Modules Extensions – Why?

- Partition large modules to ease maintenance problems.
- Allow for precompiled main program and modules, with user-provided submodules.
- Avoid recompilation / recertification cascades.
- Package proprietary software components to allow easily publishing their interfaces.
- Improve possibilities for library organization.

Modules Extensions – What?

1. Allow an interface body for a module procedure.

2. Provide for modules to have submodules.

3. Items 1 – 2 allow interface body and procedure body to be in different program units.

There is very little

New Syntax

There is a new prefix for a SUBROUTINE or FUNCTION statement in an interface body:

MODULE [(*module-or-submodule-name*)]

There are two new statements:

SUBMODULE (*parent-name*) :: &
    & *submodule-name*

And

END SUBMODULE [*submodule-name*]

That's All!

Interface body for a module procedure

- Interface body's subroutine or function statement has a MODULE prefix.

- Such an interface body DOES access its environment by host association.

- The interface body can optionally specify where the procedure body is – for humans and inter-module inlining optimizers.

- Identical to existing interface bodies in all other respects.

Example interface body for a module procedure

```
module M ! could be in a submodule

  integer, parameter :: RK = kind(0.0e0)
  interface [ generic stuff if desired ]
    MODULE(S1) subroutine sub ( arg1, arg2 )
      ! Notice that RK is accessed by host
      ! association
      real(rk), intent(in) :: arg1
      real(rk), intent(out) :: arg2
    end subroutine sub
  end interface

end module M
```

The processor doesn't check that SUB is defined in S1. That would cause M to depend on S1, not vice-versa.

Separate procedure body for a module procedure

Repeating the interface in the procedure's body is optional:

```
subroutine sub
    ...
end subroutine sub
```

or

```
subroutine sub ( arg1, arg2 )
    real(rk), intent(in) :: arg1
    real(rk), intent(out) :: arg2
    ...
end subroutine sub
```

are both allowed. If any dummy arguments are declared here, *all of the dummy arguments have to be declared here, and their characteristics and names have to be the same as in the interface body*.

Separate procedure body for a module procedure (cont.)

Characteristics of a function result have to be specified in the interface body. They can be repeated in the separate procedure's body, but if so they have to be identical to what's declared in the interface body.

Specifying PURE and ELEMENTAL are optional in the separate procedure body's declaration, but if they're specified, *they have to be specified already in the interface body*: They're characteristics.

As at present, RECURSIVE or RESULT can be specified in the interface body, but they have no effect there: They're not characteristics. If you really want them, they have to be specified in the separate procedure body's declaration.

Submodules

- A Module can have any number of submodules, including zero (for compatibility).

- Submodules can have any number of submodules, including zero.

- Each submodule specifies its parent module or submodule.

- The only place a parent specifies a submodule name is optionally in an interface body for a separate module procedure body – but it's not checked there: That would make the parent depend on its submodule, not vice-versa.

- A submodule accesses its parent program unit by host association.

Submodule example

```
SUBMODULE(M) :: S1 ! Parent program unit is M
  ! Nothing here is accessible by use
  ! association, but everything here is
  ! accessible by host association, here and
  ! in subsidiary submodules
CONTAINS
  subroutine SUB ! args in interface body
    ! SUB is accessible by use association if
    ! its interface body is.  It is accessible
    ! by host association in every descendant
    ! program unit of the one where its
    ! interface body is declared.  The
    ! processor can check that SUB is in S1,
    ! as advertised in its interface body.
    ....
  end subroutine SUB
END SUBMODULE S1
```