LIB_VTK_IO

VTK INPUT/OUTPUT FORTRAN LIBRARY

Version 0.2

AUTHOR
STEFANO ZAGHI

CO-AUTHORS
ENRICO CAVALLINI AND RENATO N. ELIAS

07-01-2008

CONTENTS

Part I

COMPILE AND INSTALL LIB_VTK_IO

Part II

LIB_VTK_IO API

1

MODULE LIB_VTK_IO

LIB_VTK_IO is a library of functions for Input and Output pure fortran data (both ascii and binary) in VTK format.

The VTK standard can be separated into two main catagories: the VTK LEGACY STANDARD and the VTK XML STANDARD. The latter is more powerful and will has a stronger support from VTk comunity than legacy standard; XML file format would to be preferred despite the legacy one.

At the present only a few functions of the final library have been implemented. The InPut functions are totaly absent, but the OutPut functions are almost complete (the "polydata" functions are the only missing).

The functions actually present are:

Functions for Legacy VTK file format

1. VTK_INI
2. VTK_GEO
3. VTK_CON
4. VTK_DAT
5. VTK_VAR
6. VTK_END

Functions for XML VTK file format

1. VTK_INI_XML
2. VTK_GEO_XML

3. VTK_CON_XML
4. VTK_DAT_XML
5. VTK_VAR_XML
6. VTK_END_XML

VTK_GEO INTERFACE

```
interface VTK_GEO
  module procedure VTK_GEO_UNST_R8, & ! real(R8P) UNSTRUCTURED_GRID
                   VTK_GEO_UNST_R4, & ! real(R4P) UNSTRUCTURED_GRID
                   VTK_GEO_STRP_R8, & ! real(R8P) STRUCTURED_POINTS
                   VTK_GEO_STRP_R4, & ! real(R4P) STRUCTURED_POINTS
                   VTK_GEO_STRG_R8, & ! real(R8P) STRUCTURED_GRID
                   VTK_GEO_STRG_R4, & ! real(R4P) STRUCTURED_GRID
                   VTK_GEO_RECT_R8, & ! real(R8P) RECTILINEAR_GRID
                   VTK_GEO_RECT_R4    ! real(R4P) RECTILINEAR_GRID
endinterface
```

VTK_VAR INTERFACE

```
interface VTK_VAR
  module procedure VTK_VAR_SCAL_R8, & ! real(R8P) scalar
                   VTK_VAR_SCAL_R4, & ! real(R4P) scalar
                   VTK_VAR_SCAL_I4, & ! integer(I4P) scalar
                   VTK_VAR_VECT_R8, & ! real(R8P) vectorial
                   VTK_VAR_VECT_R4, & ! real(R4P) vectorial
                   VTK_VAR_VECT_I4, & ! integer(I4P) vectorial
                   VTK_VAR_TEXT_R8, & ! real(R8P) vectorial (texture)
                   VTK_VAR_TEXT_R4    ! real(R4P) vectorial (texture)
endinterface
```

VTK_GEO_XML INTERFACE

```
interface VTK_GEO_XML
  module procedure VTK_GEO_XML_STRG_R4, & ! real(R4P) StructuredGrid
                   VTK_GEO_XML_STRG_R8, & ! real(R8P) StructuredGrid
                   VTK_GEO_XML_RECT_R8, & ! real(R8P) RectilinearGrid
                   VTK_GEO_XML_RECT_R4, & ! real(R4P) RectilinearGrid
                   VTK_GEO_XML_UNST_R8, & ! real(R8P) UnstructuredGrid
                   VTK_GEO_XML_UNST_R4, & ! real(R4P) UnstructuredGrid
                   VTK_GEO_XML_CLOSEP     ! closing tag "Piece" function
endinterface
```

VTK_VAR_XML INTERFACE

```
interface VTK_VAR_XML
  module procedure VTK_VAR_XML_SCAL_R8, & ! real(R8P) scalar
                   VTK_VAR_XML_SCAL_R4, & ! real(R4P) scalar
                   VTK_VAR_XML_SCAL_I8, & ! integer(I8P) scalar
                   VTK_VAR_XML_SCAL_I4, & ! integer(I4P) scalar
                   VTK_VAR_XML_SCAL_I2, & ! integer(I2P) scalar
                   VTK_VAR_XML_SCAL_I1, & ! integer(I1P) scalar
                   VTK_VAR_XML_VECT_R8, & ! real(R8P) vectorial
                   VTK_VAR_XML_VECT_R4, & ! real(R4P) vectorial
                   VTK_VAR_XML_VECT_I8, & ! integer(I4P) vectorial
                   VTK_VAR_XML_VECT_I4, & ! integer(I4P) vectorial
                   VTK_VAR_XML_VECT_I2, & ! integer(I4P) vectorial
                   VTK_VAR_XML_VECT_I1    ! integer(I4P) vectorial
endinterface
```

LIB_VTK_IO has a small set of internal variables and parameters some of which have public visibility.

The LIB_VTK_IO uses a partable kind parameters for real and integer variables. The following are the kind parameters used: these parameters are public and their use is strong encouraged.

Real precision definitions:

LIB_VTK_IO VARIABLES

```
integer, parameter:: R16P = selected_real_kind(33,4931) ! 33 digits, range [±10^-4931, ±10^+4931 - 1]
integer, parameter:: R8P  = selected_real_kind(15,307)  ! 15 digits, range [±10^-307 , ±10^+307  - 1]
integer, parameter:: R4P  = selected_real_kind(6,37)    ! 6  digits, range [±10^-37  , ±10^+37   - 1]
integer, parameter:: R_P  = R8P                         ! default real precision
```

Integer precision definitions:

LIB_VTK_IO VARIABLES

```
integer, parameter:: I8P = selected_int_kind(18) ! range [-2^63, +2^63 - 1]
integer, parameter:: I4P = selected_int_kind(9)  ! range [-2^31, +2^31 - 1]
integer, parameter:: I2P = selected_int_kind(4)  ! range [-2^15, +2^15 - 1]
integer, parameter:: I1P = selected_int_kind(2)  ! range [-2^7 , +2^7  - 1]
integer, parameter:: I_P = I4P                   ! default integer precision
```

Besides the kind parameters there are also the format parameters useful for writing in a well-ascii-format numeric variables. Also these parameters are public.

Real output formats:

LIB_VTK_IO VARIABLES

```
character(10), parameter:: FR16P = '(E41.33E4)' ! R16P output format
character(10), parameter:: FR8P  = '(E23.15E3)' ! R8P output format
character(9),  parameter:: FR4P  = '(E14.6E2)'  ! R4P output format
character(10), parameter:: FR_P  = '(E23.15E3)' ! R_P output format
```

Integer output formats:

LIB_VTK_IO VARIABLES

```
character(5), parameter:: FI8P  = '(I21)'              ! I8P output format
character(5), parameter:: FI4P  = '(I12)'              ! I4P output format
character(4), parameter:: FI2P  = '(I7)'               ! I2P output format
character(4), parameter:: FI1P  = '(I5)'               ! I1P output format
character(5), parameter:: FI_P  = '(I12)'              ! I_P output format
```

LIB_VTK_IO uses a small set of internal variables that are private (not accessible from the outside). The following are private variables:

LIB_VTK_IO VARIABLES

```
integer(I4P), parameter:: maxlen        = 500          ! max number of characters os static string
character(1), parameter:: end_rec       = char(10)     ! end-character for binary-record finalize
integer(I4P), parameter:: f_out_ascii   = 0            ! ascii-output-format parameter identifier
integer(I4P), parameter:: f_out_binary  = 1            ! binary-output-format parameter identifier
integer(I4P)::            f_out         = f_out_ascii  ! current output-format (initialized to ascii format)
character(len=maxlen)::   topology                     ! mesh topology
integer(I4P)::            Unit_VTK                     ! internal logical unit
integer(I4P)::            Unit_VTK_Append              ! internal logical unit for raw binary XML append file
integer(I4P)::            N_Byte                       ! number of byte to be written/read
real(R8P)::               tipo_R8                      ! prototype of R8P real
real(R4P)::               tipo_R4                      ! prototype of R4P real
integer(I8P)::            tipo_I8                      ! prototype of I8P integer
integer(I4P)::            tipo_I4                      ! prototype of I4P integer
integer(I2P)::            tipo_I2                      ! prototype of I2P integer
integer(I1P)::            tipo_I1                      ! prototype of I1P integer
integer(I4P)::            ioffset                      ! offset pointer
integer(I4P)::            indent                       ! indent pointer
```

In the following chapters there is the API reference of all functions of LIB_VTK_IO .

2

AUXILIARY FUNCTIONS

Contents

LIB_VTK_IO uses two auxiliary functions that are not connected with the VTK standard. These functions are private and so they cannot be called outside the library.

2.1 FUNCTION GETUNIT

GETUNIT SIGNATURE

```
function GetUnit() result(Free_Unit)
```

The GetUnit function is used for getting a free logic unit. The users of LIB_VTK_IO does not know which is the logical unit: LIB_VTK_IO handels this information without boring the users. The logical unit used is safe-free: if the program calling LIB_VTK_IO has others logical units used LIB_VTK_IO will never use these units, but will choice one that is free.

GETUNIT VARIABLES

```
integer(I4P)::  Free_Unit  ! free logic unit
integer(I4P)::  n1         ! counter
integer(I4P)::  ios        ! inquiring flag
logical(4)::    lopen      ! inquiring flag
```

The following is the code snippet of GetUnit function: the units 0, 5, 6, 9 and all non-free units are discarded.

GetUnit Code Snippet

```
Free_Unit = -1_I4P                                ! initializing free logic unit
n1=1_I4P                                          ! initializing counter
do
  if ((n1/=5_I4P).AND.(n1/=6_I4P).AND.(n1/=9_I4P)) then
    inquire (unit=n1,opened=lopen,iostat=ios)     ! verify logic units
    if (ios==0_I4P) then
      if (.NOT.lopen) then
        Free_Unit = n1                            ! assignment of free logic
        return
      endif
    endif
  endif
  n1=n1+1_I4P                                     ! updating counter
enddo
return
```

GetUnit function is private and cannot be called outside LIB_VTK_IO . If you are interested to use it change its scope to public.

2.2 FUNCTION UPPER_CASE

UPPER_CASE SIGNATURE

```
function Upper_Case(string)
```

The Upper_Case function converts the lower case characters of a string to upper case one. LIB_VTK_IO uses this function in order to achieve case-insensitive: all character variables used within LIB_VTK_IO functions are pre-processed by Uppper_Case function before these variables are used. So the users can call LIB_VTK_IO functions whitout pay attention of the case of the kwywords passed to the functions: calling the function VTK_INI with the string `E_IO = VTK_INI('Ascii',...)` or with the string `E_IO = VTK_INI('AscII',...)` is equivalent.

UPPER_CASE VARIABLES

```
character(len=*), intent(IN):: string      ! string to be converted
character(len=len(string)):: Upper_Case    ! converted string
integer::                     n1           ! characters counter
```

The following is the code snippet of Upper_Case function.

Upper_Case Code Snippet

```
Upper_Case = string
do n1=1,len(string)
  select case(ichar(string(n1:n1)))
  case(97:122)
    Upper_Case(n1:n1)=char(ichar(string(n1:n1))-32) ! Upper case conversion
  endselect
enddo
return
```

Upper_Case function is private and cannot be called outside LIB_VTK_IO . If you are interested to use it change its scope to public.

3

VTK LEGACY FUNCTIONS

Contents

3.1 FUNCTION VTK_INI

VTK_INI SIGNATURE

```
function VTK_INI(output_format,filename,title,mesh_topology) result(E_IO)
```

The VTK_INI function is used for initializing file. This function must be the first to be called.

VTK_INI VARIABLES

```
character(*), intent(IN)::  output_format  ! output format: ASCII or BINARY
character(*), intent(IN)::  filename       ! name of file
character(*), intent(IN)::  title          ! title
character(*), intent(IN)::  mesh_topology  ! mesh topology
integer(I4P)::              E_IO           ! Input/Output inquiring flag: 0 if IO is done, > 0 if IO is not done
```

The VTK_INI variables have the following meaning:

output_format indicates the "format" of output file. It can assume the following values:

A. *ascii* (it is case insensitive) → creating an ascii output file.
B. *binary* (it is case insensitive) → creating a binary (big_endian encoding) output file.

filename contains the name (with its path) of the output file.

title contains the title of the VTK dataset.

topology indicates the topology of the mesh and can assume the following values:

A. *STRUCTURED_POINTS*.
B. *STRUCTURED_GRID*.
C. *UNSTRUCTURED_GRID*.
D. *RECTILINEAR_GRID*.

E_IO contains the inquiring integer flag for error handling.

The following is an example of VTK_INI calling:

VTK_INI Calling

```
...
E_IO = VTK_INI('Binary','example.vtk','VTK legacy file','UNSTRUCTURED_GRID')
...
```

Note that the ".vtk" extension is necessary in the file name.

3.2 VTK_GEO

VTK_GEO is an interface to 8 different functions; there are 2 functions for each 4 different topologies actually supported: one function for mesh coordinates with R8P precision and one for mesh coordinates with R4P precision. This function must be called after VTK_INI. It saves the mesh geometry. The inputs that must be passed change depending on the topologies choiced. Not all VTK topologies have been implemented ("polydata" topologies are absent). The signatures for all implemented topologies are now reported.

3.2.1 VTK_GEO STRUCTURED POINTS

VTK_GEO STRUCTURED POINTS SIGNATURE

```
function VTK_GEO(Nx,Ny,Nz,Xo,Yo,Zo,Dx,Dy,Dz) result(E_IO)
```

The topology "structured points" is useful for structured grid with uniform discretization steps.

VTK_GEO STRUCTURED POINTS VARIABLES

```
integer(I4P),      intent(IN):: Nx    ! number of nodes in x direction
integer(I4P),      intent(IN):: Ny    ! number of nodes in y direction
integer(I4P),      intent(IN):: Nz    ! number of nodes in z direction
real(R8P or R4P),  intent(IN):: Xo    ! x coordinate of origin
real(R8P or R4P),  intent(IN):: Yo    ! y coordinate of origin
real(R8P or R4P),  intent(IN):: Zo    ! z coordinate of origin
real(R8P or R4P),  intent(IN):: Dx    ! space step in x
real(R8P or R4P),  intent(IN):: Dy    ! space step in y
real(R8P or R4P),  intent(IN):: Dz    ! space step in z
integer(I4P)::                  E_IO  ! Input/Output inquiring flag: 0 if IO is done, > 0 if IO is not done
```

Note that the variables `X0,Y0,Z0,Dx,Dy,Dz` can be passed both as 8-byte real kind and 4-byte real kind; the dynamic displacement interface will call the correct function. Mixing 8-byte real kind and 4-byte real kind is not allowed: be sure that all variables are 8-byte real kind or all are 4-byte real kind.

The VTK_GEO structured point variables have the following meaning:

Nx indicates the number of nodes in X direction.

Ny indicates the number of nodes in Y direction.

NZ indicates the number of nodes in Z direction.

X0 indicates the X value of coordinates system origin. It is a scalar.

Y0 indicates the Y value of coordinates system origin. It is a scalar.

Z0 indicates the Z value of coordinates system origin. It is a scalar.

Dx indicates the uniform grid step discretization in X direction. It is a scalar.

Dy indicates the uniform grid step discretization in Y direction. It is a scalar.

DZ indicates the uniform grid step discretization in Z direction. It is a scalar.

E_IO contains the inquiring integer flag for error handling.

The following is an example of VTK_GEO structured point calling:

VTK_GEO Structured Points Calling

```
...
integer(4):: Nx,Ny,Nz
real(8):: X0,Y0,Z0
real(8):: Dx,Dy,Dz
...
E_IO = VTK_GEO(Nx,Ny,Nz, &
               X0,Y0,Z0,Dx,Dy,Dz)
...
```

3.2.2 VTK_GEO STRUCTURED GRID

VTK_GEO STRUCTURED GRID SIGNATURE

```
function VTK_GEO(Nx,Ny,Nz,NN,X,Y,Z) result(E_IO)
```

The topology "structured grid" is useful for structured grid with non-uniform discretization steps.

VTK_GEO STRUCTURED GRID VARIABLES

```
integer(I4P),       intent(IN):: Nx        ! number of nodes in x direction
integer(I4P),       intent(IN):: Ny        ! number of nodes in y direction
integer(I4P),       intent(IN):: Nz        ! number of nodes in z direction
integer(I4P),       intent(IN):: NN        ! number of all nodes
real(R8P or R4P),   intent(IN):: X(1:NN)   ! x coordinates
real(R8P or R4P),   intent(IN):: Y(1:NN)   ! y coordinates
real(R8P or R4P),   intent(IN):: Z(1:NN)   ! z coordinates
integer(I4P)::                   E_IO      ! Input/Output inquiring flag: 0 if IO is done, > 0 if IO is not done
```

Note that the variables `X,Y,Z` can be passed both as 8-byte real kind and 4-byte real kind; the dynamic displacement interface will call the correct function. Mixing 8-byte real kind and 4-byte real kind is not allowed: be sure that all variables are 8-byte real kind or all are 4-byte real kind.

The VTK_GEO structured grid variables have the following meaning:

Nx indicates the number of nodes in X direction.

Ny indicates the number of nodes in Y direction.

NZ indicates the number of nodes in Z direction.

NN indicates the number of all nodes, $NN = Nx \cdot Ny \cdot Nz$.

X contains the X coordinates values of all nodes. It is a vector of $[1 : NN]$.

Y contains the Y coordinates values of all nodes. It is a vector of $[1 : NN]$.

Z contains the Z coordinates values of all nodes. It is a vector of [1 : NN].

E_IO contains the inquiring integer flag for error handling.

The following is an example of VTK_GEO structured grid calling:

VTK_GEO Structured Grid Calling

```
...
integer(4), parameter:: Nx=10,Ny=10,Nz=10
integer(4), parameter:: Nnodi=Nx*Ny*Nz
real(8):: X(1:Nnodi),Y(1:Nnodi),Z(1:Nnodi)
...
E_IO = VTK_GEO(Nx,Ny,Nz,Nnodi,X,Y,Z)
...
```

3.2.3 VTK_GEO RECTILINEAR GRID

VTK_GEO RECTILINEAR GRID SIGNATURE

```
function VTK_GEO(Nx,Ny,Nz,X,Y,Z) result(E_IO)
```

The topology "rectilinear grid" is useful for structured grid with non-uniform discretization steps even in generalized coordinates.

VTK_GEO RECTILINEAR GRID SIGNATURE

```
integer(I4P),        intent(IN):: Nx       ! number of nodes in x direction
integer(I4P),        intent(IN):: Ny       ! number of nodes in y direction
integer(I4P),        intent(IN):: Nz       ! number of nodes in z direction
real(R8P or R4P),    intent(IN):: X(1:Nx)  ! x coordinates
real(R8P or R4P),    intent(IN):: Y(1:Ny)  ! y coordinates
real(R8P or R4P),    intent(IN):: Z(1:Nz)  ! z coordinates
integer(I4P)::                     E_IO     ! Input/Output inquiring flag: 0 if IO is done, > 0 if IO is not done
```

Note that the variables `X,Y,Z` can be passed both as 8-byte real kind and 4-byte real kind; the dynamic displacement interface will call the correct function. Mixing 8-byte real kind and 4-byte real kind is not allowed: be sure that all variables are 8-byte real kind or all are 4-byte real kind.

The VTK_GEO rectilinear grid variables have the following meaning:

Nx indicates the number of nodes in X direction.

Ny indicates the number of nodes in Y direction.

Nz indicates the number of nodes in Z direction.

X contains the X coordinates values of nodes. It is a vector of $[1 : Nx]$.

Y contains the Y coordinates values of nodes. It is a vector of $[1 : Ny]$.

Z contains the Z coordinates values of nodes. It is a vector of $[1 : Nz]$.

E_IO contains the inquiring integer flag for error handling.

The following is an example of VTK_GEO rectilinear grid calling:

VTK_GEO Rectilinear Grid Calling

```
...
integer(4), parameter:: Nx=10,Ny=20,Nz=30
real(4):: X(1:Nx),Y(1:Ny),Z(1:Nz)
...
E_IO = VTK_GEO(Nx,Ny,Nz,X,Y,Z)
...
```

3.2.4 VTK_GEO UNSTRUCTURED GRID

VTK_GEO UNSTRUCTURED GRID SIGNATURE

```
function VTK_GEO(Nnodi,X,Y,Z) result(E_IO)
```

The topology "unstructured grid" is necessary for unstructured grid, the most general mesh format. This topology is also useful for scructured mesh in order to save only a non-structured clip of mesh.

VTK_GEO UNSTRUCTURED GRID VARIABLES

```
integer(I4P),        intent(IN):: NN       ! number of nodes
real(R8P or R4P),    intent(IN):: X(1:NN)  ! x coordinates of all nodes
real(R8P or R4P),    intent(IN):: Y(1:NN)  ! y coordinates of all nodes
real(R8P or R4P),    intent(IN):: Z(1:NN)  ! z coordinates of all nodes
integer(I4P)::                     E_IO     ! Input/Output inquiring flag: 0 if IO is done, > 0 if IO is not done
```

Note that the variables `X,Y,Z` can be passed both as 8-byte real kind and 4-byte real kind; the dynamic displacement interface will call the correct function. Mixing 8-byte real kind and 4-byte real kind is not allowed: be sure that all variables are 8-byte real kind or all are 4-byte real kind.

The VTK_GEO unstructured grid variables have the following meaning:

NN indicates the number of all nodes.

X contains the X coordinates values of nodes. It is a vector of $[1 : NN]$.

Y contains the Y coordinates values of nodes. It is a vector of $[1 : NN]$.

Z contains the Z coordinates values of nodes. It is a vector of $[1 : NN]$.

E_IO contains the inquiring integer flag for error handling.

The following is an example of VTK_GEO unstructured grid calling:

VTK_GEO Unstructured Grid Calling

```
...
integer(4), parameter:: NN=100
real(4):: X(1:NN),Y(1:NN),Z(1:NN)
...
E_IO = VTK_GEO(NN,X,Y,Z)
...
```

In order to use the "unstructured grid" it is necessary to save also the "connectivity" of the grid. The connectivity must be saved with the function VTK_CON.

3.3 FUNCTION VTK_CON

VTK_CON SIGNATURE

```
function VTK_CON(NC, connect, cell_type) result(E_IO)
```

This function MUST be used when unstructured grid is used. It saves the connectivity of the unstructured mesh.

VTK_CON VARIABLES

```
integer(I4P), intent(IN)::  NC                ! number of cells
integer(I4P), intent(IN)::  connect(:)        ! mesh connectivity
integer(I4P), intent(IN)::  cell_type(1:NC)   ! VTK cell type
integer(I4P)::              E_IO              ! Input/Output inquiring flag: 0 if IO is done, > 0 if IO is not done
character(len=maxlen)::     s_buffer          ! buffer string
integer(I4P)::              ncon              ! dimension of connectivity vector
```

The VTK_CON variables have the following meaning:

NC indicates the number of all cells.

connect contains the connectivity of the mesh. It is a vector.

cell_type contains the type of every cells. It is a vector of $[1 : NC]$.

E_IO contains the inquiring integer flag for error handling.

The vector CONNECT must follow the VTK legacy standard. It is passed as ASSUMED-SHAPE array because its dimensions is related to the mesh dimensions in a complex way. Its dimensions can be calculated by the following equation:

$$dc = NC + \sum_{i=1}^{NC} nvertex_i \tag{3.1}$$

where dc is connectivity vector dimension and $nvertex_i$ is the number of vertices of i^{th} cell. The VTK legacy standard for the mesh connectivity is quite obscure at least at first sight. It is more simple analizing an example. Suppose we have a mesh composed by 2 cells, one hexahedron (8 vertices) and one pyramid with square basis (5 vertices); suppose that the basis of pyramid is constitute by a face of the hexahedron and so the two cells share 4 vertices. The equation 3.1 gives $dc = 2+8+5 = 15$; the connectivity vector for this mesh can be:

Connectivity vector example for VTK legacy standard

```
! first cell
connect(1)  = 8  => number of vertices of 1 cell
connect(2)  = 0  => identification flag of 1 vertex of 1 cell
connect(3)  = 1  => identification flag of 2 vertex of 1 cell
connect(4)  = 2  => identification flag of 3 vertex of 1 cell
connect(5)  = 3  => identification flag of 4 vertex of 1 cell
connect(6)  = 4  => identification flag of 5 vertex of 1 cell
connect(7)  = 5  => identification flag of 6 vertex of 1 cell
connect(8)  = 6  => identification flag of 7 vertex of 1 cell
connect(9)  = 7  => identification flag of 8 vertex of 1 cell
! second cell
connect(10) = 5  => number of vertices of 2 cell
connect(11) = 0  => identification flag of 1 vertex of 2 cell
connect(12) = 1  => identification flag of 2 vertex of 2 cell
connect(13) = 2  => identification flag of 3 vertex of 2 cell
connect(14) = 3  => identification flag of 4 vertex of 2 cell
connect(15) = 8  => identification flag of 5 vertex of 2 cell
```

Note that the first 4 identification flags of pyramid vertices as the same of the first 4 identification flags of the hexahedron because the two cells share this face. It is also important to note that the identification flags start form 0 value: this is impose to the VTK standard. The function VTK_CON does not calculate the connectivity vector: it writes the connectivity vector conforming the VTK standard, but does not calculate it. In the future release of LIB_VTK_IO will be included a function to calculate the connectivity vector.

The vector variable TIPO must conform the VTK standard [1]. It contains the *type* of each cells. For the above example this vector is:

Cell-Type vector example for VTK legacy standard

```
tipo(1) = 12  => VTK hexahedron type of 1 cell
tipo(2) = 14  => VTK pyramid type of 2 cell
```

1 See the file VTK-Standard at the Kitware homepage.

The following is an example of VTK_CON calling:

VTK_CON Calling

```
...
integer(4), parameter:: NC=2
integer(4), parameter:: Nvertex1=8
integer(4), parameter:: Nvertex2=5
integer(4), parameter:: dc=NC+Nvertex1+Nvertex2
integer(4)::            connect(1:dc)
integer(4)::            cell_type(1:NC)
...
E_IO = VTK_CON(NC,connect,cell_type)
...
```

3.4 FUNCTION VTK_DAT

VTK_DAT SIGNATURE

```
function VTK_DAT(NC_NN, var_location) result(E_IO)
```

This function MUST be called before saving the data related to geometric mesh. This function initializes the saving of data variables indicating the *type* of variables that will be saved.

VTK_DAT VARIABLES

```
integer(I4P),  intent(IN)::  NC_NN          ! number of cells or nodes of field
character(*),  intent(IN)::  var_location   ! location of saving variables: cell for cell-centered, node for node-centered
integer(I4P)::               E_IO           ! Input/Output inquiring flag: 0 if IO is done, > 0 if IO is not done
character(len=maxlen)::      s_buffer       ! buffer string
```

The VTK_DAT variables have the following meaning:

NC_NN indicates the number of all cells or all nodes according to the value of tipo.

var_location contains the location-type of variables that will be saved after VTK_DAT. It is a scalar and cab assume the following values:

A. *cell* (it is case insensitive) → variables will be cell-centered.

B. *node* (it is case insensitive) → variables will be node-centered.

E_IO contains the inquiring integer flag for error handling.

Of course a single file can contain both cell and node centered variables; in this case the VTK_DAT function must be called two times, before saving cell-centered variables and before saving node-centered variables.

The following is an example of VTK_DAT calling:

VTK_DAT Calling

```
...
E_IO = VTK_DAT(50,'node')
...
```

3.5 VTK_VAR

VTK_VAR is an interface to 8 different functions; there are 3 functions for scalar variables, 3 functions for vectorial variables and 2 function texture variables. This function saves the data variables related to geometric mesh. The inputs that must be passed change depending on the data variables type.

3.5.1 VTK_VAR SCALAR DATA

VTK_VAR SCALAR DATA SIGNATURE

```
function VTK_VAR(formato,NC_NN,varname,var) result(E_IO)
```

This kind of call is used to save scalar data.

VTK_VAR SCALAR DATA VARIABLES

```
integer(I4P),                        intent(IN):: NC_NN        ! number of nodes or cells
character(*),                        intent(IN):: varname      ! variable name
real(R8P) or real(R4P) or integer(I4P), intent(IN):: var(1:NC_NN) ! variable to be saved
integer(I4P)::                                     E_IO         ! Input/Output inquiring flag: 0 if IO is done,
> 0 if IO is not done
```

The VTK_VAR variables have the following meaning:

NC_NN indicates the number of all cells or all nodes according to the value of tipo passed to VTK_DAT.

varname contains the name attribuited the variable saved.

var contains the values of variables in each nodes or cells. It is a vector of $[1 : NC\_NN]$.

E_IO contains the inquiring integer flag for error handling.

Note that the variables `var` can be passed both as 8-byte real kind, 4-byte real kind and 4-byte integer; the dynamic displacement interface will call the correct function.

The following is an example of VTK_VAR scalar data calling:

VTK_VAR Scalar Data Calling

```
...
integer(4), parameter:: NC_NN=100
real(4)::                var(1:NC_NN)
...
E_IO = VTK_VAR(NC_NN,'Scalar Data',var)
...
```

3.5.2 VTK_VAR REAL VECTORIAL DATA

VTK_VAR REAL VECTORIAL DATA SIGNATURE

```
function VTK_VAR(tipo,NC_NN,varname,varX,varY,varZ) result(E_IO)
```

This kind of call is used to save real vectorial data.

VTK_VAR REAL VECTORIAL DATA VARIABLES

```
character(*),     intent(IN)::  vec_type        ! vector type: vect = generic vector , norm = normal vector
integer(I4P),     intent(IN)::  NC_NN           ! number of nodes or cells
character(*),     intent(IN)::  varname         ! variable name
real(R8P or R4P), intent(IN)::  varX(1:NC_NN)   ! x component of vector
real(R8P or R4P), intent(IN)::  varY(1:NC_NN)   ! y component of vector
real(R8P or R4P), intent(IN)::  varZ(1:NC_NN)   ! z component of vector
integer(I4P)::                  E_IO            ! Input/Output inquiring flag: 0 if IO is done, > 0 if IO is not done
```

The VTK_VAR variables have the following meaning:

tipo indicates the type of vector. It can assume the following value:

A. *vect* → generic vector.
B. *norm* → normal vector of face.

NC_NN indicates the number of all cells or all nodes according to the value of tipo passed to VTK_DAT.

varname contains the name attribuited the variable saved.

varX contains the values of X component in each nodes or cells. It is a vector of [1 : NC_NN].

varY contains the values of Y component in each nodes or cells. It is a vector of [1 : NC_NN].

varZ contains the values of Z component in each nodes or cells. It is a vector of [1 : NC_NN].

E_IO contains the inquiring integer flag for error handling.

Note that the variables varX,varY,varZ can be passed both as 8-byte real kind and 4-byte real kind; the dynamic displacement interface will call the correct function.

The following is an example of VTK_VAR real vectorial data calling:

VTK_VAR Real Vectorial Data Calling

```
...
integer(4), parameter:: NC_NN=100
real(4)::              varX(1:NC_NN)
real(4)::              varZ(1:NC_NN)
real(4)::              varZ(1:NC_NN)
...
E_IO = VTK_VAR('vect',NC_NN,'Real Vectorial Data',...
            ...varX,varY,varZ)
...
```

3.5.3 VTK_VAR INTEGER VECTORIAL DATA

VTK_VAR INTEGER VECTORIAL DATA SIGNATURE

```
function VTK_VAR(NC_NN,varname,varX,varY,varZ) result(E_IO)
```

This kind of call is used to save integer vectorial data.

VTK_VAR INTEGER VECTORIAL DATA VARIABLES

```
integer(R4P),   intent(IN):: NC_NN         ! number of nodes or cells
character(*),   intent(IN):: varname       ! variable name
integer(R4P),   intent(IN):: varX(1:NC_NN) ! x component of vector
integer(R4P),   intent(IN):: varY(1:NC_NN) ! y component of vector
integer(R4P),   intent(IN):: varZ(1:NC_NN) ! z component of vector
integer(R4P)::               E_IO          ! Input/Output inquiring flag: 0 if IO is done, > 0 if IO is not done
```

The VTK_VAR variables have the following meaning:

NC_NN indicates the number of all cells or all nodes according to the value of tipo passed to VTK_DAT.

varname contains the name attribuited the variable saved.

varX contains the values of X component in each nodes or cells. It is a vector of [1 : NC_NN].

varY contains the values of Y component in each nodes or cells. It is a vector of $[1 : NC\_NN]$.

varZ contains the values of Z component in each nodes or cells. It is a vector of $[1 : NC\_NN]$.

E_IO contains the inquiring integer flag for error handling.

The following is an example of VTK_VAR real vectorial data calling:

VTK_VAR Integer Vectorial Data Calling

```
...
integer(4), parameter:: NC_NN=100
integer(4)::            varX(1:NC_NN)
integer(4)::            varZ(1:NC_NN)
integer(4)::            varZ(1:NC_NN)
...
E_IO = VTK_VAR(NC_NN,'Integer Vectorial Data', &
               varX,varY,varZ)
...
```

3.5.4 VTK_VAR TEXTURE DATA

VTK_VAR TEXTURE DATA SIGNATURE

```
function VTK_VAR(NC_NN, ,dimm,varname,textCoo) result(E_IO)
```

This kind of call is used to save texture data.

VTK_VAR TEXTURE DATA VARIABLES

```
integer(R4P),          intent(IN):: NC_NN                   ! number of nodes or cells
integer(R4P),          intent(IN):: dimm                    ! texture dimensions
character(*),          intent(IN):: varname                 ! variable name
real(R8P or R4P),      intent(IN):: textCoo(1:NC_NN,1:dimm) ! texture
integer(R4P)::                      E_IO                    ! Input/Output inquiring flag: 0 if IO is done, > 0 if IO
is not done
```

The VTK_VAR variables have the following meaning:

NC_NN indicates the number of all cells or all nodes according to the value of tipo passed to VTK_DAT.

dimm indicates the dimensions of the texture coordinates. It can assume the value:

A. $1 \rightarrow$ scalar texture.

B. $2 \rightarrow$ twodimensional texture.

C. 3 → threedimensional texture.

varname contains the name attribuited the variable saved.

textCoo contains the coordinates of texture in each nodes or cells. It is a vector of $[1 : NC\_NN, 1 : dimm]$.

E_IO contains the inquiring integer flag for error handling.

Note that the variable `textCoo` can be passed both as 8-byte real kind and 4-byte real kind; the dynamic displacement interface will call the correct function.

The following is an example of VTK_VAR texture data calling:

VTK_VAR Texture Data Calling

```
...
integer(4), parameter:: NC_NN=100
integer(4), parameter:: dimm=2
real(4)::               textCoo(1:NC_NN,1:dimm)
...
E_IO = VTK_VAR(NC_NN,dimm,'Texture Data',textCoo)
...
```

3.6 FUNCTION VTK_END

VTK_END SIGNATURE

```
function VTK_END() result(E_IO)
```

This function is used to finalize the file opened and it has not inputs. The LIB_VTK_IO manages the file unit without the user's action.

VTK_END VARIABLES

```
integer(I4P):: E_IO ! Input/Output inquiring flag: 0 if IO is done, > 0 if IO is not done
```

The VTK_END variables have the following meaning:

E_IO contains the inquiring integer flag for error handling.

The following is an example of VTK_END calling:

VTK_END Calling

```
...
E_IO = VTK_END()
...
```

4

VTK XML FUNCTIONS

Contents

THE XML standard is more powerful than legacy one. It is more flexible and free but on the other hand is more (but not so more using a library like LIB_VTK_IO ...) complex than legacy standard. The output of XML functions is a well-formated XML file at least for the ascii format (in the binary format LIB_VTK_IO use raw-data format that does not produce a well formated XML file).

The XML functions follow the same calling-convention of the legacy functions; all the LIB_VTK_IO XML functions are 4-BYTE INTEGER FUNCTION: the output of these functions is an integer that is 0 if the function calling has been done right while it is > 0 if some errors occur. The functions calling is the same as legacy functions:

Functions Calling

```
...
```

```
integer(4):: E_IO
...
E_IO = VTK_INI_XML(....
...
```

Note that the XML functions have the same name of legacy functions with the suffix "_XML".

4.1 FUNCTION VTK_INI_XML

VTK_INI_XML SIGNATURE

```
function VTK_INI_XML(output_format,filename,mesh_topology,nx1,nx2,ny1,ny2,nz1,nz2
```

The VTK_INI_XML function is used for initializing file. This function must be the first to be called.

VTK_INI_XML VARIABLES

```
character(*), intent(IN)::                  output_format  ! output format: ASCII or BINARY
character(*), intent(IN)::                  filename       ! file name
character(*), intent(IN)::                  mesh_topology  ! mesh topology
integer(I4P), intent(IN), optional::        nx1,nx2        ! initial and final nodes of x axis
integer(I4P), intent(IN), optional::        ny1,ny2        ! initial and final nodes of y axis
integer(I4P), intent(IN), optional::        nz1,nz2        ! initial and final nodes of z axis
integer(I4P)::                              E_IO           ! Input/Output inquiring flag: 0 if IO is done, > 0 if IO is
not done
character(len=maxlen)::                     s_buffer       ! buffer string
```

The VTK_INI_XML variables have the following meaning:

output_format indicates the "format" of output file. It can assume the following values:

A. *ascii* (it is case insensitive) → creating an ascii output file.

B. *binary* (it is case insensitive) → creating a binary (big_endian encoding) output file.

filename contains the name (with its path) of the output file.

topology indicates the topology of the mesh and can assume the following values:

A. *StructuredGrid*.

B. *RectilinearGrid*.

C. *UnstructuredGrid*.

nx1,nx2 contains the extent of X axis; $nx1$ is the initial node and $nx2$ is the final.

ny1,ny2 contains the extent of Y axis; $ny1$ is the initial node and $ny2$ is the final.

nz1,nz2 contains the extent of Z axis; $nz1$ is the initial node and $nz2$ is the final.

E_IO contains the inquiring integer flag for error handling.

This function is quite more complex than the rispective legacy function; it needs more inputs: the XML standard needs more informations to initialize the file.
The following is an example of VTK_INI_XML calling:

VTK_INI_XML Calling

```
...
...
E_IO = VTK_INI_XML('BINARY','XML_RECT_BINARY.vtr', &
                   'RectilinearGrid',              &
                   nx1=nx1,nx2=nx2,                &
                   ny1=ny1,ny2=ny2,                &
                   nz1=nz1,nz2=nz2)
...
```

Note that the file extension is necessary in the file name. The XML standard has different extensions for each different topologies (i.e. .VTR for rectilinear topology). See the VTK-standard file for more information.

4.2 VTK_GEO_XML

VTK_GEO_XML is an interface to 6 different functions; there are 2 functions for each 3 topologies supported. This function must be called after VTK_INI_XML. It saves the mesh geometry. The inputs that must be passed change depending on the topologies choiced. Not all VTK topologies have been implemented ("polydata" topologies are absent). The signatures for all implemented topologies are now reported.

4.2.1 VTK_GEO_XML STRUCTURED GRID

VTK_GEO_XML STRUCTURED GRID SIGNATURE

```
function VTK_GEO_XML(nx1,nx2,ny1,ny2,nz1,nz2,NN, &
                     X,Y,Z) result(E_IO)
```

The topology "structured grid" is useful for structured grid with non-uniform discretization steps.

VTK_GEO_XML STRUCTURED GRID VARIABLES

```
integer(I4P),     intent(IN):: nx1,nx2   ! initial and final nodes of x axis
integer(I4P),     intent(IN):: ny1,ny2   ! initial and final nodes of y axis
integer(I4P),     intent(IN):: nz1,nz2   ! initial and final nodes of z axis
```

```
integer(I4P),      intent(IN):: NN        ! number of all nodes
real(R8P or R4P), intent(IN):: X(1:NN)   ! x coordinates
real(R8P or R4P), intent(IN):: Y(1:NN)   ! y coordinates
real(R8P or R4P), intent(IN):: Z(1:NN)   ! z coordinates
integer(I4P)::                  E_IO      ! Input/Output inquiring flag: 0 if IO is done, > 0 if IO is not done
```

Note that the variables X,Y,Z can be passed both as 8-byte real kind and 4-byte real kind; the dynamic displacement interface will call the correct function. Mixing 8-byte real kind and 4-byte real kind is not allowed: be sure that all variables are 8-byte real kind or all are 4-byte real kind.

The VTK_GEO_XML structured grid variables have the following meaning:

nx1,nx2 contains the extent of X axis; $nx1$ is the initial node and $nx2$ is the final.

ny1,ny2 contains the extent of Y axis; $ny1$ is the initial node and $ny2$ is the final.

nz1,nz2 contains the extent of Z axis; $nz1$ is the initial node and $nz2$ is the final.

NN contains the global number of nodes $NN = (nx2 - nx1 + 1) * (ny2 - ny1 + 1) * (nz2 - nz1 + 1)$.

X contains the X coordinates values of all nodes. It is a vector of $[1 : NN]$.

Y contains the Y coordinates values of all nodes. It is a vector of $[1 : NN]$.

Z contains the Z coordinates values of all nodes. It is a vector of $[1 : NN]$.

E_IO contains the inquiring integer flag for error handling.

The following is an example of VTK_GEO_XML structured grid calling:

VTK_GEO_XML Structured Grid Calling

```
...
integer(4):: nx1,nx2
integer(4):: ny1,ny2
integer(4):: nz1,nz2
integer(4):: NN
real(4):: X(1:NN),Y(1:NN),Z(1:NN)
...
E_IO = VTK_GEO_XML(nx1,nx2,ny1,ny2,nz1,nz2, &
                   NN,                       &
                   X,Y,Z)
...
```

4.2.2 VTK_GEO_XML RECTILINEAR GRID

VTK_GEO_XML RECTILINEAR GRID SIGNATURE

```
function VTK_GEO_XML(nx1,nx2,ny1,ny2,nz1,nz2, &
                     X,Y,Z) result(E_IO)
```

The topology "rectilinear grid" is useful for structured grid with non-uniform discretization steps even in generalized coordinates.

VTK_GEO_XML RECTILINEAR GRID VARIABLES

```
integer(I4P),       intent(IN):: nx1,nx2       ! initial and final nodes of x axis
integer(I4P),       intent(IN):: ny1,ny2       ! initial and final nodes of y axis
integer(I4P),       intent(IN):: nz1,nz2       ! initial and final nodes of z axis
real(R8P or R4P),   intent(IN):: X(nx1:nx2)    ! x coordinates
real(R8P or R4P),   intent(IN):: Y(ny1:ny2)    ! y coordinates
real(R8P or R4P),   intent(IN):: Z(nz1:nz2)    ! z coordinates
integer(I4P)::                   E_IO          ! Input/Output inquiring flag: 0 if IO is done, > 0 if IO is not done
```

Note that the variables `X,Y,Z` can be passed both as 8-byte real kind and 4-byte real kind; the dynamic displacement interface will call the correct function. Mixing 8-byte real kind and 4-byte real kind is not allowed: be sure that all variables are 8-byte real kind or all are 4-byte real kind.

The VTK_GEO_XML rectilinear grid variables have the following meaning:

nx1,nx2 contains the extent of X axis; $nx1$ is the initial node and $nx2$ is the final.

ny1,ny2 contains the extent of Y axis; $ny1$ is the initial node and $ny2$ is the final.

nz1,nz2 contains the extent of Z axis; $nz1$ is the initial node and $nz2$ is the final.

X contains the X coordinates values of X nodes. It is a vector of $[nx1 : nx2]$.

Y contains the Y coordinates values of Y nodes. It is a vector of $[ny1 : ny2]$.

Z contains the Z coordinates values of Z nodes. It is a vector of $[nz1 : nz2]$.

E_IO contains the inquiring integer flag for error handling.

The following is an example of VTK_GEO_XML rectilinear grid calling:

VTK_GEO_XML Structured Grid Calling

```
...
integer(4):: nx1,nx2
integer(4):: ny1,ny2
integer(4):: nz1,nz2
real(4):: X(nx1:nx2),Y(ny1:ny2),Z(nz1:nz2)
...
E_IO = VTK_GEO_XML(nx1,nx2,ny1,ny2,nz1,nz2, &
                   X,Y,Z)
...
```

4.2.3 VTK_GEO_XML UNSTRUCTURED GRID

VTK_GEO_XML UNSTRUCTURED GRID SIGNATURE

```
function VTK_GEO_XML(Nnodi,NCelle,X,Y,Z) result(E_IO)
```

The topology “unstructured grid” is necessary for unstructured grid, the most general mesh format. This topology is also useful for scructured mesh in order to save only a non-structured clip of mesh.

VTK_GEO_XML UNSTRUCTURED GRID VARIABLES

```
integer(I4P),        intent(IN):: NN        ! number of nodes
integer(I4P),        intent(IN):: NC        ! number of cells
real(R8P or R4P),    intent(IN):: X(1:NN)   ! x coordinates
real(R8P or R4P),    intent(IN):: Y(1:NN)   ! y coordinates
real(R8P or R4P),    intent(IN):: Z(1:NN)   ! z coordinates
integer(I4P)::                    E_IO      ! Input/Output inquiring flag: 0 if IO is done, > 0 if IO is not done
```

Note that the variables `X,Y,Z` can be passed both as 8-byte real kind and 4-byte real kind; the dynamic displacement interface will call the correct function. Mixing 8-byte real kind and 4-byte real kind is not allowed: be sure that all variables are 8-byte real kind or all are 4-byte real kind.

The VTK_GEO_XML unstructured grid variables have the following meaning:

Nnodi indicates the number of all nodes.

NCelle indicates the number of all cells.

X contains the X coordinates values of nodes. It is a vector of $[1 : Nnodi]$.

Y contains the Y coordinates values of nodes. It is a vector of $[1 : Nnodi]$.

Z contains the Z coordinates values of nodes. It is a vector of $[1 : Nnodi]$.

E_IO contains the inquiring integer flag for error handling.

The following is an example of VTK_GEO_XML unstructured grid calling:

VTK_GEO_XML Unstructured Grid Calling

```
...
integer(4), parameter:: Nnodi=100
integer(4), parameter:: NCelle=50
real(4):: X(1:Nnodi),Y(1:Nnodi),Z(1:Nnodi)
...
E_IO = VTK_GEO_XML('ascii',Nnodi,NCelle,X,Y,Z)
...
```

In order to use the “unstructured grid” it is necessary to save also the “connectivity” of the grid. The connectivity must be saved with the function VTK_CON_XML.

4.2.4 VTK_GEO_XML CLOSE PIECE

VTK_GEO_XML CLOSE PIECE SIGNATURE

```
function VTK_GEO_XML() result(E_IO)
```

As we said before the XML standard is more powerful than legacy. XML file can contain more than 1 mesh with its associated variables. Thus there is the necessity to close each "pieces" that compose the data-set saved in the XML file. The VTK_GEO_XML called in the "close piece" format is used just to close the current piece before saving another piece or closing the file.

VTK_GEO_XML CLOSE PIECE VARIABLES

```
integer(I4P):: E_IO ! Input/Output inquiring flag: 0 if IO is done, > 0 if IO is not done
```

The VTK_GEO_XML close piece variables have the following meaning:

E_IO contains the inquiring integer flag for error handling.

The following is an example of VTK_GEO_XML close piece calling:

VTK_GEO_XML Unstructured Grid Calling

```
...
E_IO = VTK_GEO_XML()
...
```

4.3 FUNCTION VTK_CON_XML

VTK_CON_XML SIGNATURE

```
function VTK_CON_XML(NC, connect, offset, cell_type) result(E_IO)
```

This function MUST be used when unstructured grid is used. It saves the connectivity of the unstructured mesh.

VTK_CON_XML VARIABLES

```
integer(I4P), intent(IN):: NC             ! number of cells
integer(I4P), intent(IN):: connect(:)     ! mesh connectivity
integer(I4P), intent(IN):: offset(1:NC)   ! cell offset
```

```
integer(I1P), intent(IN)::  cell_type(1:NC)  ! VTK cell type
integer(I4P)::              E_IO             ! Input/Output inquiring flag: 0 if IO is done, > 0 if IO is not done
character(len=maxlen)::     s_buffer         ! buffer string
integer(I4P)::              n1               ! counter
```

The VTK_CON_XML variables have the following meaning:

NCelle indicates the number of all cells.

connect contains the connectivity of the mesh. It is a vector.

offset contains the offset[1] of every cells. It is a vector of $[1 : NCelle]$.

tipo contains the type of every cells. It is a vector of $[1 : NCelle]$.

E_IO contains the inquiring integer flag for error handling.

The vector CONNECT must follow the VTK XML standard. It is passed as ASSUMED-SHAPE array because its dimensions is related to the mesh dimensions in a complex way. Its dimensions can be calculated by the following equation:

$$dc = \sum_{i=1}^{NCelle} nvertex_i \tag{4.1}$$

where dc is connectivity vector dimension and $nvertex_i$ is the number of vertices of i^{th} cell. Note that this equation is different from the legacy one (eq. 3.1). The XML connectivity convention is quite different from the legacy standard. As an example considering the same mesh of section **??**: suppose we have a mesh composed by 2 cells, one hexahedron (8 vertices) and one pyramid with square basis (5 vertices); suppose that the basis of pyramid is constitute by a face of the hexahedron and so the two cells share 4 vertices. The equation 4.1 gives $dc = 8 + 5 = 13$; the connectivity vector for this mesh can be:

Connectivity vector example for VTK XML standard

```
! first cell
connect(1)  = 0  => identification flag of 1 vertex of 1 cell
connect(2)  = 1  => identification flag of 2 vertex of 1 cell
connect(3)  = 2  => identification flag of 3 vertex of 1 cell
connect(4)  = 3  => identification flag of 4 vertex of 1 cell
connect(5)  = 4  => identification flag of 5 vertex of 1 cell
connect(6)  = 5  => identification flag of 6 vertex of 1 cell
connect(7)  = 6  => identification flag of 7 vertex of 1 cell
connect(8)  = 7  => identification flag of 8 vertex of 1 cell
! second cell
connect(9)  = 0  => identification flag of 1 vertex of 2 cell
connect(10) = 1  => identification flag of 2 vertex of 2 cell
connect(11) = 2  => identification flag of 3 vertex of 2 cell
connect(12) = 3  => identification flag of 4 vertex of 2 cell
connect(13) = 8  => identification flag of 5 vertex of 2 cell
```

1 The summ of nodes of all previous cells included the current cell.

Therefore this connectivity vector convention is more simple than the legacy convention, now we must create also the OFFSET vector that contains the data now missing in the CONNECT vector. The offset vector for this mesh can be:

Offset vector example for VTK XML standard

```
! first cell
offset(1) = 8  => summ of nodes of 1 cell
! second cell
offset(2) = 13 => summ of nodes of 1 and 2 cells
```

The value of every cell-offset can be calculated by the following equation:

$$\mathtt{offset}_c = \sum_{i=1}^{c} nvertex_i \tag{4.2}$$

where $\mathtt{offset}_c$ is the value of c^{th} cell and $nvertex_i$ is the number of vertices of i^{th} cell.

The function VTK_CON_XML does not calculate the connectivity and offset vectors: it writes the connectivity and offset vectors conforming the VTK XML standard, but does not calculate them. In the future release of LIB_VTK_IO will be included a function to calculate the connectivity and offset vector.

The vector variable TIPO must conform the VTK XML standard [2] that is the same of the legacy standard presented previous (sec. **??**). It contains the *type* of each cells. For the above example this vector is:

Cell-Type vector example for VTK legacy standard

```
tipo(1) = 12  => VTK hexahedron type of 1 cell
tipo(2) = 14  => VTK pyramid type of 2 cell
```

The following is an example of VTK_CON_XML calling:

VTK_CON_XML Calling

```
...
integer(4), parameter:: NCelle=2
integer(4), parameter:: Nvertex1=8
integer(4), parameter:: Nvertex2=5
integer(4), parameter:: dc=Nvertex1+Nvertex2
integer(4)::            connect(1:dc)
integer(4)::            offset(1:NCelle)
integer(4)::            tipo(1:NCelle)
...
E_IO = VTK_CON_XML(NCelle,connect,offset,tipo)
...
```

2 See the file VTK-Standard at the Kitware homepage.

4.4 FUNCTION VTK_DAT_XML

VTK_DAT_XML SIGNATURE

```
function VTK_DAT_XML(var_location ,var_block_action) result(E_IO)
```

This function MUST be called before saving the data related to geometric mesh. This function initializes the saving of data variables indicating the *type* of variables that will be saved.

VTK_DAT_XML VARIABLES

```
character(*), intent(IN)::  var_location      ! location of saving variables: CELL for cell-centered, NODE for node-centered
character(*), intent(IN)::  var_block_action  ! variables block action: OPEN or CLOSE block
integer(I4P)::              E_IO              ! Input/Output inquiring flag: 0 if IO is done, > 0 if IO is not done
```

The VTK_DAT_XML variables have the following meaning:

var_location contains the location-type of variables that will be saved after VTK_DAT. It is a scalar and cab assume the following values:

A. *cell* (it is case insensitive) → variables will be cell-centered.

B. *node* (it is case insensitive) → variables will be node-centered.

var_block_action indicates if the block-data-variables is being opened or closed; it can assume the following values:

A. *open* (it is case insensitive) → block-data is being opened.

B. *close* (it is case insensitive) → block-data is being closed.

E_IO contains the inquiring integer flag for error handling.

Of course a single file can contain both cell and node centered variables. The VTK_DAT_XML must be called two times, before saving a block-data-variables in order to open the block, and after the block-data-variables has been saved in order to close the block. XML file can contains as many blocks as you want.

The following is an example of VTK_DAT_XML calling:

VTK_DAT_XML Calling

```
...
E_IO = VTK_DAT_XML('node','OPEN')
...
SAVE YOUR DATA WITH VTK_VAR_XML
...
E_IO = VTK_DAT_XML('node','CLOSE')
...
```

4.5 VTK_VAR_XML

VTK_VAR_XML is an interface to 12 different functions; there are 6 functions for scalar variables (1 for each supported precision: R8P, R4P, I8P, I4P, I2P and I1P) and 6 for vectorial variables (1 for each supported precision: R8P, R4P, I8P, I4P, I2P and I1P) This function saves the data variables related to geometric mesh. The inputs that must be passed change depending on the data variables type.

4.5.1 VTK_VAR_XML SCALAR DATA

VTK_VAR_XML SCALAR DATA SIGNATURE

```
function VTK_VAR_XML(NC_NN,varname,var) result(E_IO)
```

This kind of call is used to save scalar data.

VTK_VAR_XML SCALAR DATA VARIABLES

```
integer(I4P),   intent(IN):: NC_NN          ! number of cells or nodes
character(*),   intent(IN):: varname        ! variable name
real(R8P or...
     R4P) or...
integer(I8P or...
        I4P or...
        I2P or...
        I1P),   intent(IN):: var(1:NC_NN)   ! variable to be saved
integer(I4P)::               E_IO           ! Input/Output inquiring flag: 0 if IO is done, > 0 if IO is not done
```

The VTK_VAR_XML variables have the following meaning:

NC_NN indicates the number of all cells or all nodes according to the value of var_location passed to VTK_DAT_XML.

varname contains the name attribuited the variable saved.

var contains the values of variables in each nodes or cells. It is a vector of $[1 : NC\_NN]$.

E_IO contains the inquiring integer flag for error handling.

Note that the variables `var` can be passed both 8-byte real kind, 4-byte real kind, 8-byte integer, 4-byte integer, 2-byte integer and 1-byte integer; XML is very flexible; the dynamic displacement interface will call the correct function.

The following is an example of VTK_VAR_XML scalar data calling:

VTK_VAR_XML Scalar Data Calling

```
...
integer(4), parameter:: NC_NN=100
integer(2)::            var(1:NC_NN)
...
E_IO = VTK_VAR_XML(NC_NN,'Scalar Data',var)
...
```

4.5.2 VTK_VAR_XML VECTORIAL DATA

VTK_VAR_XML VECTORIAL DATA SIGNATURE

```
function VTK_VAR_XML(NC_NN,varname, &
                     varX,varY,varZ) result(E_IO)
```

This kind of call is used to save vectorial data.

VTK_VAR_XML VECTORIAL DATA VARIABLES

```
integer(I4P),                              intent(IN):: NC_NN          ! number of cells or nodes
character(*),                              intent(IN):: varname        ! variable name
real(R8P or R4P) or...
integer(I8P or I4P or I2P or I1P), intent(IN):: varX(1:NC_NN) ! x component
real(R8P or R4P) or...
integer(I8P or I4P or I2P or I1P), intent(IN):: varY(1:NC_NN) ! y component
real(R8P or R4P) or...
integer(I8P or I4P or I2P or I1P), intent(IN):: varZ(1:NC_NN) ! z component
integer(I4P)::                                 E_IO           ! Input/Output inquiring flag: 0 if IO is done,
> 0 if IO is not done
```

The VTK_VAR_XML variables have the following meaning:

NC_NN indicates the number of all cells or all nodes according to the value of var_location passed to VTK_DAT_XML.

varname contains the name attribuited the variable saved.

varX contains the values of X component in each nodes or cells. It is a vector of [1 : NC_NN].

varY contains the values of Y component in each nodes or cells. It is a vector of [1 : NC_NN].

varZ contains the values of Z component in each nodes or cells. It is a vector of [1 : NC_NN].

E_IO contains the inquiring integer flag for error handling.

Note that the variables `varX,varY,varZ` can be passed both 8-byte real kind, 4-byte real kind, 8-byte integer, 4-byte integer, 2-byte integer and 1-byte integer; XML is very flexible; the dynamic displacement interface will call the correct function.

The following is an example of VTK_VAR_XML vectorial data calling:

VTK_VAR_XML Vectorial Data Calling

```
...
integer(4), parameter:: NC_NN=100
integer(4)::           varX(1:NC_NN)
integer(4)::           varZ(1:NC_NN)
integer(4)::           varZ(1:NC_NN)
...
E_IO = VTK_VAR_XML(NC_NN,'Vectorial Data', &
                   varX,varY,varZ)
...
```

4.6 FUNCTION VTK_END_XML

VTK_END_XML SIGNATURE

```
function VTK_END_XML() result(E_IO)
```

This function is used to finalize the file opened. The LIB_VTK_IO manages the file unit without the user's action.

VTK_END_XML VARIABLES

```
integer(I4P)::                   E_IO       ! Input/Output inquiring flag: 0 if IO is done, > 0 if IO is not done
character(2)::                   var_type   ! var_type = R8,R4,I8,I4,I2,I1
real(R8P),     allocatable::     v_R8(:)    ! R8 vector for IO in AppendData
real(R4P),     allocatable::     v_R4(:)    ! R4 vector for IO in AppendData
integer(I8P),  allocatable::     v_I8(:)    ! I8 vector for IO in AppendData
integer(I4P),  allocatable::     v_I4(:)    ! I4 vector for IO in AppendData
integer(I2P),  allocatable::     v_I2(:)    ! I2 vector for IO in AppendData
integer(I1P),  allocatable::     v_I1(:)    ! I1 vector for IO in AppendData
integer(I4P)::                   N_v        ! vector dimension
integer(I4P)::                   n1         ! counter
```

The following is an example of VTK_END_XML calling:

VTK_END_XML Calling

```
...
E_IO = VTK_END_XML()
...
```