Some Fortran guidelines and pitfalls

E. Branlard - Version: 1.0-2-gf0a45d6

June 17, 2017

Contents

Introduction

Portability Portability Portability It is rather easy to write a code that is *none standards, platform-dependent, architecture-dependent, compiler-dependent and library-dependent.* Writing a portable code takes more effort on the programmer's level but it has its benefits: robustness, re-usability and more users. You learn a lot and find a lot of bugs simply by using different compilers.

Strategies There is probably two main strategies to ensure portability: (i) using preprocessor directives or (ii) wrapping non-portable code in separate files. (i) Preprocessors directives are widely used, they are written close to the code which makes them easy to implement. The down side is they pollute the code and make it often very hard to read. Also, there is an inherent dependence on a specific preprocessor, and thus this reduces portability. The C-preprocessor is built-in in most compilers, and is probably the one that should be used if a preprocessor approach is chosen. (ii) Choosing to wrap non-portable code into separate files keeps the whole code perfectly readable. Further, the developers responsible for a given library, architecture or OS have only a few files to work with, the rest of the code being "universal". The down side is that it increases the number of source files and requires the "Makefile" or the "Project" to select the proper files. Also, not all the power of the preprocessor can be achieved in this way. This is for instance the case for decoration of procedures or data like `bind(C)` or `DLLEXPORT` (see subsection 1.14). A combination of both approaches is likely to be the optimal solution.

In this document The guidelines written in this document are an attempt to lean towards portability. The content of the document might evolve in the future. Wrapping of non-portable code into a separate module is the solution generally applied in this document. Different files are used to provide the different implementations of the interface of a given wrap-module. Portability modules are called `Support*` and their source code is located in the folder `_support*`. For instance, the kind of a real is stored as `MK` in the `SupportPrecision` module (see subsection 1.10).

If recent standards offer options that helps portability, then this document will tend to favor these options despite the broken compatibility with regards to older compilers. This is the case for the `bind(C)` decoration which is very helpful for portability.

The following convention is used in this document:

```
! BAD PRACTICE
```

```
! GOOD PRACTICE
```

In this repository Preliminary attempts of "portable" modules and makefiles are provided in this repository. Most of the code originates from `Omnivor`'s implementation. It used to be compatible linux/windows and gfortran/intel/compaq/sun/portland, though it was not fully tested recently. They are far from "universal" and will hopefully evolve in the future.

1 Recommendations

1.1 Files and modules: towards a modular library-oriented code

Files and modules: general guidelines

1. Non-portable code should be placed in a separate directory to avoid polluting the "universal" code.
2. Modules containing derived types definitions should not contain anything else: no data or routines (though parameters/constants can be fine). This seriously reduces the chances to run into circular dependencies problems.
3. Modules containing data should only contain this data: no types or routines. This reduces circular dependencies problems. This helps the identification of the data "stored" by the libraries. In fact, it's best to reduce data modules to a minimum (see next point).
4. Data modules should be avoided. In a "library"-like implementation, the "user" owns the data, not the library. The library is made of tools that manipulate the user's data but does not store anything. This makes the library thread safe and the code implemented can be more readily re-used.
5. Use one module per files, it makes it easier to find them and reduces circular dependencies problem.
6. The following structure can be adopted:
   - `AirfoilTypes.f90` (contains no data)
   - `AirfoilParams.f90` (contains compile time constants, e.g. to increase code readibility)
   - `AirfoilTools.f90` (contains no data, manipulate derive type instances as arguments)
   - `AirfoilData.f90` (fine, but the aim is to remove this module at the end, see point 4 above)
7. File extensions: Some OS are case-sensitive. Make sure you display file extensions in your file manager. It's good to stay consistent in the extension you use for your source files. It helps creating makefiles that can be used on any platforms. Some general conventions are:
   - `.f90`: Fortran 90 code
   - `.f` : Old fortran code
   - `.F*` : Fortran code that needs a preprocessor or Template for code generator (e.g. cheetah)
8. Avoid having two files with the same name in your project. It's easier to implement makefiles then.

Modules: write implicit none in module only, it propagates to contained routines

```
module SupportCompiler
  implicit none
contains
  subroutine foo()
    implicit none ! Not needed
  end subroutine
end module
```

```
module SupportCompiler
  implicit none
contains
  subroutine foo()

  end subroutine
end module
```

Modules: write explicit use-statements to avoid polluting your scope and help the reader

```
subroutine foo()
  use SupportCompiler, only: FORTRAN_COMPILER
  use AirfoilTools, only: airfoil_load, airfoil_interp
  use NewTools, only: init              ! using    the subroutine init from NewTools
  use OldTools, only: init_old => init ! renaming the subroutine init from OldTools
  ![...]
  call init()
  call init_old()
  ![...]
end subroutine
```

1.2 Declarations, initialization, allocations

Initialization: not in definitions but straight after (except for derived types, see subsection 1.6)

```
subroutine foo()
  integer :: i = 0 !< implies save, bad!!!!
  real(MK), pointer :: p=>null() !< idem


end subroutine
```

```
subroutine foo()
  integer :: i
  real(MK), pointer :: p
  i=0         ! safe
  nullify(p) ! safe
end subroutine
```

Finalizing in the definition implies the attribute `save`, it's very bad practice and can lead to disastrous surprises (see examples in `_unit_tests`). Save is in general a bad practice (like global variables).

Arguments declaration: use intent in declarations, except for pointers

```
function(x,y,i,p)
  ! Arguments
  real(MK)  :: x !<
  real(MK)  :: y !<
  integer   :: i !<
  integer, pointer :: p !<
  ! Variables
end
```

```
function(x,y,i,p)
  ! Arguments
  real(MK), intent(in)    :: x !< best
  real(MK), intent(out)   :: y !< best
  integer, intent(inout) :: i !< best
  integer, pointer :: p
  ! Variables
end
```

More compiler optimizations can take place and errors detected at compilation time.

Initialization: always initialize after allocation

```
allocate(x(1:10));! x is garbage
```

```
allocate(x(1:10)); x(1:10)= 0.0_MK ! safe
```

Compilers have flags to define the behavior of allocate (e.g. set to 0 or NaN). It's more portable not to rely on it. In statements and declarations, specifying the bounds explicitly is good practice: it reminds the reader of the dimensions, it helps the compiler, and bound mismatch can be found by the compiler.

Allocations: the safe way

```
allocate(x(1:10));
! Code above can crash with no backtrace
x(1:10)=0.0_MK


!
!
```

```
allocate(x(1:10),stat=ierr);
if (ierr/=0) print*,'x alloc error'; STOP
x(1:10)=0.0_MK

! or, using a wrapped function:
use MemoryManager, only: allocate_safe
call allocate_safe('x', x, 10, 0.0_MK)
```

Data declaration within module: use save and initialization

```fortran
module A
    ! save is more or less implied
    integer :: i
    integer, pointer :: p
end module
```

```fortran
module A
    ! good practice to write 'save', and init
    integer, save          :: i = 0
    integer, save, pointer :: p => null()
end module
```

The value of `i` may be lost if the module becomes out of scope. In practice it doesn't occur, but it's just safe to write `save`... It's probably the only time the `save` attribute should be used. As mentionned in subsection 1.1, data modules are to be reduced to a minimal and it's best if they contain only data. Note: for `common` blocks, save should be used as well for the same reasons.

1.3 Stack pitfalls

Stack: do not use assumed size local variable in routines

```fortran
function(x,n)
! Arguments
integer,intent(in) :: n
real(MK),dimension(n),intent(inout) :: x !< ok
! Variables
real(MK), dimension(n) :: y !< BAD!

end
```

```fortran
function(x,n)
! Arguments
integer,intent(in) :: n
real(MK),dimension(n),intent(inout) :: x !< ok
! Variables
real(MK), dimension(:), allocatable :: y !< OK
allocate(y(1:n)); y(1:n)= 0.0_MK          !  OK
end
```

The assumed size local variables are allocated on the stack and this might result in stack overflows or corruptions.

Stack: do not use intrinsic functions for large arrays/vectors (they sometimes use the stack)

```fortran
maxval  ! Examples of known functions
maxloc  ! acting on array/vector that can use
pack    ! the stack (e.g. Intel compiler)
```

```fortran
! Instead, write your own custom function
! using a for loop and no assumed size!
! See e.g. PackFunction in folder _tools
```

Segmentation faults can result from not following this guideline (if your stack is indeed too small).

Stack: linux systems

```
ulimit -s unlimited
```

1.4 If-statements / comparisons

If-statements: logical comparison

```fortran
if(my_logical.eq.your_logical) print*,'bad'
if(my_logical.eq..true.) print*,'bad'
if(my_logical.eq..false.) print*,'bad'
```

```fortran
if(my_logical.eqv.your_logical.) print*,'good'
if(my_logical) print*,'good'
if(.not.my_logical) print*,'good'
```

For logical comparison `.eqv.` should be used. Most of the time, it can be omitted.

If-statements: real equality comparison

```fortran
if(x == 12.0_MK) ! dangerous real comparison



!
```

```fortran
if(.not.(abs(x-12.0_MK)>0.0_MK))      ! ex1
!
if(precision_equal(x,12.0_MK))        ! ex1b
!
if(abs(x-12.0_MK)<precision(1.0_MK)) ! ex2
```

In the above, ex1 is alright but not so readable. Ex1b uses a wrapped function that can be placed e.g. in the `SupportPrecision` module. Ex2 is more "physical": x and 12 are compared assuming the typical physical scale of the problem is 1. Indeed, the machine precision for a given real value is different depending on the real kind (here MK) and the absolute value.

If-statements: there is no assumed order of evaluation (example here for optional argument)

```fortran
if (present(x) .and. x>0) then
   ! Compiler might evaluate x>0 first
   ! =>Segfault if x is not present
endif
```

```fortran
if (present(x)) then
  if (x>0) then ! safe
  endif
endif
```

1.5 Do loops and memory order

Do loops: first index should runs the fastest to respect memory order

```
do i=1,n
    do j=1,m ! bad, j run the fastest
        a(i,j)=1.0_MK
    enddo
enddo
```

```
do j=1,m
    do i=1,n ! good
        a(i,j)=1.0_MK
    enddo
enddo
```

Memory: typical array dimensions for 3D geometry

```
real(MK), dimension(n,3)  :: Points ! bad
```

```
real(MK), dimension(3,n)  :: Points ! ok
```

Do loops: iteration on reals are bad practice

```
do x=0._MK, 10._MK, 0.1_MK
enddo
```

```
! Bad practice
! use loop on integer instead
```

1.6 Derived types

Derived types: use initializations, especially for pointers, always =>null() them

```
type T_mytype
  real(MK), pointer :: p
  integer :: i
end type
```

```
type T_mytype
  real(MK), pointer :: p=>null() !< always!
  integer :: i=0 !< safe to rely on it
end type
```

Components initialization is standard.

Derived types: component access

```
T.i = 0 ! . is not standard
```

```
T%i = 0  ! % is OK
```

Derived types: deallocate the components before the parent

```
type T_mytype
  real(MK), pointer :: p=>null()
end type
![...]
type(T_mytype), pointer :: t
nullify(t)
![...]
if (associated(t)) then

   deallocate(t) ! potential memory loss
endif
```

```
type T_mytype
  real(MK), pointer :: p=>null()
end type
![...]
type(T_mytype), pointer :: t
nullify(t)
![...]
if (associated(t)) then
    if (associated(t%p)) deallocate(t%p)
    deallocate(t) ! fine
endif
```

Derived types: automatic code generation
A bit of advertisement here, `simple-fortran-parser` can generate automatic code for derived types (like read-/write to binary, init/dealloc).

1.7 Characters

Characters: use the `len` specification

```
character*16 :: s ! not standard
```

```
character(len=16) :: s
```

Characters: use the `len` specification for arguments

```
function f(s)
  character*(*) :: s ! akward character array
end
```

```
function f(s)
  character(len=*), intent(in) :: s
end
```

Characters array: it's best to used fixed length for old compilers

```fortran
character(len=20), dimension(5)              :: strings  ! fine
character(len=20), dimension(:), allocatable :: strings2 ! fine
character(len=:),  dimension(:), allocatable :: strings3 ! fine but not for old compilers
strings(1)='a'
print*,string(1)(1:20)
! Allocation:
! allocate(strings2(size(strings,1),size(strings,2)))                   ! WRONG
allocate(character(len=len(strings(1))) :: strings2(size(strings,1)))! GOOD
```

Characters: retrieving a string from C

```fortran
subroutine string_switch(s_c) BIND(C, NAME='string_switch')
    use SupportPrecision, only: C_CHAR
    use CStrings,         only: cstring2fortran, fortranstring2c ! see folder _tools
    ! Argument
    character(kind=C_CHAR,len=1),dimension(*),intent(inout) :: s_c !< c string
    ! Variable
    character(len=255) :: s_f    !< fortran string
    ! [...]
    call cstring2fortran(s_c,s_f)
    ! [...]
    s_f='fortran'
    call fortranstring2c(s_f,s_c)
end
```

1.8 Arrays

Arrays: array construct with double dot is not standard

```fortran
integer, dimension(10) :: x

x=[1:10] ! not standard




!
```

```fortran
integer, dimension(10) :: x
integer :: i
x=[ (i,i=1,10) ] ! allright
x=(/(i,i=1,10)/) ! even more portable

do i=1,10
    x(i)=i ! readable, less bug
enddo
```

Unroll loops for large arrays

```fortran
M(1:3,1:n)=0.0_MK ! n is a large number


!
```

```fortran
! Unrolled loop (segfault observed otherwise)
do i=1,n
  M(1:3,i)=0.0_MK
enddo
```

Depending on the compiler and the compiler version, the code on the left may result in a segmentation fault without obvious reason, this can be hard to debug. Unrolling loops when manipulating large arrays is highly recommended. See also the stack pitfalls subsection 1.3

1.9 File IO

Unit value: don't use a fixed value

```fortran


open(99, ...) ! what if 99 is already opened?
read(99)
```

```fortran
use MainIO, only: get_free_unit()
iunit=get_free_unit()
open(iunit, ...)
read(iunit)
```

MainIO is defined in `_tools`

Binaries with direct access (e.g. Mann box): watch for the record length, wrap it in a module!

```fortran
!
open(iunit,file='u',recl=1,& !no standard
    access='direct',form='unformatted',&
    status='old')
```

```fortran
use SupportCompiler, only: RECORD_LENGTH
open(iunit,file='u',recl=RECORD_LENGTH,&
    access='direct',form='unformatted',&
    status='old')
```

Unfortunately, there is no standard for what recl should be. For intel and compaq by default `recl=1`. For gfortran (or intel with the flag `-assume byterecl`) `recl=4`

Binaries with stream access (e.g. VTK bin): not available on old compilers

```
open(iunit,'a.dat',form='UNFORMATTED',&
     access = 'stream', action = 'WRITE',&
     convert='BIG_ENDIAN')
```

```
use SupportCompiler, only: open_stream_write
call open_stream_write('a.dat')
!
```

Namelists: fine for derived types, but no pointers or allocatable.

```
    type T_RandomVar ! No pointers or allocatables
        character(len=56)    :: sname = ''
        real(MK)             :: value = 0._MK
        type(T_Stats)        :: stats ! No pointers or allocatables
    end type
    type T_Stats ! No pointers or allocatables
        real(MK), dimension(4)   :: moments = (/0._MK,0._MK,0._MK,0._MK/)
    end type
    ![...]
    type(T_RandomVar) :: RandomVar
    ![...]
    namelist/RandomVarInputs/RandomVar
    read(iunit,RandomVarInputs,iostat=ierr)
```

STOP and return status:

```
    STOP -1 ! not supported by Compaq
```

```
    !
```

1.10 Precision

Precision: in general, use a custom module

```

real*8
integer(int_ptr_kind())
```

```
use SupportPrecision, only: MK, PK
real(MK)
integer(PK)
```

The syntax `*8` is depreciated. It is convenient if you need a real that takes exactly 8 bytes, but still, it's depreciated (see next paragraph). Note that `real*8` and `real(8)` have no reason to be the same (the kinds are compiler dependent). `int_ptr_kind` is convenient to support multiple architecture (32/64bits) but is not standard (hence the SupportPrecision module).

Precision: If you really want to precise the size in bytes (8 bit)

```

real*4, real*8,
integer*4, integer*8
```

```
use iso_fortran_env
real(REAL32), real(REAL64)
integer(INT32), real(INT64)
```

The syntax `*4` is depreciated. The `iso_fortran_env` module is not available on old compilers => Use a SupportCompiler module wrapped in a SupportPrecision module. See `_support/SupportPrecision.f90` subsection 3.2

Precision: If you need to communicate with C (recommended for DLLs)

```

real
double precision
integer
character
logical
```

```
use iso_c_binding
real(C_FLOAT)
real(C_DOUBLE)
integer(C_INT)
character(kind=C_CHAR)
logical(C_BOOL)
```

The `iso_c_binding` module is not available on old compilers => Use a SupportCompiler module wrapped in a SupportPrecision module. See `_support/SupportPrecision.f90` subsection 3.2

Precision: use explicit type conversions (with compiler warnings)

```
!
real*4           :: x
double precision :: y
![...]
y = x  !implicit type conversion
```

```
use SupportPrecision, only: MPI_DOUBLE, MK
real(MK)          :: x
real(MPI_DOUBLE)  :: y
![...]
y = real(x, MPI_DOUBLE) ! explicit conversion
```

1.11 Operating System and filesystem

A lot of fortran builtin routines are cross-platform. The main problems can be found when creating directories and inquiring about files. Compilers have some non-standards extensions. Cross platform solutions can easily be implemented. The solution advised here is to put OS-specific parameters (like commands, and slash) in a module

SupportSystem (see folder _support) which is then included by a FileSystem module (in folder _tools).

Checking if a file exist: do not use stat, it's not standard, use the old inquire

```fortran
integer :: iFileExist
iFileExist=stat(filename,info_array)
if (iFileExist/=0) then
 ! file does not exist
else
  ! file exists
endif
!
```

```fortran
logical :: bFileExist
inquire(file=filename, exist=bFileExist)
if (.not.bFileExist) then
 ! file does not exist
else
  ! file exists
endif

! Or even better: use a wrapped function
use FileSystem, only: file_exists !see  _tools
if(.not.file_exists(filename)) then
! [..]
```

1.12 Makefile

Makefiles are convenient to compile code on multiple platforms using different compilers and libraries. The current repository contains examples in the _\mkf folder. MakefileOS.mk attempts to detect the OS and architecure and unify OS-specific parameters. MakefileFortran.mk attempts to unify Fortan compiler flags.

1.13 Preprocessor

This section presents some generalities about preprocessors and examples of cases where the preprocessor directives can be replaced by wrapped code in separate modules. Examples where this approach is not possible are found in subsection 1.14. As mentioned in the introduction, relying on a preprocessor is not really portable and the wrapped approach should be preferred whenever possible. Also, the C-preprocessor being the most used one, it's best to use this one.

Macros
The most used feature is something of the form `if defined MACRO then ... endif`. The string MACRO is defined by the compiler or the user. Macros are defined on the command line using -DMACRO. Since it is a compiler variable intended to be defined in the entire scope of the program, a convention is to surround the macro name with double underscores, e.g. __dtu__ (see POSIX standard and ANSI-C standards). For a list of predefined macros for Compilers/OS/Archi: http://sourceforge.net/p/predef/wiki/Home/
Given the variability of definitions, it is advised to always (re)define the macros that your are using on the command line: e.g. -D__linux__, -D_WIN32, D__intel__, D__compaq__,D__amd64__,D__i386__. For instance -D__linux__ is not defined by gfortran on linux.

C-Preprocessor
The C-Preprocessor is supported by: intel, gfortran, compaq, sun, pgi portland.
Macros are **case sensitive**.

gfortran: -cpp: use C-preprocessor, -E -dM: show preprocessor macros
ifort : -fpp: use C-preprocessor, -E -dM -dryrun: show preprocessor macros
sun : -xpp: use C-preprocessor
pgf90: -Mpreproc : use C-preprocessor

DEC-Preprocessor
The DEC-Preprocessor is supported by: intel and compaq.
Macros are **case in-sensitive**.

GNU-Preprocessor
The GNU-Preprocessor is supported by: gfortran.
It is only used to define ATTRIBUTES, it doesn't support `if defined`.
`!GCC$ ATTRIBUTES DLLEXPORT :: init`

Preprocessor: Examples where preprocessor directives can be removed

```fortran
!DEC$ IF DEFINED(__HDF5__)
   call hdf5_init()
!DEC$ END IF
```

```fortran
use SupportHDF5, only: hdf5_init()
call hdf5_init()
!
```

```fortran
! C preprocessor
#if defined _WIN32
  call mkdir_windows('fold')
#elif defined __unix__
  call mkdir_linux('fold')
#endif
```

```fortran
! FileSystem is found in _tools, it uses
    SupportSystem
use FileSystem, only: system_mkdir

call system_mkdir('fold')
!
```

A possible variation (not as clean):

```fortran
! C preprocessor
#if defined _WIN32
  !do something
#elif defined __unix__
  !do another thing
#endif
```

```fortran
use SupportSystem, only: OSNAME
if (OSNAME(1:7)=='windows') then
    !do something
elseif if (OSNAME(1:5)=='linux') then
    !do another thing
endif
```

The if statements will not affect performances since they relies on compile time constants. Compiler optimization should remove dead-code and dead-if statements. This method cannot be used around "use" statements or routines declarations.

1.14 DLLs, cross-language interoperability

-For C-strings see subsection 1.7 and the file `CStrings.f90`
-For C-types see subsection 1.10

Procedure names/alias: bind(C) is really convenient, but not supported by Compaq

```fortran
subroutine init(array1)
!DEC$ ATTRIBUTES C, ALIAS:'init'::init
```

```fortran
subroutine init(array1) bind(C,name='init')
!
```

The code on the left is standard 2003, cross-platform, cross-compiler, preprocessor-independent and just easy to use. The only down side is that the Compaq compiler does not support it. Note: for dllexport it makes it easier if the subroutine name and the bind-name are the same. NOTE: An array dummy argument of a BIND(C) procedure must be an explicit shape (`dimension(n)`, `dimension(n,m)`) or assumed size array (`dimension(*)`, `dimension(lda,*)`). If it's an assumed size array, the size of the array is not computable and thus the upper bound should always be precised, i.e. `A(1,:)` should be something like `A(1,1:n)`

Procedure exports for dll: the problem of the def file

```fortran
subroutine init(array1) bind(c,name='init')
!DEC$ IF .NOT. DEFINED(__LINUX__)
!DEC$ ATTRIBUTES DLLEXPORT ::init
!GCC$ ATTRIBUTES DLLEXPORT ::init
!DEC$ END IF
```

```fortran
subroutine init(array1) bind(C,name='init')



! Generate the def file yourself
```

The code above is not compatible with old compilers like Compaq due to the `bind(C)` directive. The code on the left should work for Intel and GCC but it relies on preprocessor directives. The code on the right is clean and portable. It requires more work on the windows users since the `.def` file needs to be written. A dll interface is not expected to change that often, so the work is not that heavy. The python tool `simple-fortran-parser` can generate the `.def` automatically based on all the `bind(C)` subroutines it finds in the code.

Procedure exports for dll: a more-or-less portable way

```fortran
#if defined OLD_COMPILER
subroutine init(array1)
!DEC$ ATTRIBUTES C, ALIAS:'init'::init
#else
subroutine init(array1) bind(c,name='init')
#endif
!DEC$ IF .NOT. DEFINED(__LINUX__)
!DEC$ ATTRIBUTES DLLEXPORT ::init
!GCC$ ATTRIBUTES DLLEXPORT ::init
!DEC$ END IF
```

```fortran
#if defined OLD_COMPILER
subroutine init(array1)
!DEC$ ATTRIBUTES C, ALIAS:'init'::init
#else
subroutine init(array1) bind(c,name='init')
#endif
!
!
! Generate the def file yourself
!
```

The above should work with compaq,intel,gcc on windows and linux as long a C preprocessor flag is given to the compilers (i.e. `-fpp` or `-cpp`) and as long as the Compaq compiler defines the flag `-DOLD_COMPILER`. On linux with gfortran the `-Wno-attributes` could be use to avoid the warning.

2 Compilers

Preprocessor directives defined by compilers to identify themselves (see `_unit_tests/preproc`): `__INTEL_COMPILER`, `__GFORTRAN__`, `_DF_VERSION_`

Compaq
Setup the path:
`call dfvars.bat` (32 bits)
The script is likely located in:
`C:\Program Files\Microsoft Visual Studio\Df 98\BIN`

If you see messages like "cannot find dfort.lib" then you probably didnt run dfvars.bat.
If you see messages like "LINK: fatal error .. /ignore:505" then you probably didnt run dfvars.bat.

Visual studio C compiler
Setup the path:
`call vcvarsall.bat x86` (32bit)
`call vcvarsall.bat amd64` (64 bits)
The script is likely located in:
`C:\Program Files (x86)\Microsoft Visual Studio 10.0\VC`

Intel fortran
Setup the path:
`call ifortvars.bat ia32 vs2010` (32 bits)
`call ifortvars.bat intel64 vs2010` (64 bits)
The script is likely located in:
`C:\Program Files (x86)\Intel \ComposerXE -2011\bin`

3 Support files

3.1 Compiler (example for intel)

../_support/SupportCompiler_intel.f90

```fortran
module SupportCompiler
    implicit none
    !
    integer, parameter :: IPTRK=int_ptr_kind() !< for pointers
    integer, parameter :: RECORD_LENGTH=1      !< for direct access binaries
    !
    integer, parameter :: ISTR_LEN = 64 !< parameter for ease of comparison of parameter-strings
    character(len=ISTR_LEN), parameter :: FORTRAN_COMPILER='ifort'

end module
```

3.2 Precision

../_support/SupportPrecision.f90

```fortran
module SupportPrecision
    ! Compiler interface to iso_c_binding
    use SupportISO, only: C_FLOAT, C_DOUBLE, C_CHAR, C_INT, C_BOOL
    use SupportISO, only: C_INTPTR_T, C_NULL_FUNPTR
    ! Compiler interface to iso_fortran_env
    use SupportISO, only: REAL32, REAL64, INT32, INT64
    ! Compiler interface to int_ptr_kind
    use SupportCompiler, only: IPTRK
    !
    integer, parameter :: R4 = REAL32    ! 32 bits
    integer, parameter :: R8 = REAL64    ! 64 bits
    integer, parameter :: SP = kind(1e0)! "Single precision"
    integer, parameter :: DP = kind(1d0)! "Double precision"
    integer, parameter :: MK = C_DOUBLE ! MK stands for My Kind

contains
    ! --------------------------------------------------------------------------------
    ! ---
    ! --------------------------------------------------------------------------------
```

```fortran
    subroutine print_precision_kinds()
        print*,'C_INT        ',C_INT
        print*,'C_FLOAT      ',C_FLOAT
        print*,'C_DOUBLE     ',C_DOUBLE
        print*,'C_CHAR       ',C_CHAR
        print*,'C_BOOL       ',C_BOOL
        print*,'INT32        ',INT32
        print*,'INT64        ',INT64
        print*,'REAL32       ',REAL32
        print*,'REAL64       ',REAL64
        print*,'SP           ',kind(1e0)
        print*,'DP           ',kind(1d0)
        print*,'C_INTPTR_T   ',C_INTPTR_T
        print*,'C_NULL_FPTR  ',C_NULL_FUNPTR
        print*,'IPTRK        ',IPTRK
    end subroutine


    ! Below we have functions for MK, DP, SP (no interface is used because redundancy is possible
    )
    ! --------------------------------------------------------------------------------
    ! --- MK, default
    ! --------------------------------------------------------------------------------
    logical function precision_equal(x,y) result(b)
        real(MK), intent(in) :: x,y
        b=.not.precision_different(x,y)
    end function

    logical function precision_different(x,y) result(b)
        real(MK), intent(in) :: x,y
        b= abs(x -y) >0.0_MK
    end function

    ! --------------------------------------------------------------------------------
    ! ---  Double precision
    ! --------------------------------------------------------------------------------
    logical function precision_equal_dp(x,y) result(b)
        real(DP), intent(in) :: x,y
        b=.not.precision_different_dp(x,y)
    end function

    logical function precision_different_dp(x,y) result(b)
        real(DP), intent(in) :: x,y
        b= abs(x -y) >0.0_MK
    end function

    ! --------------------------------------------------------------------------------
    ! ---  Single precision
    ! --------------------------------------------------------------------------------
    logical function precision_equal_sp(x,y) result(b)
        real(SP), intent(in) :: x,y
        b=.not.precision_different_sp(x,y)
    end function

    logical function precision_different_sp(x,y) result(b)
        real(SP), intent(in) :: x,y
        b= abs(x -y) >0.0_MK
    end function

end module
```

3.3 Sytem (example for linux)

../_support/SupportSystem_linux.f90

```fortran
!> Contains Parameters/Data that are system/architecture specific, autogenerated by Makefile
module SupportSystem
    implicit none
    !LINUX
    character(len=10), parameter :: OSNAME = "linux"
    character(len=1),  parameter :: SLASH  = '/'  !< Path separator.
    character(len=1),  parameter :: BADSLASH  = '\'  !< Bad slash
    character(len=1),  parameter :: SWITCH = '-'  !< switch for command-line options.
    character(len=20), parameter :: COPY          = "cp "
    character(len=20), parameter :: RENAME        = "mv "
    character(len=20), parameter :: REMOVE        = "rm -f "
    character(len=10), parameter :: MKDIR         = "mkdir -p "
```

```
    character(len=20), parameter :: RMDIR          = "rmdir "
    character(len=20), parameter :: FILELIST       = "ls "      ! Only file names!
    character(len=20), parameter :: REDIRECT       = ">"
    character(len=20), parameter :: TEMPDIR        = "/tmp/"

    character(len=20), parameter :: BG_CMD_PREFIX  = "nohup"
    character(len=20), parameter :: BG_CMD_SUFFIX  = "&"
    character(len=20), parameter :: SUPPRESS_MSG   = "2>/dev/null"
end module
```

3.4 C Strings

../_tools/CStrings.f90

```
module CStrings
    implicit none
contains
    subroutine cstring2fortran(s_c,s)
        use SupportPrecision, only: C_CHAR
        character(kind=C_CHAR,len=1),dimension(*),intent(in) :: s_c
        character(len=*),intent(inout):: s
        integer :: i
        loop_string: do i=1,len(s)
            if ( s_c(i) == CHAR(0) ) then
                exit loop_string
            else
                s(i:i) = s_c(i)
            end if
        end do loop_string

        if(i==1) then
            s=''
        else
            s = s(1:(i-1))
            s = trim(s)
        endif
    end subroutine

    subroutine fortranstring2c(s_f,s_c,n)
        use SupportPrecision, only: C_CHAR
        character(len=*),intent(in):: s_f
        character(kind=C_CHAR,len=1),dimension(*),intent(inout) :: s_c
        integer, intent(out), optional :: n
        integer :: i
        loop_string: do i=1,len(s_f)
            if ( s_f(i:i) == CHAR(0) ) then
                exit loop_string
            else
                s_c(i) = s_f(i:i)
            end if
        end do loop_string
        if(present(n))then
            n=i-1
        endif
    end subroutine
end module
```