Package: Rtinycc (via r-universe)

November 12, 2026

Title Builds the 'TinyCC' Command-Line Interface and Library for 'C' Scripting in 'R'

Version 0.1.16.9000

Description Builds the 'TinyCC' (Tiny 'C' Compiler) command-line interface and library for package use in 'R'. The package compiles 'TinyCC' from source and provides R functions to interact with the compiler. 'TinyCC' can be used for header preprocessing, just-in-time compilation of 'C' code in 'R', and lightweight 'C' scripting workflows.

License GPL (>= 3)

Copyright See inst/LICENSE.note for bundled and adapted component copyright and licensing details.

Depends R (>= 4.4.0)

Suggests bench, callme, knitr, rmarkdown, tinytest, treesitter.c (>= 0.0.4)

VignetteBuilder knitr

SystemRequirements GNU make

Encoding UTF-8

Roxygen list(markdown = TRUE)

URL https://github.com/sounkou-bioinfo/Rtinycc, https://sounkou-bioinfo.github.io/Rtinycc/

BugReports https://github.com/sounkou-bioinfo/Rtinycc/issues

Config/roxygen2/version 8.0.0

Config/pak/sysreqs make

Repository https://sounkou-bioinfo.r-universe.dev

Date/Publication 2026-10-13 18:15:55 UTC

RemoteUrl https://github.com/sounkou-bioinfo/Rtinycc

RemoteRef HEAD

RemoteSha 5a163a0e8cc74e9b7154adc18fb09f451672c1aa

Contents

$.tcc_compiled . . . 4
as.character.tcc_cstring . . . 5
blas_lapack_info . . . 5
generate_trampoline . . . 6
get_external_ptr_addr . . . 6
is_callback_type . . . 7
parse_callback_type . . . 7
print.tcc_callback . . . 8
print.tcc_compiled . . . 8
print.tcc_cstring . . . 9
print.tcc_ffi . . . 9
tcc_add_file . . . 10
tcc_add_include_path . . . 10
tcc_add_library . . . 11
tcc_add_library_path . . . 11
tcc_add_symbol . . . 12
tcc_add_sysinclude_path . . . 12
tcc_bind . . . 13
tcc_call_symbol . . . 14
tcc_callback . . . 15
tcc_callback_async_drain . . . 16
tcc_callback_async_schedule . . . 16
tcc_callback_close . . . 17
tcc_callback_ptr . . . 17
tcc_callback_valid . . . 18
tcc_compile . . . 18
tcc_compile_string . . . 19
tcc_container_of . . . 19
tcc_cstring . . . 20
tcc_cstring_object . . . 20
tcc_data_ptr . . . 21
tcc_enum . . . 21
tcc_ffi . . . 22
tcc_field_addr . . . 22
tcc_free . . . 23
tcc_generate_bindings . . . 24
tcc_get_symbol . . . 25
tcc_global . . . 25
tcc_header . . . 26
tcc_include . . . 27
tcc_include_paths . . . 27
tcc_introspect . . . 28
tcc_library . . . 28
tcc_library_path . . . 29
tcc_link . . . 29
tcc_list_symbols . . . 31

tcc_malloc . . . 31
tcc_map_c_type_to_ffi . . . 32
tcc_null_ptr . . . 33
tcc_options . . . 33
tcc_output . . . 34
tcc_output_file . . . 34
tcc_path . . . 35
tcc_prefix . . . 35
tcc_ptr_addr . . . 36
tcc_ptr_free_set_null . . . 36
tcc_ptr_is_null . . . 37
tcc_ptr_is_owned . . . 37
tcc_ptr_set . . . 38
tcc_ptr_utils . . . 38
tcc_read_bytes . . . 39
tcc_read_cstring . . . 39
tcc_read_f32 . . . 40
tcc_read_f64 . . . 40
tcc_read_i16 . . . 41
tcc_read_i32 . . . 41
tcc_read_i64 . . . 42
tcc_read_i8 . . . 42
tcc_read_ptr . . . 43
tcc_read_u16 . . . 43
tcc_read_u32 . . . 44
tcc_read_u64 . . . 44
tcc_read_u8 . . . 45
tcc_recompile . . . 45
tcc_relocate . . . 46
tcc_run_cli . . . 46
tcc_set_options . . . 47
tcc_source . . . 47
tcc_state . . . 48
tcc_struct . . . 48
tcc_struct_raw_access . . . 49
tcc_symbol_is_valid . . . 50
tcc_treesitter_bindings . . . 50
tcc_treesitter_defines . . . 51
tcc_treesitter_enum_bindings . . . 52
tcc_treesitter_enum_members . . . 53
tcc_treesitter_enums . . . 53
tcc_treesitter_functions . . . 54
tcc_treesitter_global_types . . . 55
tcc_treesitter_globals . . . 55
tcc_treesitter_struct_accessors . . . 56
tcc_treesitter_struct_bindings . . . 57
tcc_treesitter_struct_members . . . 57
tcc_treesitter_structs . . . 58

tcc_treesitter_union_accessors 59
tcc_treesitter_union_bindings 60
tcc_treesitter_union_members 60
tcc_treesitter_unions 61
tcc_union 62
tcc_write_bytes 62
tcc_write_f32 63
tcc_write_f64 63
tcc_write_i16 64
tcc_write_i32 64
tcc_write_i64 65
tcc_write_i8 65
tcc_write_ptr 66
tcc_write_u16 66
tcc_write_u32 67
tcc_write_u64 67
tcc_write_u8 68

Index **69**

`$.tcc_compiled` *Access a compiled FFI symbol*

Description

Overrides `$` to detect dead pointers after deserialization and recompile transparently from the stored recipe.

Usage

```
## S3 method for class 'tcc_compiled'
x$name
```

Arguments

`x`	A tcc_compiled object
`name`	Symbol name to access

Value

The callable function or metadata field

as.character.tcc_cstring *Convert a* tcc_cstring *object to an R string*

Description

Convert a tcc_cstring object to an R string

Usage

```
## S3 method for class 'tcc_cstring'
as.character(x, ...)
```

Arguments

x	A tcc_cstring object.
...	Ignored.

Value

A character scalar containing the string value. Returns the cached R copy when available; otherwise reads the current NUL-terminated C string from x$ptr.

blas_lapack_info *Report active BLAS/LAPACK runtime information from R*

Description

Returns the BLAS/LAPACK runtime details as reported by R itself, plus convenience flags indicating whether Rblas and Rlapack appear available in loaded DLLs/shared objects.

Usage

```
blas_lapack_info()
```

Value

A named list with fields: blas_path, lapack_path, has_rblas, has_rlapack, loaded_dlls.

generate_trampoline *Generate trampoline code for a callback argument*

Description

Generate trampoline code for a callback argument

Usage

```
generate_trampoline(trampoline_name, sig)
```

Arguments

trampoline_name	Name of the trampoline function
sig	Parsed signature

Value

C code string for the trampoline

get_external_ptr_addr *Get the address of an external pointer*

Description

Extract the memory address from an external pointer as a numeric value. This is primarily useful for debugging and inspection purposes.

Usage

```
get_external_ptr_addr(ptr)
```

Arguments

ptr	An external pointer object (e.g., from `tcc_get_symbol()`).

Value

The memory address as a numeric value.

is_callback_type *Check if a type represents a callback*

Description

Check if a type represents a callback

Usage

```
is_callback_type(type)
```

Arguments

type	Type string to check

Value

Logical

parse_callback_type *Parse callback type specification*

Description

Parse callback type specification

Usage

```
parse_callback_type(type)
```

Arguments

type	Type string like "callback:double(int,int)"

Value

Parsed signature list or NULL

print.tcc_callback *Print tcc_callback object*

Description

Print tcc_callback object

Usage

```
## S3 method for class 'tcc_callback'
print(x, ...)
```

Arguments

x	A tcc_callback object
...	Ignored

Value

The input `tcc_callback` object, invisibly. Called for its side effect of printing the callback signature, thread-safety flag, and validity status.

print.tcc_compiled *Print tcc_compiled object*

Description

Print tcc_compiled object

Usage

```
## S3 method for class 'tcc_compiled'
print(x, ...)
```

Arguments

x	A tcc_compiled object
...	Ignored

Value

The input `tcc_compiled` object, invisibly. Called for its side effect of printing the compilation output mode and the status of compiled callable symbols.

print.tcc_cstring *Print a* tcc_cstring *object*

Description

Print a tcc_cstring object

Usage

```
## S3 method for class 'tcc_cstring'
print(x, ...)
```

Arguments

x	A tcc_cstring object.
...	Ignored.

Value

The input tcc_cstring object, invisibly. Called for its side effect of printing the current string value.

print.tcc_ffi *Print tcc_ffi object*

Description

Print tcc_ffi object

Usage

```
## S3 method for class 'tcc_ffi'
print(x, ...)
```

Arguments

x	A tcc_ffi object
...	Ignored

Value

The input tcc_ffi object, invisibly. Called for its side effect of printing the configured output mode, registered symbols, and selected libraries/include paths.

tcc_add_file *Add a source file to a libtcc state*

Description

Add a source file to a libtcc state

Usage

```
tcc_add_file(state, path)
```

Arguments

state	A tcc_state.
path	Path to a C source file.

Value

Integer status code (0 = success).

tcc_add_include_path *Add an include path to a libtcc state*

Description

Add an include path to a libtcc state

Usage

```
tcc_add_include_path(state, path)
```

Arguments

state	A tcc_state.
path	Path to include directory.

Value

Integer status code (0 = success).

tcc_add_library *Add a library to a libtcc state*

Description

Add a library to a libtcc state

Usage

```
tcc_add_library(state, library)
```

Arguments

state	A tcc_state.
library	Library name (e.g., "m" for libm, "R" for libR).

Value

Integer status code (0 = success).

tcc_add_library_path *Add a library path to a libtcc state*

Description

Add a library path to a libtcc state

Usage

```
tcc_add_library_path(state, path)
```

Arguments

state	A tcc_state.
path	Path to library directory.

Value

Integer status code (0 = success).

`tcc_add_symbol` *Add a symbol to a libtcc state*

Description

Add a symbol to a libtcc state

Usage

```
tcc_add_symbol(state, name, addr)
```

Arguments

`state`	A `tcc_state`.
`name`	Symbol name.
`addr`	External pointer address or symbol value.

Value

Integer status code (0 = success).

`tcc_add_sysinclude_path` *Add a system include path to a libtcc state*

Description

Add a system include path to a libtcc state

Usage

```
tcc_add_sysinclude_path(state, path)
```

Arguments

`state`	A `tcc_state`.
`path`	Path to system include directory.

Value

Integer status code (0 = success).

tcc_bind *Bind symbols with type specifications*

Description

Define symbols with Bun-style type specifications for API mode. This is the core of the declarative FFI API.

Usage

```
tcc_bind(ffi, ...)
```

Arguments

ffi	A tcc_ffi object
...	Named list of symbol definitions. Each definition is a list with: • args: List of fixed FFI argument types (e.g., list("i32", "f64")) • returns: FFI type for return value (e.g., "f64", "cstring") • variadic: Set TRUE for C varargs functions • varargs: Legacy typed variadic tail (exact/prefix mode) • varargs_types: Allowed scalar FFI types for true variadic tails • varargs_min: Minimum number of trailing varargs • varargs_max: Maximum number of trailing varargs (required for true variadic mode, defaults to varargs_min) • code: Optional C code for the symbol (for embedded functions) Callback arguments should use the form callback:<signature> (e.g., callback:double(double)). The generated trampoline expects a tcc_callback_ptr(cb) to the corresponding user-data parameter in the C API. For worker-thread scheduling, use callback_async:<signature>. Void callbacks are queued fire-and-forget; non-void callbacks block the worker until the main thread returns a result.

Value

Updated tcc_ffi object (for chaining)

Examples

```
ffi <- tcc_ffi() |>
  tcc_bind(
    add = list(args = list("i32", "i32"), returns = "i32"),
    greet = list(args = list("cstring"), returns = "cstring")
  )
```

`tcc_call_symbol` *Call a symbol from a TinyCC state*

Description

The memory state must first be successfully finalized with `tcc_relocate()`.

Usage

```
tcc_call_symbol(
  .state,
  .NAME,
  ...,
  return = c("int", "double", "void"),
  NAOK = FALSE
)
```

Arguments

`.state`	A `tcc_state`. Named `state =` is also accepted for compatibility with earlier releases.
`.NAME`	Symbol name to call. Named `name =` is also accepted for compatibility with earlier releases.
`...`	Optional arguments for `.C()`-style pointer calls.
`return`	One of `"int"`, `"double"`, or `"void"`. If `...` is present, the only supported value is `"void"`; when omitted with `...`, it defaults to `"void"`.
`NAOK`	If `FALSE`, integer/logical `NA` and non-finite numeric/complex values are rejected before the call, matching `.C()`'s default safety check. If `TRUE`, those values are passed through.

Details

With no additional arguments, `tcc_call_symbol()` preserves its historical quick-test behavior: call a zero-argument symbol and box an `int`, `double`, or `void` return value.

With additional arguments, it uses an R `.C()`-style calling convention: the target C function must be `void`, each atomic or character R argument is copied to guarded mutable call storage and passed by pointer, and the result is a list of the modified argument values. Supported argument mappings follow R's `.C()` interface: raw as `unsigned char *`, integer/logical as `int *`, numeric as `double *` or `float *` when `attr(x, "Csingle")` is true, complex as `Rcomplex *`, character as `char **`, lists as read-only `SEXP *`, and functions/environments/other R objects as read-only `SEXP`. Up to 65 arguments are supported. Guard bytes around copied buffers are checked after the call to catch simple native underwrites and overwrites; character code may edit string contents in place but must not replace `char *` elements in the `char **` array.

Non-atomic R objects are borrowed for the duration of the call only. C code must not mutate them through this interface, and must call `R_PreserveObject()` if it deliberately stores a `SEXP` beyond the call. This is a low-level convenience interface; for typed scalar returns, explicit zero-copy arrays, ownership metadata, and clearer signatures, prefer `tcc_ffi()`.

Value

For zero-argument scalar calls, the boxed return value (`NULL` for `void`). For `.C()`-style calls, a list mirroring `...` with any C-side modifications copied back.

`tcc_callback` *Register an R function as a callback*

Description

Wraps an R function so it can be passed as a C function pointer to compiled code. The callback will be invoked via a trampoline that marshals arguments between C and R.

Usage

```
tcc_callback(fun, signature, threadsafe = FALSE)
```

Arguments

`fun`	An R function to be called from C
`signature`	C function signature string (e.g., "double (*)(int, double)")
`threadsafe`	Informational compatibility flag. Thread dispatch is selected by `callback_async:<signature>` in `tcc_bind()`, not by this flag.

Details

Thread safety: ordinary `callback:<signature>` trampolines must be invoked on the R main thread. Use `callback_async:<signature>` when native workers need to dispatch to R. The `threadsafe` flag is informational only.

If a callback raises an error, a warning is emitted and a type-appropriate default value is returned.

When binding callbacks with `tcc_bind()`, use a `callback:<signature>` argument type so a synchronous trampoline is generated. The trampoline expects a `void*` user-data pointer as its first argument; pass `tcc_callback_ptr(cb)` as the user-data argument to the C API. For worker-thread dispatch, use `callback_async:<signature>`, which schedules the R call on the main thread. Void callbacks are fire-and-forget; non-void callbacks block the worker until the main thread returns the real result. Type-appropriate defaults are used only when dispatch or conversion fails.

Pointer arguments (e.g., `double*`, `int*`) are passed as external pointers. Lengths must be supplied separately if needed.

A callback `ptr` or `cstring` return is borrowed. C may consume it during the enclosing compiled call, but must not retain it after that call unless the owner is kept alive by some other explicit contract. Copy string data into C-owned storage when it must escape the callback call chain.

The return type may be any scalar type supported by the FFI mappings (e.g., `i32`, `f64`, `bool`, `cstring`), or `SEXP` to return an R object directly.

Callback lifetime: callbacks are eventually released by finalizers. Call `tcc_callback_close()` when you want deterministic invalidation and earlier release of the preserved R function.

Value

A tcc_callback object (externalptr wrapper)

---

tcc_callback_async_drain

Drain the async callback queue

Description

Execute any pending async callbacks immediately on the main R thread. Normally callbacks fire automatically via R's event loop (input handler on POSIX, message pump on Windows), so explicit draining is only needed in test harnesses or tight batch loops that never yield to R's event loop.

Usage

```
tcc_callback_async_drain()
```

Details

TCC-compiled C code that is known to be running on the main R thread can call `RC_callback_async_drain_c()` directly instead of returning to R. Functions bound with `callback_async:*` arguments are normally executed on a worker by the generated wrapper while the main thread drains, so user code in that target function should not assume it is itself on the main thread.

Value

NULL (invisible)

---

tcc_callback_async_schedule

Schedule a callback to run on the main thread

Description

Enqueue a callback for main-thread execution. Arguments must be basic scalars or external pointers. The callback must remain open until the task executes. External-pointer arguments are borrowed addresses: their owners and pointees must remain valid until execution finishes.

Usage

```
tcc_callback_async_schedule(callback, args = list())
```

Arguments

`callback`	A tcc_callback object
`args`	List of arguments to pass to the callback

Value

NULL (invisible)

---

tcc_callback_close *Close/unregister a callback*

Description

Invalidates a callback immediately, releases the preserved R function reference, and cleans up callback resources as early as possible. This is recommended for deterministic lifetime management, but callbacks are also eventually released by finalizers if you simply drop all references.

Usage

```
tcc_callback_close(callback)
```

Arguments

callback	A tcc_callback object returned by tcc_callback()

Value

NULL (invisible)

---

tcc_callback_ptr *Get the C-compatible function pointer*

Description

Returns an external pointer that can be passed to compiled C code as user data for trampolines. Keep this handle (and the original `tcc_callback`) alive for as long as C may call back. The pointer handle keeps the underlying token storage alive until it is garbage collected. Closing the original callback still invalidates the callback registry entry, so C must not continue invoking it after `tcc_callback_close()`.

Usage

```
tcc_callback_ptr(callback)
```

Arguments

callback	A tcc_callback object

Details

Pointer arguments and return values are treated as external pointers. Use `tcc_read_bytes()`, `tcc_read_u8()`, or `tcc_read_f64()` to inspect pointed data when needed.

Value

An external pointer (address of the callback token)

---

tcc_callback_valid *Check if callback is still valid*

Description

Check if callback is still valid

Usage

```
tcc_callback_valid(callback)
```

Arguments

callback	A tcc_callback object

Value

Logical indicating if callback can be invoked

---

tcc_compile *Compile FFI bindings*

Description

Compile the defined symbols into callable functions. This generates C wrapper code and compiles it with TinyCC.

Usage

```
tcc_compile(ffi, verbose = FALSE)
```

Arguments

ffi	A tcc_ffi object
verbose	Print compilation info

Value

A tcc_compiled object with callable functions

tcc_compile_string *Compile C code from a character string*

Description

Compile C code from a character string

Usage

```
tcc_compile_string(state, code)
```

Arguments

state	A tcc_state.
code	C source code string.

Value

Integer status code (0 = success).

tcc_container_of *Generate container_of helper for struct member*

Description

Creates a function that recovers the parent struct pointer from a pointer to one of its members. This is the classic Linux kernel container_of macro made accessible from R.

Usage

```
tcc_container_of(ffi, struct_name, member_name)
```

Arguments

ffi	A tcc_ffi object
struct_name	Struct name
member_name	Member field name to compute offset from

Value

Updated tcc_ffi object

Examples

```
## Not run:
ffi <- tcc_ffi() |>
  tcc_struct("student", list(id = "i32", marks = "i32")) |>
  tcc_container_of("student", "marks") # Creates struct_student_from_marks()

## End(Not run)
```

`tcc_cstring` *Create a C-style string pointer*

Description

Convert R character strings to C-style null-terminated string pointers. This handles UTF-8 encoding and null termination automatically.

Usage

```
tcc_cstring(str)
```

Arguments

`str`	Character string

Value

An external pointer tagged `"rtinycc_owned"` pointing to a malloc'd copy of the string. Freed on garbage collection or via `tcc_free()`.

`tcc_cstring_object` *CString S3 Class*

Description

Wrapper around a C string pointer with an optional cached R copy. Ownership follows the underlying external pointer; this wrapper does not add finalizer or freeing behavior on top of that pointer.

Usage

```
tcc_cstring_object(ptr, clone = TRUE, owned = FALSE)
```

Arguments

`ptr`	External pointer to C string
`clone`	Whether to clone the string immediately (safe for R use)
`owned`	Currently unused. Reserved for future finalizer support.

Value

A tcc_cstring object

---

tcc_data_ptr *Dereference a pointer-to-pointer*

Description

Treats `ptr_ref` as a pointer to a pointer and returns the pointed address as an external pointer. This is useful for fields like `void**` or `T**`.

Usage

```
tcc_data_ptr(ptr_ref)
```

Arguments

`ptr_ref`	External pointer to a pointer value (e.g., address of a field).

Value

An external pointer tagged `"rtinycc_borrowed"`. Not owned by Rtinycc and never freed on garbage collection. Do not pass to `tcc_free()`.

---

tcc_enum *Declare enum for FFI helper generation*

Description

Generate R-callable helpers for enum constants and type conversions. The enum must be defined in a header.

Usage

```
tcc_enum(ffi, name, constants = NULL, export_constants = FALSE)
```

Arguments

`ffi`	A tcc_ffi object
`name`	Enum name (as defined in C header)
`constants`	Character vector of constant names to export
`export_constants`	Whether to export enum constants as R functions

Value

Updated tcc_ffi object

Examples

```
## Not run:
ffi <- tcc_ffi() |>
  tcc_header("#include <errors.h>") |>
  tcc_enum("error_code", constants = c("OK", "ERROR"), export_constants = TRUE)

## End(Not run)
```

---

tcc_ffi *Create a new FFI compilation context*

Description

Initialize a Bun-style FFI context for API-mode compilation. This is the entry point for the modern FFI API.

Usage

```
tcc_ffi()
```

Value

A tcc_ffi object with chaining support

Examples

```
ffi <- tcc_ffi()
```

---

tcc_field_addr *Generate field address getter helpers*

Description

Creates functions that return pointers to specific struct fields. Useful for passing field pointers to C functions or for container_of.

Usage

```
tcc_field_addr(ffi, struct_name, fields)
```

Arguments

ffi	A tcc_ffi object
struct_name	Struct name
fields	Character vector of field names

Value

Updated tcc_ffi object

Examples

```
## Not run:
ffi <- tcc_ffi() |>
  tcc_struct("point", list(x = "f64", y = "f64")) |>
  tcc_field_addr("point", c("x", "y")) # point_x_addr(), point_y_addr()

## End(Not run)
```

---

tcc_free *Free owned memory*

Description

Free memory whose external pointer is tagged "rtinycc_owned" (e.g. from tcc_malloc() or tcc_cstring()). Errors on struct pointers (use the generated struct_<name>_free()) or borrowed pointers from tcc_data_ptr().

Usage

```
tcc_free(ptr)
```

Arguments

ptr	External pointer to free

Value

NULL.

`tcc_generate_bindings` *Generate bindings from header declarations*

Description

Generate bindings from header declarations

Usage

```
tcc_generate_bindings(
  ffi = NULL,
  header,
  mapper = tcc_map_c_type_to_ffi,
  functions = TRUE,
  structs = TRUE,
  unions = TRUE,
  enums = TRUE,
  globals = TRUE,
  bitfield_type = "u8",
  include_bitfields = TRUE
)
```

Arguments

`ffi`	A tcc_ffi object. If NULL, a new one is created.
`header`	Character scalar containing C declarations.
`mapper`	Function to map C types to FFI types.
`functions`	Logical; generate `tcc_bind()` specs for functions.
`structs`	Logical; generate `tcc_struct()` helpers.
`unions`	Logical; generate `tcc_union()` helpers.
`enums`	Logical; generate `tcc_enum()` helpers.
`globals`	Logical; generate `tcc_global()` getters/setters.
`bitfield_type`	FFI type to use for bitfields.
`include_bitfields`	Whether to include bitfields.

Value

Updated tcc_ffi object.

Examples

```
## Not run:
header <- "double sqrt(double x); struct point { double x; double y; };"
ffi <- tcc_generate_bindings(tcc_ffi(), header)

## End(Not run)
```

tcc_get_symbol *Get a symbol pointer from a libtcc state*

Description

The memory state must first be successfully finalized with `tcc_relocate()`.

Usage

```
tcc_get_symbol(state, name)
```

Arguments

`state`	A `tcc_state`.
`name`	Symbol name to look up.

Value

External pointer of class `tcc_symbol`. The pointer retains `state` so its relocated code remains alive for as long as the symbol is reachable.

tcc_global *Declare a global variable getter*

Description

Register a global C symbol so the compiled object exposes getter/setter functions `global_<name>_get()` and `global_<name>_set()`.

Usage

```
tcc_global(ffi, name, type)
```

Arguments

`ffi`	A tcc_ffi object
`name`	Global symbol name
`type`	FFI type for the global (scalar types only)

Details

Globals are limited to scalar FFI types. Array types are rejected. Scalar conversions follow the same rules as wrapper arguments:

- Integer inputs (`i8`, `i16`, `i32`, `u8`, `u16`) must be finite and within range; `NA` values error.
- Large integer types (`i64`, `u32`, `u64`) are mediated through R numeric (double). Values must be integer-valued and within range; for `i64`/`u64` only exact integers up to 2^53 are accepted.
- Getter wrappers for `i64`/`u64` warn when the stored value exceeds R's exact integer range for numeric vectors.
- `bool` rejects `NA` logicals.

Ownership notes:

- `ptr` globals store the raw address from an external pointer. If the external pointer owns memory, keep it alive; otherwise the pointer may be freed while the global still points to it.
- `cstring` globals are rejected because a global would outlive the borrowed R string view. Declare the global as `ptr` and manage its storage explicitly with an owned object such as `tcc_cstring()`.

Value

Updated tcc_ffi object (for chaining)

Note

Global helpers are generated inside the compiled TCC unit. Recompiling creates a new instance of the global variable; existing compiled objects continue to refer to their own copy.

Examples

```
ffi <- tcc_ffi() |>
  tcc_source("int global_counter = 7;") |>
  tcc_global("global_counter", "i32") |>
  tcc_compile()
ffi$global_global_counter_get()
```

---

tcc_header *Add C headers*

Description

Add C headers

Usage

```
tcc_header(ffi, header)
```

Arguments

ffi	A tcc_ffi object
header	Header string or include directive

Value

Updated tcc_ffi object (for chaining)

---

tcc_include *Add include path to FFI context*

Description

Add include path to FFI context

Usage

```
tcc_include(ffi, path)
```

Arguments

ffi	A tcc_ffi object
path	Include directory path

Value

Updated tcc_ffi object (for chaining)

---

tcc_include_paths *TinyCC include search paths*

Description

Returns the include directories used by the bundled TinyCC, including the package headers under `include/rtinycc`.

Usage

```
tcc_include_paths()

tcc_sysinclude_paths()
```

Value

A character vector of include directories.

tcc_introspect *Enable introspection helpers*

Description

Generates sizeof, alignof, and offsetof helper functions for structs, unions, and enums. Useful for debugging or when you need to know C layout information from R.

Usage

```
tcc_introspect(ffi)
```

Arguments

ffi	A tcc_ffi object

Value

Updated tcc_ffi object

tcc_library *Add library to link against*

Description

Add library to link against

Usage

```
tcc_library(ffi, library)
```

Arguments

ffi	A tcc_ffi object
library	Library name (e.g., "m", "sqlite3") or a path to a shared library (e.g., "libm.so.6"). When a path or platform library file name is provided, the library directory is added automatically and TinyCC is asked to link that exact file name. This keeps versioned runtime libraries such as `libm.so.6` distinct from generic linker names such as `m/libm.so`.

Value

Updated tcc_ffi object (for chaining)

tcc_library_path *Add library path to FFI context*

Description

Add library path to FFI context

Usage

```
tcc_library_path(ffi, path)
```

Arguments

ffi	A tcc_ffi object
path	Library directory path

Value

Updated tcc_ffi object (for chaining)

tcc_link *Link an external shared library with Bun-style FFI bindings*

Description

Link a system library (like libsqlite3) and generate type-safe wrappers automatically using TinyCC JIT compilation (API mode). Unlike dlopen(), this uses TinyCC to compile bindings that handle type conversion between R and C automatically.

Usage

```
tcc_link(
  path,
  symbols,
  headers = NULL,
  libs = character(0),
  lib_paths = character(0),
  include_paths = character(0),
  user_code = NULL,
  verbose = FALSE
)
```

Arguments

path	Library short name (e.g., "m", "sqlite3") or full path to the shared library. Short names stay on the normal linker-name path (-l<name>). File names such as libm.so.6 or full paths are resolved through the configured library search paths when needed and linked as exact files rather than collapsed to generic names like m.
symbols	Named list of symbol definitions with: • args: List of FFI types for arguments • returns: FFI type for return value
headers	Optional C headers to include
libs	Library names to link (e.g., "sqlite3")
lib_paths	Additional library search paths
include_paths	Additional include search paths
user_code	Optional custom C code to include in the compilation
verbose	Print debug information

Value

A tcc_compiled object with callable functions

Examples

```
## Not run:
# Link SQLite with type-safe bindings
sqlite <- tcc_link(
  "libsqlite3.so",
  symbols = list(
    sqlite3_libversion = list(args = list(), returns = "cstring"),
    sqlite3_open = list(args = list("cstring", "ptr"), returns = "i32")
  ),
  libs = "sqlite3"
)

# Call directly - type conversion happens automatically
sqlite$sqlite3_libversion()

# Example with custom user code for helper functions
math_with_helpers <- tcc_link(
  "m",
  symbols = list(
    sqrt = list(args = list("f64"), returns = "f64"),
    safe_sqrt = list(args = list("f64"), returns = "f64")
  ),
  user_code = "
    #include <math.h>

    // Helper function that validates input before calling sqrt
    double safe_sqrt(double x) {
```

```
      if (x < 0) {
        return NAN;
      }
      return sqrt(x);
    }
  ",
  libs = "m"
)
math_with_helpers$safe_sqrt(16.0)
math_with_helpers$safe_sqrt(-4.0) # Returns NaN for negative input

## End(Not run)
```

`tcc_list_symbols` *List symbols known to a libtcc state*

Description

Return the global symbols currently reported by libtcc for a state. This is a best-effort symbol-table inspection helper for compiled/linked TCC states, not a portable exhaustive symbol enumerator, not a DLL export scanner, and not a C signature discovery API. Platform backends may omit symbols that are still resolvable with `tcc_get_symbol()`. For meaningful runtime addresses, call it after `tcc_relocate()`.

Usage

```
tcc_list_symbols(state)
```

Arguments

`state`	A `tcc_state`.

Value

A data frame with columns `name` and `address`, where `address` is a hexadecimal character string.

`tcc_malloc` *Allocate memory buffer*

Description

Allocate a memory buffer of specified size, equivalent to C malloc. Returns an external pointer that can be passed to FFI functions.

Usage

```
tcc_malloc(size)
```

Arguments

size	Number of bytes to allocate

Value

An external pointer tagged "rtinycc_owned" with an R finalizer. Freed on garbage collection or explicitly via tcc_free().

---

tcc_map_c_type_to_ffi *Map a C type string to an Rtinycc FFI type*

Description

Map a C type string to an Rtinycc FFI type

Usage

```
tcc_map_c_type_to_ffi(c_type)
```

Arguments

c_type	C type string (e.g., "int", "double", "char *").

Details

This mapper is intentionally conservative for pointer types. char* is treated as a raw pointer (ptr) because C does not guarantee NUL-terminated strings. If you know the API expects a C string, map it explicitly to cstring in your custom mapper. Unknown scalar types and long double are rejected rather than assigned a potentially incompatible calling convention; provide a custom mapper when their ABI is known.

Value

A single FFI type string.

Examples

```
## Not run:
tcc_map_c_type_to_ffi("int")
tcc_map_c_type_to_ffi("const char *")

## End(Not run)
```

tcc_null_ptr *Create a NULL pointer*

Description

Creates a NULL pointer equivalent for use with FFI functions that expect optional pointer arguments.

Usage

```
tcc_null_ptr()
```

Value

An external pointer with NULL address

tcc_options *Add TinyCC compiler options to FFI context*

Description

Append raw options that are passed to `tcc_set_options()` before compiling generated wrappers (for example `"-O2"` or `"-Wall"`).

Usage

```
tcc_options(ffi, options)
```

Arguments

`ffi`	A tcc_ffi object
`options`	Character vector of option fragments

Value

Updated tcc_ffi object (for chaining)

tcc_output *Set output type for FFI compilation*

Description

The high-level FFI creates callable in-memory wrappers, so only `"memory"` output is supported. Executable and shared-library artifact output requires the low-level `tcc_state()` and `tcc_output_file()` path and is rejected here rather than returning dangling callable addresses.

Usage

```
tcc_output(ffi, output = "memory")
```

Arguments

`ffi`	A tcc_ffi object
`output`	Must be `"memory"`.

Value

Updated tcc_ffi object (for chaining)

tcc_output_file *Write a non-memory TinyCC output file*

Description

Finalize a state created with `output = "obj"`, `"dll"`, `"exe"`, or `"preprocess"` and write the resulting artifact. A state can be finalized only once. Memory states must instead use `tcc_relocate()`.

Usage

```
tcc_output_file(state, path)
```

Arguments

`state`	A non-memory `tcc_state`.
`path`	Destination file path.

Value

Integer status code (0 = success).

tcc_path *Locate the TinyCC executable*

Description

Returns the platform-specific `tcc` binary path (or `tcc.exe` on Windows), preferring the bundled installation.

Usage

```
tcc_path()
```

Value

A character scalar path.

tcc_prefix *TinyCC paths*

Description

Helpers to locate the bundled tinycc installation after the package is installed.

Usage

```
tcc_prefix()

tcc_lib_path()

tcc_lib_paths()

tcc_include_path()

tcc_bin_path()

tcc_cli()
```

Value

A character scalar path.

tcc_ptr_addr *Get pointer address as integer*

Description

Get the numeric address of an external pointer, useful for debugging and when APIs require pointer addresses as integers. Optional hex mode available.

Usage

```
tcc_ptr_addr(ptr, hex = FALSE)
```

Arguments

ptr	External pointer
hex	Whether to display in hexadecimal (default: FALSE)

Value

Character representation of pointer address (hex if requested, decimal otherwise)

tcc_ptr_free_set_null *Free the pointed memory and set to NULL*

Description

Frees the memory pointed to by `ptr_ref` and sets the pointer to NULL. Use this only when the pointed memory is not already owned by another external pointer with its own finalizer.

Usage

```
tcc_ptr_free_set_null(ptr_ref)
```

Arguments

ptr_ref	External pointer to a pointer value.

Value

The updated pointer reference (invisibly).

tcc_ptr_is_null *Check whether an external pointer is NULL*

Description

Returns TRUE if the external pointer address is NULL, FALSE otherwise.

Usage

```
tcc_ptr_is_null(ptr)
```

Arguments

ptr	External pointer

Value

Logical scalar

tcc_ptr_is_owned *Check for the "rtinycc_owned" tag*

Description

Returns TRUE only for pointers created by tcc_malloc() or tcc_cstring(). Struct pointers (tagged "struct_<name>") and borrowed pointers return FALSE.

Usage

```
tcc_ptr_is_owned(ptr)
```

Arguments

ptr	External pointer

Value

Logical scalar

tcc_ptr_set *Set a pointer-to-pointer value*

Description

Assigns the address in `ptr_value` to the location pointed to by `ptr_ref`.

Usage

```
tcc_ptr_set(ptr_ref, ptr_value)
```

Arguments

`ptr_ref`	External pointer to a pointer value (e.g., address of a field).
`ptr_value`	External pointer to store.

Value

The updated pointer reference (invisibly).

tcc_ptr_utils *Pointer and Buffer Utilities for FFI*

Description

Generic helper functions for common FFI operations inspired by Bun's FFI. These utilities handle pointer creation, buffer management, and memory operations that are commonly needed when working with external libraries.

Value

No return value. This is a documentation topic that groups the pointer and buffer utility functions described below.

tcc_read_bytes *Read raw bytes from a pointer*

Description

Read a fixed number of bytes from an external pointer into a raw vector.

Usage

```
tcc_read_bytes(ptr, nbytes)
```

Arguments

ptr	External pointer
nbytes	Number of bytes to read

Value

Raw vector

tcc_read_cstring *Read C-style string from pointer*

Description

Convert a C-style null-terminated string pointer back to R character string. Handles UTF-8 decoding automatically.

Usage

```
tcc_read_cstring(
  ptr,
  max_bytes = NULL,
  null_action = c("na", "empty", "error")
)
```

Arguments

ptr	External pointer to C string
max_bytes	Optional maximum number of bytes to read (fixed-length read).
null_action	Behavior when ptr is NULL: one of "na", "empty", "error". Only effective when max_bytes is provided; without it a NULL pointer returns "".

Value

Character string, or NA/"" for NULL pointers depending on null_action.

tcc_read_f32 *Read 32-bit float*

Description

Read 32-bit float

Usage

```
tcc_read_f32(ptr, offset = 0L)
```

Arguments

ptr	External pointer
offset	Byte offset from ptr (scalar interface). Ignored when n is supplied.

Value

Numeric scalar (promoted to double).

tcc_read_f64 *Read 64-bit doubles from a pointer*

Description

Read 64-bit doubles from a pointer

Usage

```
tcc_read_f64(ptr, n = NULL, offset = 0L)
```

Arguments

ptr	External pointer
n	Number of values to read (legacy vectorised interface). If provided, reads n consecutive f64 values starting at byte 0.
offset	Byte offset from ptr (scalar interface). Ignored when n is supplied.

Value

Numeric scalar (offset form) or numeric vector (n form).

tcc_read_i16 *Read signed 16-bit integer*

Description

Read signed 16-bit integer

Usage

```
tcc_read_i16(ptr, offset = 0L)
```

Arguments

ptr	External pointer
offset	Byte offset from ptr (scalar interface). Ignored when n is supplied.

Value

Integer scalar

tcc_read_i32 *Read signed 32-bit integers from a pointer*

Description

Read signed 32-bit integers from a pointer

Usage

```
tcc_read_i32(ptr, n = NULL, offset = 0L)
```

Arguments

ptr	External pointer
n	Number of values to read (legacy vectorised interface). If provided, reads n consecutive i32 values starting at byte 0.
offset	Byte offset from ptr (scalar interface). Ignored when n is supplied.

Value

Integer scalar (offset form) or integer vector (n form).

tcc_read_i64 *Read signed 64-bit integer*

Description

Read signed 64-bit integer

Usage

```
tcc_read_i64(ptr, offset = 0L)
```

Arguments

ptr	External pointer
offset	Byte offset from ptr (scalar interface). Ignored when n is supplied.

Value

Numeric scalar (double, exact up to 2^53).

tcc_read_i8 *Read signed 8-bit integer*

Description

Read signed 8-bit integer

Usage

```
tcc_read_i8(ptr, offset = 0L)
```

Arguments

ptr	External pointer
offset	Byte offset from ptr (scalar interface). Ignored when n is supplied.

Value

Integer scalar

tcc_read_ptr *Read a pointer at byte offset*

Description

Dereferences a `void*` at the given byte offset from `ptr`. Equivalent to `*(void**)(ptr + offset)`. The returned pointer is tagged `"rtinycc_borrowed"` and will not be freed by the garbage collector.

Usage

```
tcc_read_ptr(ptr, offset = 0L)
```

Arguments

`ptr`	External pointer
`offset`	Byte offset from `ptr` (scalar interface). Ignored when `n` is supplied.

Value

External pointer

tcc_read_u16 *Read unsigned 16-bit integer*

Description

Read unsigned 16-bit integer

Usage

```
tcc_read_u16(ptr, offset = 0L)
```

Arguments

`ptr`	External pointer
`offset`	Byte offset from `ptr` (scalar interface). Ignored when `n` is supplied.

Value

Integer scalar

tcc_read_u32 *Read unsigned 32-bit integer*

Description

Read unsigned 32-bit integer

Usage

```
tcc_read_u32(ptr, offset = 0L)
```

Arguments

ptr	External pointer
offset	Byte offset from ptr (scalar interface). Ignored when n is supplied.

Value

Numeric scalar (double, exact up to 2^32-1).

tcc_read_u64 *Read unsigned 64-bit integer*

Description

Read unsigned 64-bit integer

Usage

```
tcc_read_u64(ptr, offset = 0L)
```

Arguments

ptr	External pointer
offset	Byte offset from ptr (scalar interface). Ignored when n is supplied.

Value

Numeric scalar (double, exact up to 2^53).

tcc_read_u8 *Read unsigned 8-bit values from a pointer*

Description

Read unsigned 8-bit values from a pointer

Usage

```
tcc_read_u8(ptr, n = NULL, offset = 0L)
```

Arguments

ptr	External pointer
n	Number of values to read (legacy vectorised interface). If provided, reads n consecutive u8 values starting at byte 0.
offset	Byte offset from ptr (scalar interface). Ignored when n is supplied.

Value

Integer scalar (offset form) or integer vector (n form).

tcc_recompile *Recompile a tcc_compiled object*

Description

Explicitly recompile from the stored FFI recipe. Useful after deserialization (readRDS, unserialize) or to force a fresh compilation.

Usage

```
tcc_recompile(compiled)
```

Arguments

compiled	A tcc_compiled object

Value

The recompiled tcc_compiled object (invisibly, same environment)

tcc_relocate *Relocate compiled code*

Description

Relocation is valid only for a state created with `output = "memory"` and can be performed only once. Callable symbols are unavailable until relocation succeeds. Use `tcc_output_file()` for object, shared-library, executable, or preprocessor output.

Usage

```
tcc_relocate(state)
```

Arguments

`state`	A `tcc_state`.

Value

Integer status code (0 = success).

tcc_run_cli *Run the tinycc CLI*

Description

Run the tinycc CLI

Usage

```
tcc_run_cli(args = character(), tcc_path = check_cli_exists())
```

Arguments

`args`	Character vector of CLI arguments (e.g., `c("-c", file, "-o", out)`).
`tcc_path`	Optional path to the `tcc` binary; defaults to the bundled CLI.

Value

Integer status from `system2()`.

tcc_set_options *Apply raw TinyCC options to a libtcc state*

Description

Passes options directly to `tcc_set_options()` for the given state.

Usage

```
tcc_set_options(state, options)
```

Arguments

`state`	A `tcc_state`.
`options`	Character scalar of options (for example `"-O2 -Wall"`).

Value

Integer status code (`0` on success; negative on parse error).

tcc_source *Add C source code*

Description

Add C source code

Usage

```
tcc_source(ffi, code)
```

Arguments

`ffi`	A tcc_ffi object
`code`	C source code string

Value

Updated tcc_ffi object (for chaining)

tcc_state *Create a libtcc state*

Description

Initialize a libtcc compilation state, optionally pointing at the bundled include/lib paths. Memory states are finalized with `tcc_relocate()`; other output modes are written with `tcc_output_file()`. Finalization is single-shot, after which compilation options and source cannot be changed.

Usage

```
tcc_state(
  output = c("memory", "obj", "dll", "exe", "preprocess"),
  include_path = tcc_include_paths(),
  lib_path = tcc_lib_paths()
)
```

Arguments

`output`	Output type: one of "memory", "obj", "dll", "exe", "preprocess".
`include_path`	Path(s) to headers; defaults to the bundled include dirs.
`lib_path`	Path(s) to libraries; defaults to the bundled lib dirs (lib and lib/tcc).

Value

An external pointer of class `tcc_state`.

tcc_struct *Declare struct for FFI helper generation*

Description

Generate R-callable helpers for struct allocation, field access, and pointer management. The struct must be defined in a header.

Usage

```
tcc_struct(ffi, name, accessors)
```

Arguments

ffi	A tcc_ffi object
name	Struct name (as defined in C header)
accessors	Named list of field accessors where names are field names and values are FFI types (e.g., list(x="f64", y="f64")). Named nested struct fields can use "struct:<name>" to generate borrowed nested-view getters and copy-in setters (for example child = "struct:child"). Fixed character arrays use list(type = "cstring", size = n). Bare cstring pointer fields are rejected because setters cannot safely retain borrowed R string storage; use ptr with explicitly owned storage instead.

Value

Updated tcc_ffi object

Examples

```
## Not run:
ffi <- tcc_ffi() |>
  tcc_header("#include <point.h>") |>
  tcc_struct("point", list(x = "f64", y = "f64", id = "i32"))

## End(Not run)
```

---

tcc_struct_raw_access *Enable raw byte access for struct*

Description

Generates helper functions to read/write raw bytes from struct memory. Useful for bitwise operations, debugging, or manual serialization.

Usage

```
tcc_struct_raw_access(ffi, struct_name)
```

Arguments

ffi	A tcc_ffi object
struct_name	Struct name

Value

Updated tcc_ffi object

tcc_symbol_is_valid *Check if a tcc_symbol external pointer is valid*

Description

Check if a tcc_symbol external pointer is valid

Usage

```
tcc_symbol_is_valid(ptr)
```

Arguments

ptr	External pointer from `tcc_get_symbol()`.

Value

TRUE if the pointer address is non-null, FALSE otherwise.

tcc_treesitter_bindings *Generate bindings from a header*

Description

Generate bindings from a header

Usage

```
tcc_treesitter_bindings(
  header,
  mapper = tcc_map_c_type_to_ffi,
  ffi = NULL,
  functions = TRUE,
  structs = FALSE,
  unions = FALSE,
  enums = FALSE,
  globals = FALSE,
  bitfield_type = "u8",
  include_bitfields = TRUE
)
```

Arguments

header	Character scalar containing C declarations.
mapper	Function to map C types to FFI types.
ffi	Optional tcc_ffi object. When provided, returns an updated FFI object with generated bindings.
functions	Logical; generate tcc_bind() specs for functions.
structs	Logical; generate tcc_struct() helpers.
unions	Logical; generate tcc_union() helpers.
enums	Logical; generate tcc_enum() helpers.
globals	Logical; generate tcc_global() getters/setters.
bitfield_type	FFI type to use for bitfields.
include_bitfields	Whether to include bitfields.

Value

Named list suitable for tcc_bind() when ffi is NULL, otherwise an updated tcc_ffi object.

Examples

```
## Not run:
header <- "double sqrt(double x);"
symbols <- tcc_treesitter_bindings(header)

## End(Not run)
```

---

tcc_treesitter_defines

Extract macro defines from a header file

Description

Extract macro defines from a header file

Usage

```
tcc_treesitter_defines(
  file,
  use_cpp = TRUE,
  cc = treesitter.c::r_cc(),
  ccflags = NULL
)
```

Arguments

file	Path to a header file.
use_cpp	Logical; use the C preprocessor if available.
cc	Compiler string; passed to system2() if use_cpp = TRUE.
ccflags	Additional flags for the compiler.

Value

Character vector of macro names defined in file.

Examples

```
## Not run:
tcc_treesitter_defines("/usr/include/math.h")

## End(Not run)
```

---

tcc_treesitter_enum_bindings

Apply tcc_enum() bindings from a header

Description

Apply tcc_enum() bindings from a header

Usage

```
tcc_treesitter_enum_bindings(ffi, header, constants = NULL)
```

Arguments

ffi	A tcc_ffi object.
header	Character scalar containing C declarations.
constants	Named list of enum constants.

Value

Updated tcc_ffi object.

Examples

```
## Not run:
header <- "enum status { OK = 0, ERR = 1 };"
ffi <- tcc_ffi()
ffi <- tcc_treesitter_enum_bindings(ffi, header, constants = list(status = c("OK", "ERR")))

## End(Not run)
```

tcc_treesitter_enum_members *Parse enum members with treesitter.c*

Description

Parse enum members with treesitter.c

Usage

```
tcc_treesitter_enum_members(header, ...)
```

Arguments

header	Character scalar containing C declarations.
...	Additional arguments passed to `treesitter.c::get_enum_members_from_root()`.

Value

A data frame of enum members.

Examples

```
## Not run:
header <- "enum status { OK = 0, ERR = 1 };"
tcc_treesitter_enum_members(header)

## End(Not run)
```

tcc_treesitter_enums *Parse enum declarations with treesitter.c*

Description

Parse enum declarations with treesitter.c

Usage

```
tcc_treesitter_enums(header, ...)
```

Arguments

header	Character scalar containing C declarations.
...	Additional arguments passed to `treesitter.c::get_enum_nodes()`.

Value

A data frame of enum nodes.

Examples

```
## Not run:
header <- "enum status { OK = 0, ERR = 1 };"
tcc_treesitter_enums(header)

## End(Not run)
```

`tcc_treesitter_functions` *Parse function declarations with treesitter.c*

Description

Parse function declarations with treesitter.c

Usage

```
tcc_treesitter_functions(header, ...)
```

Arguments

`header`	Character scalar containing C declarations.
`...`	Additional arguments passed to `treesitter.c::get_function_nodes()`.

Value

A data frame of function nodes.

Examples

```
## Not run:
header <- "double sqrt(double x);"
tcc_treesitter_functions(header)

## End(Not run)
```

tcc_treesitter_global_types

Parse global declarations with types using treesitter.c

Description

Parse global declarations with types using treesitter.c

Usage

```
tcc_treesitter_global_types(header, ...)
```

Arguments

header	Character scalar containing C declarations.
...	Additional arguments passed to `treesitter.c::get_globals_with_types_from_root()`.

Value

A data frame of global names and C types.

Examples

```
## Not run:
header <- "int global_counter;"
tcc_treesitter_global_types(header)

## End(Not run)
```

tcc_treesitter_globals

Parse global declarations with treesitter.c

Description

Parse global declarations with treesitter.c

Usage

```
tcc_treesitter_globals(header, ...)
```

Arguments

header	Character scalar containing C declarations.
...	Additional arguments passed to `treesitter.c::get_globals_from_root()`.

Value

A data frame of global names.

Examples

```
## Not run:
header <- "int global_counter;"
tcc_treesitter_globals(header)

## End(Not run)
```

`tcc_treesitter_struct_accessors` *Generate tcc_struct() accessors from header structs*

Description

Generate tcc_struct() accessors from header structs

Usage

```
tcc_treesitter_struct_accessors(
  header,
  mapper = tcc_map_c_type_to_ffi,
  bitfield_type = "u8",
  include_bitfields = TRUE
)
```

Arguments

`header`	Character scalar containing C declarations.
`mapper`	Function to map C types to FFI types.
`bitfield_type`	FFI type to use for bitfields.
`include_bitfields`	Whether to include bitfields.

Value

Named list of accessors by struct name. Bitfields are returned as lists with `type`, `bitfield = TRUE`, and `width`. Named nested struct fields are returned as `"struct:<name>"`; ambiguous or anonymous nested structs fall back to `"ptr"`.

Examples

```
## Not run:
header <- "struct point { double x; double y; };"
tcc_treesitter_struct_accessors(header)

## End(Not run)
```

tcc_treesitter_struct_bindings

Apply tcc_struct() bindings from a header

Description

Apply tcc_struct() bindings from a header

Usage

```
tcc_treesitter_struct_bindings(ffi, header, ...)
```

Arguments

ffi	A tcc_ffi object.
header	Character scalar containing C declarations.
...	Passed to tcc_treesitter_struct_accessors().

Value

Updated tcc_ffi object.

Examples

```
## Not run:
header <- "struct point { double x; double y; };"
ffi <- tcc_ffi()
ffi <- tcc_treesitter_struct_bindings(ffi, header)

## End(Not run)
```

tcc_treesitter_struct_members

Parse struct members (including bitfields) with treesitter.c

Description

Parse struct members (including bitfields) with treesitter.c

Usage

```
tcc_treesitter_struct_members(header, ...)
```

Arguments

header	Character scalar containing C declarations.
...	Additional arguments passed to treesitter.c::get_struct_members().

Value

A data frame of struct members.

Examples

```
## Not run:
header <- "struct point { double x; double y; };"
tcc_treesitter_struct_members(header)

## End(Not run)
```

`tcc_treesitter_structs`

Parse struct declarations with treesitter.c

Description

Parse struct declarations with treesitter.c

Usage

```
tcc_treesitter_structs(header, ...)
```

Arguments

`header`	Character scalar containing C declarations.
`...`	Additional arguments passed to `treesitter.c::get_struct_nodes()`.

Value

A data frame of struct nodes.

Examples

```
## Not run:
header <- "struct point { double x; double y; };"
tcc_treesitter_structs(header)

## End(Not run)
```

`tcc_treesitter_union_accessors` *Generate tcc_union() accessors from header unions*

Description

Generate tcc_union() accessors from header unions

Usage

```
tcc_treesitter_union_accessors(
  header,
  mapper = tcc_map_c_type_to_ffi,
  bitfield_type = "u8",
  include_bitfields = TRUE
)
```

Arguments

`header`	Character scalar containing C declarations.
`mapper`	Function to map C types to FFI types.
`bitfield_type`	FFI type to use for bitfields.
`include_bitfields`	Whether to include bitfields.

Value

Named list of accessors by union name. Bitfields are returned as lists with `type`, `bitfield = TRUE`, and `width`. Nested struct members are returned as `list(type = "struct", struct_name = <name>)` when the struct name is available, otherwise `list(type = "struct")`.

Examples

```
## Not run:
header <- "union data { int i; double d; };"
tcc_treesitter_union_accessors(header)

## End(Not run)
```

tcc_treesitter_union_bindings

Apply tcc_union() bindings from a header

Description

Apply tcc_union() bindings from a header

Usage

```
tcc_treesitter_union_bindings(ffi, header, ...)
```

Arguments

ffi	A tcc_ffi object.
header	Character scalar containing C declarations.
...	Passed to tcc_treesitter_union_accessors().

Value

Updated tcc_ffi object.

Examples

```
## Not run:
header <- "union data { int i; double d; };"
ffi <- tcc_ffi()
ffi <- tcc_treesitter_union_bindings(ffi, header)

## End(Not run)
```

tcc_treesitter_union_members

Parse union members with treesitter.c

Description

Parse union members with treesitter.c

Usage

```
tcc_treesitter_union_members(header, ...)
```

Arguments

header	Character scalar containing C declarations.
...	Additional arguments passed to treesitter.c::get_union_members_from_root().

Value

A data frame of union members.

Examples

```
## Not run:
header <- "union data { int i; double d; };"
tcc_treesitter_union_members(header)

## End(Not run)
```

`tcc_treesitter_unions` *Parse union declarations with treesitter.c*

Description

Parse union declarations with treesitter.c

Usage

```
tcc_treesitter_unions(header, ...)
```

Arguments

`header`	Character scalar containing C declarations.
`...`	Additional arguments passed to `treesitter.c::get_union_nodes()`.

Value

A data frame of union nodes.

Examples

```
## Not run:
header <- "union data { int i; double d; };"
tcc_treesitter_unions(header)

## End(Not run)
```

tcc_union *Declare union for FFI helper generation*

Description

Generate R-callable helpers for union allocation and member access. The union must be defined in a header.

Usage

```
tcc_union(ffi, name, members, active = NULL)
```

Arguments

ffi	A tcc_ffi object
name	Union name (as defined in C header)
members	Named list of union members with FFI types. Bare cstring pointer members are rejected; use ptr with explicitly owned storage.
active	Default active member for accessors

Value

Updated tcc_ffi object

Examples

```
## Not run:
ffi <- tcc_ffi() |>
  tcc_union("data_variant",
    members = list(as_int = "i32", as_float = "f32"),
    active = "as_int"
  )

## End(Not run)
```

tcc_write_bytes *Write raw bytes to a pointer*

Description

Write a raw vector into memory pointed to by an external pointer.

Usage

```
tcc_write_bytes(ptr, raw)
```

Arguments

ptr	External pointer
raw	Raw vector to write

Value

NULL.

---

tcc_write_f32 *Write a 32-bit float*

Description

Write a 32-bit float

Usage

```
tcc_write_f32(ptr, offset, value)
```

Arguments

ptr	External pointer
offset	Byte offset
value	Integer value to write

Value

NULL (invisibly). Called for its side effect of writing one 32-bit floating-point value into native memory at ptr + offset.

---

tcc_write_f64 *Write a 64-bit double*

Description

Write a 64-bit double

Usage

```
tcc_write_f64(ptr, offset, value)
```

Arguments

ptr	External pointer
offset	Byte offset
value	Integer value to write

Value

NULL (invisibly). Called for its side effect of writing one 64-bit floating-point value into native memory at ptr + offset.

---

tcc_write_i16 *Write a signed 16-bit integer*

Description

Write a signed 16-bit integer

Usage

```
tcc_write_i16(ptr, offset, value)
```

Arguments

ptr	External pointer
offset	Byte offset
value	Integer value to write

Value

NULL (invisibly). Called for its side effect of writing one signed 16-bit value into native memory at ptr + offset.

---

tcc_write_i32 *Write a signed 32-bit integer*

Description

Write a signed 32-bit integer

Usage

```
tcc_write_i32(ptr, offset, value)
```

Arguments

ptr	External pointer
offset	Byte offset
value	Integer value to write

Value

NULL (invisibly). Called for its side effect of writing one signed 32-bit value into native memory at ptr + offset.

tcc_write_i64 *Write a signed 64-bit integer*

Description

Write a signed 64-bit integer

Usage

```
tcc_write_i64(ptr, offset, value)
```

Arguments

ptr	External pointer
offset	Byte offset
value	Integer value to write

Value

NULL (invisibly). Called for its side effect of writing one signed 64-bit value into native memory at ptr + offset.

tcc_write_i8 *Write a signed 8-bit integer*

Description

Write a signed 8-bit integer

Usage

```
tcc_write_i8(ptr, offset, value)
```

Arguments

ptr	External pointer
offset	Byte offset
value	Integer value to write

Value

NULL (invisibly).

tcc_write_ptr *Write a pointer at byte offset*

Description

Write a pointer at byte offset

Usage

```
tcc_write_ptr(ptr, offset, value)
```

Arguments

ptr	External pointer (destination buffer)
offset	Byte offset
value	External pointer to write

Value

NULL (invisibly).

tcc_write_u16 *Write an unsigned 16-bit integer*

Description

Write an unsigned 16-bit integer

Usage

```
tcc_write_u16(ptr, offset, value)
```

Arguments

ptr	External pointer
offset	Byte offset
value	Integer value to write

Value

NULL (invisibly). Called for its side effect of writing one unsigned 16-bit value into native memory at ptr + offset.

tcc_write_u32 *Write an unsigned 32-bit integer*

Description

Write an unsigned 32-bit integer

Usage

```
tcc_write_u32(ptr, offset, value)
```

Arguments

ptr	External pointer
offset	Byte offset
value	Integer value to write

Value

NULL (invisibly). Called for its side effect of writing one unsigned 32-bit value into native memory at ptr + offset.

tcc_write_u64 *Write an unsigned 64-bit integer*

Description

Write an unsigned 64-bit integer

Usage

```
tcc_write_u64(ptr, offset, value)
```

Arguments

ptr	External pointer
offset	Byte offset
value	Integer value to write

Value

NULL (invisibly). Called for its side effect of writing one unsigned 64-bit value into native memory at ptr + offset.

`tcc_write_u8` *Write an unsigned 8-bit integer*

Description

Write an unsigned 8-bit integer

Usage

```
tcc_write_u8(ptr, offset, value)
```

Arguments

`ptr`	External pointer
`offset`	Byte offset
`value`	Integer value to write

Value

`NULL` (invisibly). Called for its side effect of writing one unsigned 8-bit value into native memory at `ptr + offset`.

Index

$.tcc_compiled, 4

as.character.tcc_cstring, 5

blas_lapack_info, 5

generate_trampoline, 6
get_external_ptr_addr, 6

is_callback_type, 7

parse_callback_type, 7
print.tcc_callback, 8
print.tcc_compiled, 8
print.tcc_cstring, 9
print.tcc_ffi, 9

tcc_add_file, 10
tcc_add_include_path, 10
tcc_add_library, 11
tcc_add_library_path, 11
tcc_add_symbol, 12
tcc_add_sysinclude_path, 12
tcc_bin_path *(*tcc_prefix*)*, 35
tcc_bind, 13
tcc_bind(), *15*
tcc_call_symbol, 14
tcc_callback, 15
tcc_callback_async_drain, 16
tcc_callback_async_schedule, 16
tcc_callback_close, 17
tcc_callback_ptr, 17
tcc_callback_valid, 18
tcc_cli *(*tcc_prefix*)*, 35
tcc_compile, 18
tcc_compile_string, 19
tcc_container_of, 19
tcc_cstring, 20
tcc_cstring(), *23*, *26*, *37*
tcc_cstring_object, 20
tcc_data_ptr, 21
tcc_data_ptr(), *23*
tcc_enum, 21
tcc_ffi, 22
tcc_ffi(), *14*
tcc_field_addr, 22
tcc_free, 23
tcc_free(), *20*, *21*, *32*
tcc_generate_bindings, 24
tcc_get_symbol, 25
tcc_get_symbol(), *31*
tcc_global, 25
tcc_header, 26
tcc_include, 27
tcc_include_path *(*tcc_prefix*)*, 35
tcc_include_paths, 27
tcc_introspect, 28
tcc_lib_path *(*tcc_prefix*)*, 35
tcc_lib_paths *(*tcc_prefix*)*, 35
tcc_library, 28
tcc_library_path, 29
tcc_link, 29
tcc_list_symbols, 31
tcc_malloc, 31
tcc_malloc(), *23*, *37*
tcc_map_c_type_to_ffi, 32
tcc_null_ptr, 33
tcc_options, 33
tcc_output, 34
tcc_output_file, 34
tcc_path, 35
tcc_prefix, 35
tcc_ptr_addr, 36
tcc_ptr_free_set_null, 36
tcc_ptr_is_null, 37
tcc_ptr_is_owned, 37
tcc_ptr_set, 38
tcc_ptr_utils, 38
tcc_read_bytes, 39
tcc_read_cstring, 39

tcc_read_f32, 40
tcc_read_f64, 40
tcc_read_i16, 41
tcc_read_i32, 41
tcc_read_i64, 42
tcc_read_i8, 42
tcc_read_ptr, 43
tcc_read_u16, 43
tcc_read_u32, 44
tcc_read_u64, 44
tcc_read_u8, 45
tcc_recompile, 45
tcc_relocate, 46
tcc_relocate(), *14*, *25*, *31*, *34*, *48*
tcc_run_cli, 46
tcc_set_options, 47
tcc_source, 47
tcc_state, 48
tcc_state(), *34*
tcc_struct, 48
tcc_struct_raw_access, 49
tcc_symbol_is_valid, 50
tcc_sysinclude_paths
        *(*tcc_include_paths*)*, 27
tcc_treesitter_bindings, 50
tcc_treesitter_defines, 51
tcc_treesitter_enum_bindings, 52
tcc_treesitter_enum_members, 53
tcc_treesitter_enums, 53
tcc_treesitter_functions, 54
tcc_treesitter_global_types, 55
tcc_treesitter_globals, 55
tcc_treesitter_struct_accessors, 56
tcc_treesitter_struct_bindings, 57
tcc_treesitter_struct_members, 57
tcc_treesitter_structs, 58
tcc_treesitter_union_accessors, 59
tcc_treesitter_union_bindings, 60
tcc_treesitter_union_members, 60
tcc_treesitter_unions, 61
tcc_union, 62
tcc_write_bytes, 62
tcc_write_f32, 63
tcc_write_f64, 63
tcc_write_i16, 64
tcc_write_i32, 64
tcc_write_i64, 65
tcc_write_i8, 65
tcc_write_ptr, 66
tcc_write_u16, 66
tcc_write_u32, 67
tcc_write_u64, 67
tcc_write_u8, 68