolAccountInfos *\/ uint64_t ka_num; / Number of SolAccountInfo entries in ka* / const uint8_t *data; / pointer to the instruction data*/ uint64_t data_len; / **Length in bytes of the instruction data *\/ const SolPubkey* program_id;** / program_id of the currently executing program */ } SolParameters; 'ka' is an ordered array of the accounts referenced by the instruction and represented as aSolAccountInfo 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 theSolAccountInfo structure are read-only except forlamports anddata . Both may be modified by the program in accordance with theruntime enforcement policy . When an instruction reference the same account multiple times there may be duplicateSolAccountInfo entries in the array but they both point back to the original input byte array. A program should handle these cases delicately to avoid overlapping read/writes to the same buffer. If a program implements their own deserialization function care should be taken to handle duplicate accounts appropriately.

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

program_id is the public key of the currently executing program.

Heap#

C programs can allocate memory via the system callcalloc or implement their own heap on top of the 32KB heap region starting at virtual address x300000000. The heap region is also used bycalloc so if a program implements their own heap it should not also callcalloc .

Logging#

The runtime provides two system calls that take data and log it to the program logs.

- sol_log(const char*)
- sol_log_64(uint64_t, uint64_t, uint64_t, uint64_t, uint64_t)

Thedebugging section has more information about working with program logs.

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 SBF 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 make dump_

Examples#

TheSolana Program Library github repo contains a collection of C examples