



Inside Stellar

The Rust Behind Stellar's Smart Contracts

soroban-sdk

Leigh McCulloch
Principal Software Engineer
Stellar Development Foundation

G'day everyone.

I'm Leigh, one of the maintainers of the soroban-sdk, the Rust SDK for building smart contracts on Stellar.

If you've written a contract, you know it looks like...ordinary Rust with a couple of attributes.

```
#[contract]
struct Token;

#[contractimpl]
impl Token {
    pub fn transfer(
        from: Address,
        to: Address,
        amount: i128,
    ) { ... }
}
```

If you've written a contract, you know it looks like...ordinary Rust with a couple of attributes.

That's intentional.

The attributes in the SDK are Rust macros that generate code. They take care of a lot of boilerplate, so contract developers get to focus on what matters: writing and testing their contract's logic.

Let's look at a simple contract, and see what gets generated.

Macros generate code.

```
#[contract]
struct Token;

#[contractimpl]
impl Token {
    pub fn transfer(
        from: Address,
        to: Address,
        amount: i128,
    ) { ... }
}
```

```
impl Token {
    pub fn transfer(...) { ... }
}

#[export_name = "transfer"]
pub extern "C" fn __Token__transfer__(
    arg_0: soroban_sdk::Val,
    arg_1: soroban_sdk::Val,
    arg_2: soroban_sdk::Val,
) → soroban_sdk::Val {
    Token::transfer(...)
}
```

On the left is what you write: one function. On the right is part of what the implementation macro turns it into: an extern "C" function that becomes the Wasm entry point.

Every contract function needs this plumbing, and this is only a small part of it. The macro also generates an interface definition, a test client, and code that registers the contract in tests.

If you want to see all of it for your own contract, run the cargo-expand tool.

So that's the simple case. Here's another contract with a lot more going on.

This impl block is empty. How does the Wasm export three functions?

crate **token** (our contract)

```
use pause::Pause;

#[contractimpl(contracttrait)]
impl Pause for Token {}
```

stellar contract info interface

```
fn paused() → bool
```

```
fn pause()
```

```
fn unpause()
```

This contract implements the Pause trait from a library. The Pause trait provides three functions with default implementations: paused, pause and unpause.

When we build the contract, it exports all three. The stellar-cli lists them in the contract interface, and the test client has a method for each.

It looks simple, but look closely, and something doesn't add up.

This impl block is empty. How does the Wasm export three functions?

crate **token** (our contract)

```
use pause::Pause;

#[contractimpl(contracttrait)]
impl Pause for Token {}
```

stellar contract info interface

```
fn paused() → bool
```

```
fn pause()
```

```
fn unpause()
```

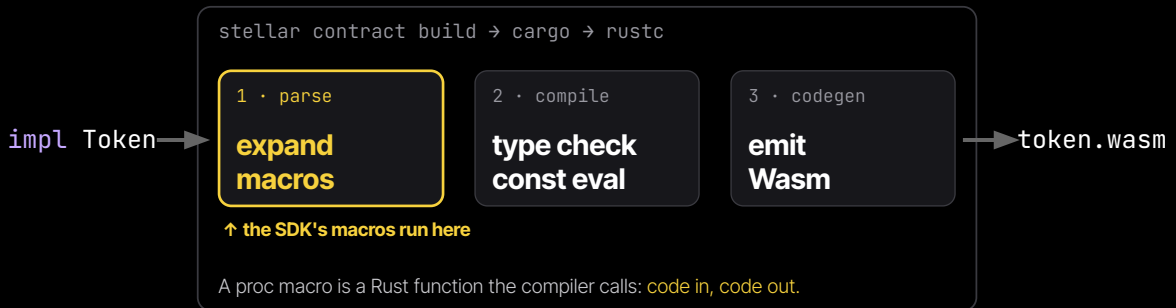
The contract implementation macro is on an empty implementation block.

So how does it generate all that plumbing for functions it doesn't see?

Today we'll find out, and look at three Rust features we combine with macros to do amazing things, like this.

Let's start with where macros run.

Macros run inside the build.



So macros run as part of the build.

A procedural macro, or proc macro, is a Rust function that the compiler calls to transform code: code goes in, code comes out.

When you build a contract, cargo runs the Rust compiler, and it calls the macros early in the pipeline.

Running early in the pipeline means that macros run before the compiler knows much about the program.

Proc macros are powerful, but they've got limitations.

01 · visibility

Macros only see their own items

→ recursive expansion

02 · types

Macros see code, not types

→ const evaluation

03 · coordination

Macros can't combine results across calls

→ module init functions

That gives them some limits.

The first is that a macro sees only the code it's attached to.

The second is that a macro sees code, not types. It sees the word "Address", not what Address is.

And the third is that a macro can't combine results across multiple call sites.

~~The SDK uses three Rust features to work around these limits.~~

`#[contractimpl]` can't see the trait.

01 · visibility → recursive expansion

crate `pause`

```
#[contracttrait]
pub trait Pause {
    fn paused() → bool { ... }
    fn pause() { ... }
    fn unpause() { ... }
}
```

crate `token` (our contract)

```
use pause::Pause;

#[contractimpl(contracttrait)]
impl Pause for Token {}
```

So remember that empty implementation block? That's problem one: visibility.

~~The `Pause` trait and its default functions live in a library. Our implementation lives in our contract.~~

When the implementation macro runs, it sees only this empty block. It can't see the trait or its functions. But it still has to generate an export, an interface entry and a client method for each function.

proc → declarative → proc

01 · visibility → recursive expansion

crate **pause**

```
#[contracttrait]
pub trait Pause { ... }
```

↓ generates, with the defaults baked in

```
macro_rules! Pause {
    paused, pause, unpause
}
```

crate **token** (our contract)

```
#[contractimpl(contracttrait)]
impl Pause for Token {}
```

The solution is that macros can generate macros, and Rust will keep expanding macros until there are none left.

The trait macro can see the default functions. So it generates a `macro_rules! Pause` macro with those functions baked into its body.

~~It names that macro `Pause`, the same as the trait. Types and macros live in separate namespaces, so the names don't clash, and one import brings in both.~~

The generated macro is going to be... a data channel from the trait macro to the implementation macro.

proc → declarative → proc

01 · visibility → recursive expansion

crate **pause**

```
#[contracttrait]
pub trait Pause { ... }
```

↓ generates, with the defaults baked in

```
macro_rules! Pause {
  paused, pause, unpause
}
```

crate **token** (our contract)

```
#[contractimpl(contracttrait)]
impl Pause for Token {}
```

↓ calls, with overridden fns

```
Pause!(Token, overridden: [])
```

↓ passes both lists to

```
contractimpl_trait_default_fns_
not_overridden!
```

↓ generates exports, spec, client

paused()

pause()

unpause()

The generated macro is going to be... a data channel from the trait macro to the implementation macro.

1) The implementation macro calls the generated macro, passing the functions the implementation overrides. In our case, we don't override any, so it passes the empty array.

proc → declarative → proc

01 · visibility → recursive expansion

crate **pause**

```
#[contracttrait]
pub trait Pause { ... }
```

↓ generates, with the defaults baked in

```
macro_rules! Pause {
    paused, pause, unpause
}
```

crate **token** (our contract)

```
#[contractimpl(contracttrait)]
impl Pause for Token {}
```

↓ calls, with overridden fns

```
Pause!(Token, overridden: [])
```

↓ passes both lists to

```
contractimpl_trait_default_fns_
not_overridden!
```

↓ generates exports, spec, client

paused()

pause()

unpause()

2) The compiler expands the generated macro where the implementation block is, which brings over the default functions from the trait.

proc → declarative → proc

01 · visibility → recursive expansion

crate **pause**

```
#[contracttrait]
pub trait Pause { ... }
```

↓ generates, with the defaults baked in

```
macro_rules! Pause {
  paused, pause, unpause
}
```

crate **token** (our contract)

```
#[contractimpl(contracttrait)]
impl Pause for Token {}
```

↓ calls, with overridden fns

```
Pause!(Token, overridden: [])
```

↓ passes both lists to

```
contractimpl_trait_default_fns_
not_overridden!
```

↓ generates exports, spec, client

`paused()`

`pause()`

`unpause()`

3) The generated macro hands both lists to a second proc macro, which generates the plumbing for the defaults that aren't overridden.

That's how an empty implementation block can generate the boilerplate for functions it can't see, and how traits support code reuse in contracts.

Macros see code, not types.

02 · types → const evaluation

```
crate token (our contract)
```

```
use soroban_sdk::Address;  
type Destination = Address;  
  
#[contractevent]  
pub struct Transfer {  
    #[topic] from: Address,  
    #[topic] to: Destination,  
    amount: i128,  
}
```

Macro sees**"Transfer"****"Address"****"Destination"****Compiler knows****token::Transfer****soroban_sdk::Address****soroban_sdk::Address**

The second problem is that a macro sees code, not types.

Every contract records its interface in the Wasm: its functions, types, errors and events. To write that accurately, the macro needs to know more about a type than its name.

The event macro sees only the name `Transfer`. If another module or library also has a `Transfer` type, there would be two types in the interface with the same name.

Macros see code, not types.

02 · types → const evaluation

crate **token** (our contract)

```
use soroban_sdk::Address;  
type Destination = Address;  
  
#[contractevent]  
pub struct Transfer {  
    #[topic] from: Address,  
    #[topic] to: Destination,  
    amount: i128,  
}
```

Macro sees

"Transfer"

"Address"

"Destination"

Compiler knows

token::Transfer

soroban_sdk::Address

soroban_sdk::Address

Type aliases make this even harder. The macro sees the word `Destination` and can't tell what it is. It could be a struct or an enum. Here, it's actually an `Address`.

Macros see code, not types.

02 · types → const evaluation

crate **token** (our contract)

```
use soroban_sdk::Address;
type Destination = Address;

#[contractevent]
pub struct Transfer {
    #[topic] from: Address,
    #[topic] to: Destination,
    amount: i128,
}
```

Macro sees

"Transfer"

"Address"

"Destination"

Compiler knows

token::Transfer

soroban_sdk::Address

soroban_sdk::Address

The compiler knows this `Transfer` is the one in our token contract, and that `Destination` is an `Address`.

So how can the code that the macros generate get at what the compiler knows?

Macros plus const expressions.

02 · types → const evaluation

```
const ENTRY: ScSpecEntry = ScSpecEntry::EventV0(ScSpecEventV0 {
    name: Transfer::SPEC_NAME, // "token::Transfer"
    params: [
        { name: "from", type: Address::SPEC_TYPE_DEF, ... },
        { name: "to", type: Destination::SPEC_TYPE_DEF, ... }, ...
    ], ...
});

#[link_section = "contractspecv0"]
static ENTRY_XDR: [u8; ENTRY.const_xdr_len()] = ENTRY.const_to_xdr();

impl SpecName for Transfer {
    const SPEC_NAME: &str = concat!(module_path!(), "::Transfer");
}
```

The answer is const expressions. Like macros, they're evaluated during the build, but later, by the compiler, once it knows the types.

The event macro doesn't build and encode the interface entry itself. It writes constants, and the compiler evaluates them: first the entry, then the encoding into bytes.

The values in the entry come from constants on each type.

For the name, it uses a constant defined next to the type, so it gets the full name: token::Transfer.

Macros plus const expressions.

02 · types → const evaluation

```
const ENTRY: ScSpecEntry = ScSpecEntry::EventV0(ScSpecEventV0 {
  name: Transfer::SPEC_NAME, // "token::Transfer"
  params: [
    { name: "from", type: Address::SPEC_TYPE_DEF, ... },
    { name: "to",   type: Destination::SPEC_TYPE_DEF, ... }, ...
  ], ...
});

#[link_section = "contractspecv0"]
static ENTRY_XDR: [u8; ENTRY.const_xdr_len()] = ENTRY.const_to_xdr();

impl SpecTypeDef for Address {
  const SPEC_TYPE_DEF: ScSpecTypeDef = ScSpecTypeDef::Address;
}
```



For each field, it uses a constant defined on that type, to get the real type. The macro doesn't know what `Destination` is, but the compiler does. It resolves it to `Address`, and uses `Address`'s constant.

Macros plus const expressions.

02 · types → const evaluation

```
const ENTRY: ScSpecEntry = ScSpecEntry::EventV0(ScSpecEventV0 {
    name: Transfer::SPEC_NAME, // "token::Transfer"
    params: [
        { name: "from", type: Address::SPEC_TYPE_DEF, ... },
        { name: "to",   type: Destination::SPEC_TYPE_DEF, ... }, ...
    ], ...
});

#[link_section = "contractspecv0"]
static ENTRY_XDR: [u8; ENTRY.const_xdr_len()] = ENTRY.const_to_xdr();
```

```
impl SpecTypeDef for Address {
    const SPEC_TYPE_DEF: ScSpecTypeDef = ScSpecTypeDef::Address;
}
```

The entry is encoded into bytes that end up in a section of the Wasm, where tools like the stellar-cli, the JavaScript SDK and stellar.expert read them.

All of this looks like program code, but it all runs during the build. None of this code runs on chain.

Each macro sees one block. None sees them all.

```
#[contractimpl]  
impl Token {  
  pub fn transfer(...) { ... }  
}
```

transfer

```
#[contractimpl(contracttrait)]  
impl Pause for Token {}
```

paused
pause,
unpause

The third problem is that a macro can't combine results across multiple call sites.

A contract's functions can be spread across multiple implementation blocks. Each macro sees only its own block, so no macro sees all the functions.

So why does that matter...

Native tests need a name → fn map. No macro sees all fns.

```
#[test]
fn test_transfer() {
    let env = Env::default();
    let addr = env.register(Token, ());
    let client = TokenClient::new(&env, &addr);

    ""
    // Call transfer function:
    client.transfer(&from, &to, &10);
}
```

So why does that matter... Tests.

Tests run our contract as native Rust, not as Wasm, so building is fast and we get to step through it in a debugger.

When a contract function is called, the test environment needs a map of function names to Rust functions, but no macro can build that map on its own.

Each impl registers its functions before main, before tests.

```
// From impl Token
#[ctor] fn ...() {
    FUNCS.insert("transfer", ...);
}
```

```
// From impl Pause for Token
#[ctor] fn ...() {
    FUNCS.insert("paused", ...);
    FUNCS.insert("pause", ...);
    FUNCS.insert("unpause", ...);
}
```

```
"transfer" → fn
"paused"   → fn
"pause"    → fn
"unpause"  → fn
```

```
#[test]
fn test_transfer() {
    ...
    client.transfer(&from, &to, &10);
}
```

The solution is module initialization functions: code that runs when the binary loads, before main and before any test. Every platform does this a little differently, but the ctor library abstracts all of that away.

Each implementation block generates a small ctor function that adds the functions to a...map.

Each impl registers its functions before main, before tests.

```
// From impl Token
#[ctor] fn ...() {
    FUNCS.insert("transfer", ...);
}
```

```
// From impl Pause for Token
#[ctor] fn ...() {
    FUNCS.insert("paused", ...);
    FUNCS.insert("pause", ...);
    FUNCS.insert("unpause", ...);
}
```

```
"transfer" → fn
"paused"   → fn
"pause"    → fn
"unpause"  → fn
```

```
#[test]
fn test_transfer() {
    ...
    client.transfer(&from, &to, &10);
}
```

Each implementation block generates a small ctor function that adds the functions to a...map.

By the time the first test runs, the map has every function, from every implementation block.

In the test...

Each impl registers its functions before main, before tests.

```
// From impl Token
#[ctor] fn ...() {
    FUNCS.insert("transfer", ...);
}
```

```
// From impl Pause for Token
#[ctor] fn ...() {
    FUNCS.insert("paused", ...); ...
}
```

```
"transfer" → fn
"paused"   → fn
"pause"    → fn
"unpause"  → fn
```

```
#[test]
fn test_transfer() {
    ...
    client.transfer(&from, &to, &10);
}
```

```
// In the soroban-env
invoke(addr, "transfer", args)
```

In the test... the Soroban environment looks up the function in that map and calls straight into our contract code.

This makes it possible to deploy and debug Rust contracts in tests directly.

By using these Rust features with macros, the SDK enables

recursive expansion

**Code sharing
with traits
across crates
and libraries**

const evaluation

**A contract
interface that
has strong type
identity**

module init functions

**Fast,
step-through
debuggable
Rust tests**

Quick recap.

Recursive macro expansion and the ability to generate macros make it possible to share code with traits.

Const evaluation gives us an interface with strong type identities, at no runtime cost.

Module initialization functions give us fast, debuggable native Rust tests.

If you're a contract developer, this is just some of what's inside the SDK, so that you get to focus on your contract.

Patterns for your own macros

recursive expansion

Need data from another item or crate?

Generate a macro as a data channel.

```
macro_rules! Pause {...}
```

const evaluation

Need to know what a type really is?

Emit a const, let the compiler evaluate.

```
const SPEC: _ = ...;
```

module init functions

Need to combine results from many macro calls?

Have each register a ctor.

```
#[ctor] fn reg() { ... }
```

 Each has its own limits and traps. Read up before using them.

If you write macros, you may be able to use these patterns in your Rust code.

If you:

~~Need data from another item or library, generate a macro as a data channel.~~

~~Need to know what a type is? Emit a const and let the compiler answer.~~

~~Need to combine results across call sites? Have each one generate a ctor function.~~

~~Each of these has its own gotchas, so read up on them before you use them.~~

Check out the code

`github.com/stellar/rs-soroban-sdk`

`soroban-sdk-macros/`

`tests-expanded/`

`soroban-sdk/docs/contracttrait.md`

Everything we looked at is in the soroban-sdk repo, including the full code behind these simplified examples. These directories are a great place to start.

Find me with questions. Thanks.