REIMS DLL Developer Guide

Release V2.1

Authors: D. Furfaro, J. Kosek

Sep 11, 2026

CONTENTS

REIMS can load user functions from external DLLs at runtime. A single DLL can implement any combination of DLL entry points: friction correlations, Nusselt correlations, signals, and material properties.

All interfaces follow the C calling convention (`bind(C)`), so the DLL can be written in any language capable of producing a Windows DLL whose functions are directly callable by other programs — such as Fortran, C, C++, etc. The examples in this guide and in `tests/dynamic_lib/` use Intel Fortran (ifx), but the interfaces are identical in any other language. Only the export syntax differs (`__declspec(dllexport)` in C/C++ instead of `!DIR$ ATTRIBUTES DLLEXPORT` in Fortran).

CHAPTER

ONE

HOW IT WORKS

Every DLL interface follows the same two-phase pattern:

- **Init function** (called once at startup): reads the YAML parameters, stores them in a Fortran derived type, and shares with REIMS a pointer to that type (`c_pt`) and a function pointer to the user function.
- **User function** (called at every time step): receives `c_pt`, recovers the Fortran type from it, and computes the result.

`c_pt` is an opaque `type(c_ptr)`: in the init function store your type with `c_pt = c_loc(state)`; in the user function recover it with `call c_f_pointer(c_pt, state_ptr)`.

CHAPTER

TWO

BUILD

```
call "<path_to_intel_oneAPI>\compiler\<version>\env\vars.bat"
del my_lib.dll my_lib.exp my_lib.lib my_lib.obj
ifx /dll my_lib.f90 /Fe:my_lib.dll
```

Place the DLL in the REIMS working directory and declare it once in the YAML:

```
external_libs:
- my_lib   # covers all DLL entry points implemented in my_lib.dll
```

2.1 Parsing YAML parameters with `next_pair`

The `next_pair` helper parses the YAML parameters passed to every init function as a serialized `key\0value\0...` string. Copy it into your module:

```
function next_pair(buf, buf_len, offset, key, val) result(has_data)
  character(*,kind=c_char), intent(in)    :: buf
  integer,                  intent(in)    :: buf_len
  integer,                  intent(inout) :: offset
  character(:), allocatable, intent(out)  :: key, val
  logical :: has_data
  has_data = offset < buf_len
  if (.not. has_data) return
  key    = buf(offset : offset + index(buf(offset:buf_len), c_null_char) - 2)
  offset = offset + len(key) + 1
  val    = buf(offset : offset + index(buf(offset:buf_len), c_null_char) - 2)
  offset = offset + len(val) + 1
end function next_pair
```

Typical usage in an init function:

```
type(my_state_t), pointer :: state   ! your derived type
integer                   :: offset
character(:), allocatable :: key, val

offset = 1
do while (next_pair(cfg_str, cfg_len, offset, key, val))
  select case (key)
  case ("alpha")
    read(val, *) state%alpha
  case ("beta")
    read(val, *) state%beta
  end select
end do
```

2.2 Minimal complete example (friction)

The following is a self-contained, compilable friction correlation that reads one YAML parameter (`alpha`) and returns `alpha / Re`:

```
module my_lib_m
  use iso_c_binding
  use iso_fortran_env, only: dp => real64
  implicit none
  public :: init_friction_ext, my_friction

  type :: state_t
    real(c_double) :: alpha
  end type

contains

  ! --- helper ---------------------------------------------------------
  function next_pair(buf, buf_len, offset, key, val) result(has_data)
    character(*,kind=c_char), intent(in)    :: buf
    integer,                  intent(in)    :: buf_len
    integer,                  intent(inout) :: offset
    character(:), allocatable, intent(out)  :: key, val
    logical :: has_data
    has_data = offset < buf_len
    if (.not. has_data) return
    key    = buf(offset : offset + index(buf(offset:buf_len), c_null_char) - 2)
    offset = offset + len(key) + 1
    val    = buf(offset : offset + index(buf(offset:buf_len), c_null_char) - 2)
    offset = offset + len(val) + 1
  end function

  ! --- user function --------------------------------------------------
  function my_friction(c_pt, Re) bind(C, name="my_friction") result(f)
    !DIR$ ATTRIBUTES DLLEXPORT :: my_friction
    type(c_ptr),    value :: c_pt
    real(c_double), value :: Re
    real(c_double)        :: f
    type(state_t), pointer :: s
    call c_f_pointer(c_pt, s)
    f = s%alpha / Re
  end function

  ! --- init function --------------------------------------------------
  function init_friction_ext(cfg_len, cfg_str, c_pt, c_frict) &
      bind(C, name="init_friction_ext") result(ok)
    !DIR$ ATTRIBUTES DLLEXPORT :: init_friction_ext
    integer(c_int), value :: cfg_len
    character(*)          :: cfg_str
    type(c_ptr)           :: c_pt
    type(c_funptr)        :: c_frict
    logical(c_bool)       :: ok
    type(state_t), pointer :: s
    integer                :: offset
    character(:), allocatable :: key, val

    c_pt    = C_NULL_PTR
    c_frict = C_NULL_FUNPTR
```

(continues on next page)

(continued from previous page)

```
    allocate(s)
    s%alpha = 0.316d0   ! default

    offset = 1
    do while (next_pair(cfg_str, cfg_len, offset, key, val))
      if (key == "alpha") read(val, *) s%alpha
    end do

    c_pt    = c_loc(s)
    c_frict = c_funloc(my_friction)
    ok      = logical(.true., c_bool)
  end function

end module my_lib_m
```

Note

If the init function returns `ok = .false.`, REIMS aborts the simulation with an error message referencing the correlation name.

CHAPTER

THREE

DLL ENTRY POINTS

The table below summarises the interface of each DLL interface.

DLL interface	Init function name	User function signature
Friction	`init_friction_ext` (fixed)	`f(c_pt, Re)` → `real`
Nusselt	`init_nusselt_ext` (fixed)	`Nu(c_pt, Re, Pra, Tother, T)` → `real`
Signal	freely chosen in YAML `external` key	`subroutine(c_pt, time, signal(*))`
Material	`init_material_ext` (fixed)	up to 8 user functions, signature depends on property (see section below)

Each DLL entry point is detailed in its own section below.

CHAPTER

FOUR

FRICTION CORRELATION

```yaml
friction_correlations:
  - friction: my_friction   # name chosen freely
    alpha:    0.75          # user parameter

components:
  - type: channel
    friction: my_friction
```

```
function my_friction(c_pt, Re) bind(C, name="my_friction") result(f)
  !DIR$ ATTRIBUTES DLLEXPORT :: my_friction
  type(c_ptr),    value :: c_pt
  real(c_double), value :: Re
  real(c_double)        :: f
  ! ...
end function

function init_friction_ext(cfg_len, cfg_str, c_pt, c_frict) &
    bind(C, name="init_friction_ext") result(ok)
  !DIR$ ATTRIBUTES DLLEXPORT :: init_friction_ext
  integer(c_int), value :: cfg_len
  character(*)          :: cfg_str
  type(c_ptr)           :: c_pt
  type(c_funptr)        :: c_frict
  logical(c_bool)       :: ok
  c_pt    = C_NULL_PTR
  c_frict = C_NULL_FUNPTR
  ! allocate state, parse cfg_str, set c_frict = c_funloc(my_friction)
  ok = logical(.true., c_bool)
end function
```

CHAPTER

FIVE

NUSSELT CORRELATION

```yaml
nusselt_correlations:
  - nusselt:   my_nusselt   # name chosen freely
    my_param:  1.5          # user parameter

components:
  - type: fluidlink
    nusselt: my_nusselt
```

```
function my_nusselt(c_pt, Re, Pra, Tother, T) &
    bind(C, name="my_nusselt") result(Nu)
  !DIR$ ATTRIBUTES DLLEXPORT :: my_nusselt
  type(c_ptr),    value :: c_pt
  real(c_double), value :: Re, Pra, Tother, T
  real(c_double)        :: Nu
  ! ...
end function

function init_nusselt_ext(cfg_len, cfg_str, c_pt, c_nuss) &
    bind(C, name="init_nusselt_ext") result(ok)
  !DIR$ ATTRIBUTES DLLEXPORT :: init_nusselt_ext
  integer(c_int), value :: cfg_len
  character(*)          :: cfg_str
  type(c_ptr)           :: c_pt
  type(c_funptr)        :: c_nuss
  logical(c_bool)       :: ok
  c_pt   = C_NULL_PTR
  c_nuss = C_NULL_FUNPTR
  ! allocate state, parse cfg_str, set c_nuss = c_funloc(my_nusselt)
  ok = logical(.true., c_bool)
end function
```

CHAPTER

SIX

SIGNAL

The init function name is set freely in the YAML under the `external` key. Each signal (field, flux, current…) can point to a different init function in the same DLL.

```yaml
components:
  - type: strand
    field:
      external: my_field_init   # name of init function in the DLL
      coef:     3.5             # user parameter
```

```
subroutine my_signal(c_pt, time, signal) bind(C, name="my_signal")
  !DIR$ ATTRIBUTES DLLEXPORT :: my_signal
  type(c_ptr),    value :: c_pt
  real(c_double), value :: time
  real(c_double)        :: signal(*)   ! size = n (number of spatial nodes)
  ! ...
end subroutine

function my_field_init(fct_ptr_from_reims, cfg_len, cfg_str, c_pt, c_sign, n) &
    bind(C, name="my_field_init") result(ok)
  !DIR$ ATTRIBUTES DLLEXPORT :: my_field_init
  type(c_funptr), value :: fct_ptr_from_reims   ! pointer to expose_state_to_ext_
↪library
  integer(c_int), value :: cfg_len
  character(*)          :: cfg_str
  type(c_ptr)           :: c_pt
  type(c_funptr)        :: c_sign
  integer(c_int), value :: n                    ! number of spatial nodes
  logical(c_bool)       :: ok
  ! allocate state, store n, parse cfg_str, set c_sign = c_funloc(my_signal)

  ! To access the REIMS simulation state (see next section):
  if (.not. associated(reims_state%fpt)) then
    call c_f_procpointer(fct_ptr_from_reims, reims_state%fpt)
    call reims_state%fpt(reims_state%h5_ptr)
  end if

  ok = logical(.true., c_bool)
end function
```

6.1 Accessing REIMS simulation state

This mechanism is only available for signals — it is the only init function that receives `fct_ptr_from_reims`. It points to an internal REIMS routine that fills a C-compatible structure giving read access to all simulation tables and component data. Declare the following types and a module-level variable in your DLL:

```
abstract interface
  subroutine expose_state_if(h5_ptr) bind(C)
    import :: c_ptr
    type(c_ptr) :: h5_ptr
  end subroutine
end interface

type, bind(C) :: h5_export_c_t
  integer(c_int) :: nb_tables
  type(c_ptr)    :: tables_ptr   ! → tbl_dsc_c_t(:)
end type

type, bind(C) :: tbl_dsc_c_t
  type(c_ptr)    :: name_ptr     ! table name
  integer(c_int) :: name_len
  type(c_ptr)    :: vars_ptr     ! variable names (packed buffer, fixed width var_len)
  integer(c_int) :: n_vars
  integer(c_int) :: var_len
  type(c_ptr)    :: comps_ptr    ! → comp_desc_c_t(:)
  integer(c_int) :: n_comps
end type

type, bind(C) :: comp_desc_c_t
  type(c_ptr)    :: name_ptr     ! component name
  integer(c_int) :: name_len
  type(c_ptr)    :: node_x_ptr   ! node positions  → real(c_double)(:)
  integer(c_int) :: n_node
  type(c_ptr)    :: data_ptr     ! data matrix     → real(c_double)(:,:)
  integer(c_int) :: n1           ! = n_node
  integer(c_int) :: n2           ! = number of variables
end type

type reims_state_t
  procedure(expose_state_if), pointer, nopass :: fpt => null()
  type(c_ptr) :: h5_ptr = C_NULL_PTR
end type

type(reims_state_t), save :: reims_state   ! module-level, shared across all signals
```

Once `reims_state%h5_ptr` is populated (called once in any init function), navigate the hierarchy in the user function (e.g. in the signal function body):

```
type(h5_export_c_t), pointer :: H
type(tbl_dsc_c_t),   pointer :: tables(:)
type(comp_desc_c_t), pointer :: comps(:)
real(c_double),      pointer :: data(:,:)

call c_f_pointer(reims_state%h5_ptr, H)
call c_f_pointer(H%tables_ptr, tables, [H%nb_tables])
call c_f_pointer(tables(1)%comps_ptr, comps, [tables(1)%n_comps])
call c_f_pointer(comps(1)%data_ptr, data, [comps(1)%n1, comps(1)%n2])
```

CHAPTER

SEVEN

MATERIAL PROPERTIES

Only implement the properties you need — any function pointer left as `C_NULL_FUNPTR` will fall back to the REIMS built-in implementation.

The user functions fall into three groups depending on which property they implement:

Group 1 — 2 arguments (`res`, `th_cond`, `stra`, `t_crit`, `bc`):

Property	Signature
Resistivity	`f(c_pt, T, B) → real`
Thermal conductivity	`f(c_pt, T, B) → real`
Strain	`f(c_pt, B, I) → real`
Critical temperature	`f(c_pt, B, St) → real`
Critical field	`f(c_pt, T, St) → real`

```
! Example for resistivity (same pattern for th_cond, stra, t_crit, bc)
function my_res(c_pt, T, B) bind(C, name="my_res") result(val)
  !DIR$ ATTRIBUTES DLLEXPORT :: my_res
  type(c_ptr),    value :: c_pt
  real(c_double), value :: T   ! temperature (K)
  real(c_double), value :: B   ! magnetic field (T)
  real(c_double)        :: val
  ! ...
end function
```

Group 2 — 5 arguments (`jc`, `heat_cap`):

Property	Signature
Critical current density	`f(c_pt, T, B, St, Tc0, Bc) → real`
Heat capacity	`f(c_pt, T, B, TC, TCS, TC0) → real`

```
! Example for critical current density
function my_jc(c_pt, T, B, St, Tc0, Bc) bind(C, name="my_jc") result(val)
  !DIR$ ATTRIBUTES DLLEXPORT :: my_jc
  type(c_ptr),    value :: c_pt
  real(c_double), value :: T, B, St, Tc0, Bc
  real(c_double)        :: val
  ! ...
end function
```

Group 3 — 8 arguments (`tcs`):

Property	Signature
Current sharing temperature	`f(c_pt, B, St, Jop, Bc0, Jc0, Tc, Tc0, Bc) → real`

```
! Example for current sharing temperature
function my_tcs(c_pt, B, St, Jop, Bc0, Jc0, Tc, Tc0, Bc2) bind(C, name="my_tcs")␣
↪result(val)
  !DIR$ ATTRIBUTES DLLEXPORT :: my_tcs
  type(c_ptr),    value :: c_pt
  real(c_double), value :: B, St, Jop, Bc0, Jc0, Tc, Tc0, Bc2
  real(c_double)        :: val
  ! ...
end function

function init_material_ext(cfg_len, cfg_str, c_pt, &
    c_res, c_th_cond, c_stra, c_t_crit, c_bc, c_jc, c_tcs, c_heat_cap, &
    density, E0, nPow) bind(C, name="init_material_ext") result(ok)
  !DIR$ ATTRIBUTES DLLEXPORT :: init_material_ext
  integer(c_int), value :: cfg_len
  character(*)          :: cfg_str
  type(c_ptr)           :: c_pt
  type(c_funptr)        :: c_res, c_th_cond, c_stra, c_t_crit
  type(c_funptr)        :: c_bc, c_jc, c_tcs, c_heat_cap
  real(c_double)        :: density, E0
  integer               :: nPow
  logical(c_bool)       :: ok
  c_pt      = C_NULL_PTR
  c_res     = C_NULL_FUNPTR;  c_th_cond = C_NULL_FUNPTR
  c_stra    = C_NULL_FUNPTR;  c_t_crit  = C_NULL_FUNPTR
  c_bc      = C_NULL_FUNPTR;  c_jc      = C_NULL_FUNPTR
  c_tcs     = C_NULL_FUNPTR;  c_heat_cap= C_NULL_FUNPTR
  density   = -1.0d0;  E0 = -1.0d0;  nPow = -1
  ! set only what you implement, e.g.:
  c_res   = c_funloc(my_res)
  density = 8960.0d0
  ok = logical(.true., c_bool)
end function
```