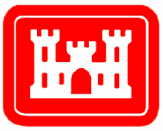
ERDC SR-07-1



US Army Corps
of Engineers®
Engineer Research and
Development Center

System-Wide Water Resources Program

eXtensible Model Data Format (XMDF)

Cary D. Butler, David R. Richards, Robert M. Wallace,
Norman L. Jones, and Russell Jones

January 2007

Engineer Research
and Development Center

ERDC SR-07-1
January 2007

eXtensible Model Data Format (XMDF)

Cary D. Butler, David R. Richards

Information Technology Laboratory
U.S. Army Engineer Research and Development Center
3909 Halls Ferry Road
Vicksburg, MS 39180-6199

Robert M. Wallace

Coastal and Hydraulics Laboratory
U.S. Army Engineer Research and Development Center
3909 Halls Ferry Road
Vicksburg, MS 39180-6199

Norman L. Jones, Russell Jones

Environmental Modeling Research Laboratory
242 Clyde Building
Brigham Young University
Provo, Utah 84602

Final report

Prepared for U.S. Army Corps of Engineers
Washington, DC 20314-1000

Under Projects 122401 and 122425

Abstract: The U.S. Army Engineer Research and Development Center, in conjunction with the Environmental Modeling Research Laboratory (EMRL) at Brigham Young University (BYU), is developing an efficient Application Programming Interface (API) for handling multidimensional data produced for water resource computational modeling. This API, in conjunction with a corresponding data standard, is being implemented within ERDC computational models to facilitate rapid data access, enhanced data compression and data sharing, and cross-platform independence. The API and data standard are known as the eXtensible Model Data Format (XMDF), and version 1.0 is available for public use and free dissemination. This report presents the purpose and architecture of the XMDF API and data format.

Contents

List of Figures

List of Tables

Preface

The work described herein was sponsored by Headquarters, U.S. Army Corps of Engineers (HQUSACE), as part of the System-Wide Water Resources Program. The Program Manager was Dr. Steven L. Ashby, Environmental Laboratory, Vicksburg, MS, U.S. Army Engineer Research and Development Center (ERDC). The research was performed under Project 122401, entitled "Data Management," for which Mr. James T. Stinson, Engineering and Informatic Systems Division, Information Technology Laboratory (ITL), Vicksburg, MS, ERDC, was the Principal Investigator, and Project 122425, "Model Integration," for which Dr. Robert M. Wallace, Flood and Storm Protection Division, Coastal and Hydraulics Laboratory (CHL), ERDC, was Principal Investigator. The HQUSACE Technical Monitor was Mr. M. K. Miles, CECW-EE.

This report was prepared by Dr. Cary D. Butler and Mr. David R. Richards, both Technical Directors, ITL; Dr. Wallace; Dr. Norman L. Jones, Professor of Civil and Environmental Engineering and Director of the Environmental Modeling Research Laboratory, Brigham Young University, Provo, UT; and Mr. Russell Jones, Research Assistant, also of the Environmental Modeling Research Laboratory. The work was conducted under the general supervision of Dr. Jeffery P. Holland, Director, ITL; and Mr. Thomas W. Richardson, Director, CHL.

COL James R. Rowan, EN, was Commander and Executive Director of ERDC. Dr. James R. Houston was Director.

1 Introduction

Multidimensional numerical modeling studies have traditionally necessitated a substantial number of large data files (both input and output). The shear number of data files can lead to confusion, misplaced or lost data, and poor communication. Also, each numerical model uses and generates data files in its own format, making the sharing of data and coupling of models difficult. Add to this the fact that files typically need to be ASCII (inefficient) in order to be passed across platforms, and the result is that too much computer time, computer space, and human interaction are needed in the modeling process.

To address these problems, the U.S. Army Engineer Research and Development Center (ERDC) has taken the initiative to develop a standard file format to be used in modeling studies. In the first phase of this project, the Environmental Modeling Research Laboratory (EMRL) at Brigham Young University, Provo, UT, has developed a standard format for geometry data storage. In this context, geometric data include

- River cross sections.
- Two-dimensional (2-D)/three-dimensional (3-D) structured grids.
- 2-D/3-D unstructured meshes (including sets of scattered data points triangulated into a triangulated irregular network (TIN)).
- Geometric paths through space with associated time data.

The data file will be able to store multiple instances of each of these data objects along with data sets associated with the object. Data sets represent scalar and vector arrays such as model solution data. In addition to developing the standard format, EMRL has provided an application programming interface (API) that developers of numerical codes can use to quickly and easily access this format without minimal difficulty. This standard is available to all of ERDC, and everyone is encouraged to adopt it.

Chapter 2 gives background, motivation, and an overview of the approach being used. Chapter 3 describes how to incorporate the eXtensible Model Data Format (XMDF) into a specific model and introduces the most commonly used functions. Chapter 4 includes technical implementation details for programmers.

Appendix A lists the constants used to store coordinate system information inside an XMDF file, and Appendix B summarizes the types and functions defined for the XMDF API.

2 Overview

The long-term goal of this project is to create a fast, efficient, and simple methodology for storing, accessing, and sharing data used in a numerical simulation. In developing a strategy, the format (both general and specific) must first be defined. The general format question is that of binary versus ASCII. The specific format issues include detailed data access and storage methods.

Binary files have size and speed performance advantages over ASCII files. However, because binary files are difficult to view and use, and vary from one platform to another, ASCII files are often utilized for clarity and portability, thus becoming cumbersome when large simulations are considered.

2.1 Overcoming Binary Portability Issues

A number of libraries have been developed to read and write cross-platform binary files. Two popular libraries include NetCDF and HDF5. After a search was performed for other alternatives, it was concluded that these two libraries have the most stability and support. NetCDF and HDF5 were evaluated. Both libraries have similar input/output (IO) performance times. HDF5 has more flexibility for data storage, compression, and data mining. HDF5 supports data folders and data structures, making it more customizable. Therefore, it was decided to utilize the HDF5 library for generic model data API.

HDF5 was developed at the National Center for Supercomputing Applications (NCSA) at the University of Illinois at Urbana-Champaign (see *http://hdf.ncsa.uiuc.edu/HDF5/*). HDF5 has many users in both government and private organizations. HDF5 is a free library developed primarily with funding from the Department of Energy and the National Aeronautics and Space Administration. It is a powerful library that includes the following features:

a. Data are organized in a structure similar to a directory system. This makes it possible to easily store related data in a single file in an organized manner.

b. Data are stored in a binary format for fast file IO.

c. Large files (> 2 GB) can be managed easily.

d. Data are platform- and language-independent due to automatic conversions performed by the HDF5 library. Platforms include AIX,

Cray, FreeBSD, HP-UX, IRIX, Linux, OSF1, Solaris, ASCI TFLOPS, and Windows.

e. The data can be easily compressed using tools embedded in HDF5.

f. Data are stored in arraylike entities called data sets. A data set can be read whole or data-mined for specific elements, slices, etc.

g. Data sets can include metadata, called attributes by HDF5, to describe the data.

2.2 HDF5 Methodology

The HDF5 library stores data in individual data sets that can be single values, arrays, bitstreams, or tables. These HDF5 data sets will be referred to as variables to avoid confusion with the term data set as it applies to a numerical solution. Each variable has its own identifying name. The variables can be organized into "groups" to define a directorylike structure. Attributes can be created and associated with individual variables or groups. Attributes are generally used to explain the use of the array in question. HDF5 files are random access and have optional compression algorithms.

The generic data format will be referred to as XMDF. The geometry formats (or types) supported will include one-dimensional (1-D) cross sections, 1-D meshes, 2-D meshes, 2-D grids, 3-D meshes, and 3-D grids. Both cell-centered and mesh-centered grids and both Cartesian and curvilinear grids will be supported.

2.3 HDF5 Viewer

One of the disadvantages of using a binary format is that the data can be difficult to view and edit compared with those for ASCII files. However, a generic viewing and editing tool is available for HDF5 that can viewed to traverse the hierarchical tree associated with an HDF5 file and edit any of the information in the file. The viewer is written in JAVA so it can run on multiple platforms. The viewer can be downloaded at *http://hdf.ncsa.uiuc.edu/hdf-java-html/hdfview/*.

2.4 XMDF Organization

An XMDF file consists of one or more groups. Each group represents an unstructured mesh (or set of scattered data points), a structured grid (either Cartesian or curvilinear), a set of cross sections, or a group of geometric paths in space (such as particle paths). Each of these groups may include one or more subgroups with functional data sets for the group. Also associated with each group, the XMDF file stores a subgroup that contains a set of variables that define the properties of the group. Each group type has a set of required properties that must be specified in order to define the group. Additional properties can

be added to the group for specific applications. Properties can include a single value for the group (such as a type of structured grid), or a value for each node, cell, or element in the group (such as a node id or name).

2.4.1 Mesh groups

Meshes will include 1-, 2-, and 3-D unstructured finite element meshes. For models that have multiple mesh types for the same simulation, 1-, 2-, and 3-D components are allowed to be stored in separate groups or as part of the same group.

Meshes consist of two main types of data: nodes and elements. For the nodes the nodal coordinates need to be defined. For elements, the element type and the element topology (connectivity) are defined. An option to store the coordinate system associated with a mesh is also included. The mesh group layout is shown in Figure 1. It includes two subgroups, one for elements and the other for nodes.

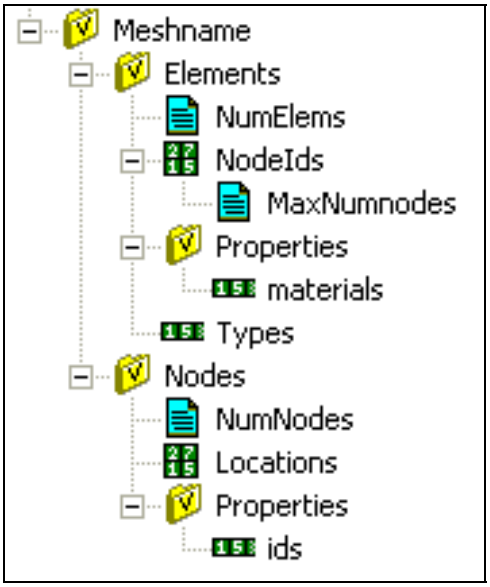


Figure 1. Mesh group layout

2.4.2 Grid groups

Grids differ from meshes in that they are structured. Each cell has an *ij* or *ijk* index. Both 2- and 3-D grids are supported. Data sets with solutions at cell centers, cell corners (mesh centered), as well as cell faces are supported (Figure 2).

2-D grids have rows and columns, and 3-D grids have rows, columns, and layers. These dimensions are referred to both as *u*, *v*, and *w* as well as *i*, *j*, and *k*. The first direction of a grid (row direction) is the *u* direction. The index in that direction is *i*. Changing the value of *u* or *i* is equivalent to moving back and forth between the different rows of a grid. The second direction of the grid is the *v* direction. This represents the grid columns, and the index is *j*. The third direction (for 3-D grids) is the *w* direction. This is the grid layers, and the index in this direction is *k*. Different models make different assumptions concerning how the *uvw* or *ijk* axes are configured. All possible configurations are represented with an orientation and a rotation. There are optional parameters to define a computation origin, and for 3-D grids there is another optional parameter that defines the orientation of the *uvw* axes at the computational origin. The orientation parameter defines whether the grid axes follow the right-hand rule or left-hand rule (for 2-D, this is viewed from above). The rotation includes two angles. The first is a rotation about the x-axis or the angle between the z-axis and third grid direction. This angle would be set to 90 to stand a 2-D grid on edge for vertically averaged simulations (such as CEQUAL-W2) and zero for depth-averaged simulations (such as STWAVE, M2D, and BOUSS2D). The angle also allows

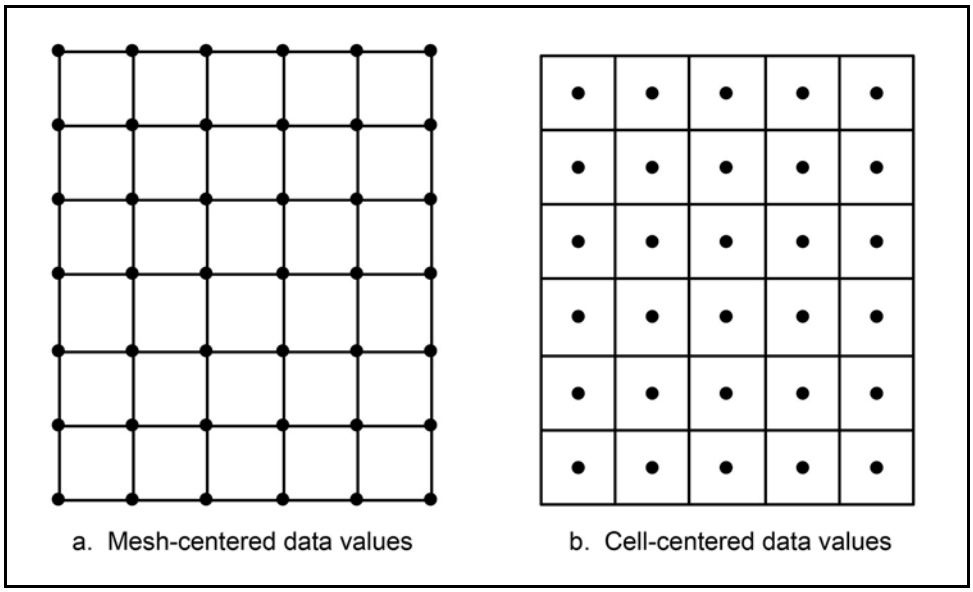


Figure 2. Data value

inclined planes for 3-D grids. The second angle defines the angle from the x-axis to the first grid direction. This can be thought of as a bearing or azimuth adjustment. It should be noted that these rotations are primarily for visualization since the models perform computations in local space. Numbering for data sets, attributes, etc., is done in *ijk* order. Both angles default to zero and are included as optional grid definition values.

2-D grids can be either Cartesian or curvilinear. 3-D grids can be Cartesian, curvilinear, or extruded 2-D grids. Options to specify 3-D layers on extruded 2-D grids include sigma stretch, Cartesian, specifying each corner z-value for every layer (curvilinear at corners), and specifying each column z-value for every layer (curvilinear at midsides/cell centers). Sigma stretch grids have varying z-values for the top and bottom of each column. Every layer in a sigma stretch grid has a constant percent thickness of the column thickness. All of the extrusion options are available regardless of the type of 2-D grid. For example, MODFLOW uses a 2-D Cartesian grid extruded using the curvilinear at cell centers option.

The data required to define a grid consist of global parameters and the grid geometry. The grid geometry consists of the row, column, layer dimensions (coordinates), or, in the case of curvilinear grids, the coordinates of the node corners.

2.4.3 Cross-section groups

Cross-section data include the typical cross sections that define channel bathymetry and profile (longitudinal) lines that can be used to represent centerline, bank line, or other “stream” paths within the channel defined by the cross

sections. Important line (material) and point (thalweg, bank) properties associated with cross sections, as well as other attributes, are stored.

A 1-D cross section consists of a set of distance (station) and elevation values (Figure 3). If the cross section is to be used in conjunction with terrain data, it must also be georeferenced. This means that the distance or station value can be converted into x- and y-values, and the elevation is assumed to be a z-value. The georeferencing can be provided on a point-by-point basis, in which case each point of the cross sections has an x- and y-coordinate defining its Cartesian location. Georeferencing also can be established using one or two points on the cross section. Single point georeferencing provides the x- and y-values associated with a distance along the section as well as an azimuth. Two-point georeferencing requires the specification of two (x, y) pairs as well as two distances along the section.

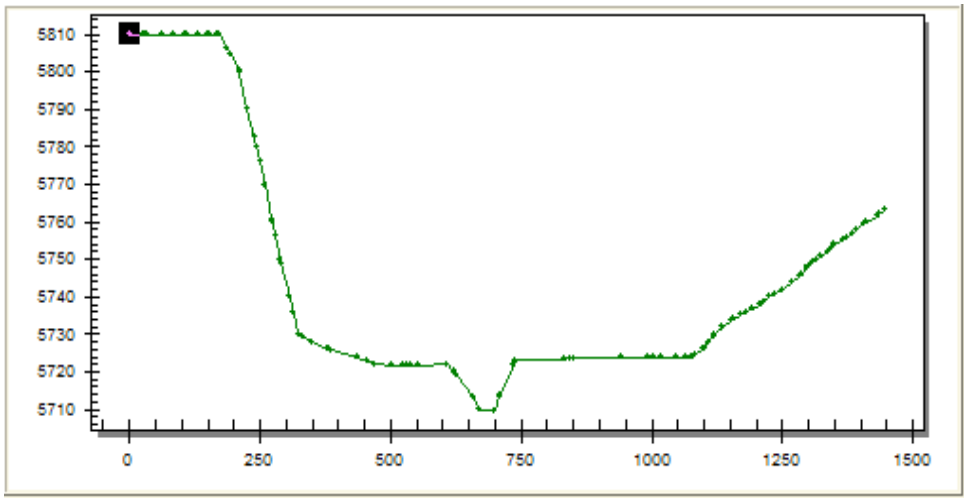


Figure 3. Sample cross section

A cross-section group in XMDF includes a set of cross sections, georeferencing parameters, and other properties associated with portions of or positions on the cross section.

2.4.4 Geometric path groups

A geometric path is a list of points (x, y, z) that are connected to form a series of line segments in space. The path represents the motion of an object through space. Each entry in the path is associated with a time. Additional data values may be associated with each entry in the path as well. For example, this could include the velocity of the object at this point in time, or the size of the object at this point in time.

Generally, there will be many paths grouped (or computed) together. For a group of geometric paths, each path would be defined from a shared set of time values. This allows the data to be stored as one 3-D array and a series of 2-D arrays. The 3-D array includes the coordinates (x, y, z) for a point on a path in

one direction, the point index in the second direction, and the time values in the third direction. The 2-D arrays include a value for each point on the path at each time.

Also, the geometric path group will support an optional spatial mapping object that includes spatial bins of the area covered by any path in the group (Figure 4). Each bin will contain a list of the paths that intersect that bin and the time ranges for which that path is inside the bin.

2.4.5 Data-set groups

A data set is a group of data that contains a functional value (scalar or vector) for each entity in a mesh or grid. Data sets on cross sections are handled separately.

The data sets are generally stored in a group below the group for the mesh or grid to which the data set belongs. The exception to this is when a file contains data sets for only an outside mesh or grid.

Scalar data sets have one value for each entity in a mesh or grid. Vector data sets may have either two (x, y) or three components (x, y, and z) depending on whether the data are 2- or 3-D.

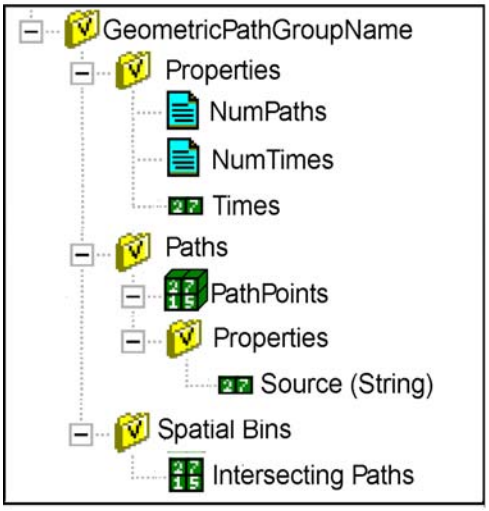


Figure 4. Geometric path group layout

Like XMDF files themselves, data set folders are organized into directory-like group folders. In each group all subfolder names must be unique. Every scalar or vector data set has its own folder.

A data set may be steady state or time varying. Time-varying data sets may begin at a specific reference time or be relative times from an arbitrary zero hour. Reference times are specified in Julian days. Time values for specific time-steps are given as time offsets from either zero or from the reference time. The units for the offset times can be given in days, hours, minutes, or seconds.

2.4.6 Coordinate system groups

The coordinate system group contains several items that define the coordinate system. Not all of the items are necessary to define a coordinate system. Figure 5 shows the group members.

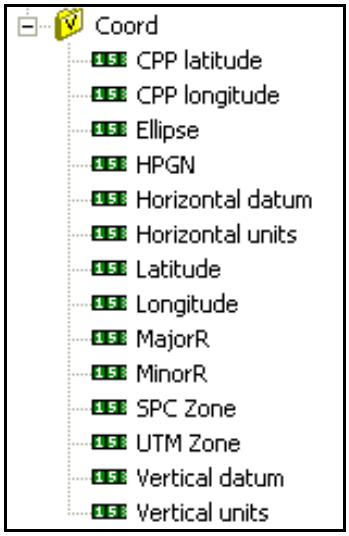


Figure 5. Schematic of coordinate system group

2.5 API Overview

The API defined in this report provides a set of C function calls and FORTRAN subroutines that can be used by model developers to store and retrieve data used in their model from an XMDF file. The use of these function calls will be illustrated by sample utility programs. The API includes functions/ subroutines to open and close project files, add and retrieve groups in a project, and add and retrieve variables or parts of variables in a group. The API is constructed as a layer that operates on top of the HDF5 API. While the user will need to be familiar with the HDF5 organizational structure and methodology, he or she will not make direct calls to the HDF5 library.

The XMDF API consists of both a FORTRAN module and a C module. This will allow model developers to utilize XMDF from C, C++, or FORTRAN analysis engines. A complete list of the C function and FORTRAN subroutine names can be found in Chapter 4.

3 Quick Start for Model Developers

The XMDF library contains a large amount of functions because it has to handle the needs of various model definitions as well as pre- and postprocessors. The developer of a modeling code generally will need to use only a small subset of these functions. This chapter will discuss the general strategy to incorporate XMDF in a specific model and will introduce the most commonly used functions.

3.1 XMDF File Paths

HDF5 files are organized in a directory-like structure. Files created by the XMDF library are not required to have a specific organization. Figure 6 is a screenshot of an XMDF file created by Surface-water Modeling System (SMS) visualized inside NCSA's HDFView browser. An attribute is written to files written by XMDF within each group that identifies the type of data stored in the group. Generic groups have no specific type of data associated with them and are used to organize the file. Special group types include mesh, grid, multiple data sets, scalar data set, vector data set, and property groups. Multi-data-set groups are special groups that hold data sets that all must belong to a corresponding spatial data object (mesh or grid). The other group types should be self-explanatory. Models relying upon XMS, the overarching modeling environment that supports the three modeling systems—SMS, the Groundwater Modeling System (GMS), and the Watershed Modeling System (WMS)—to create XMDF files for geometry input should not expect a specific file organization because the file format may change without warning. Instead, the model developer should work with EMRL to have the XMS package write out paths to required objects to an external file or use command line arguments. For models without a specific XMS interface, the necessary information can be determined for an individual file using an HDF5 browser.

3.2 Reading Geometry

Models reading geometry from XMDF files need to know the filename and the path inside the file to the geometry group. In the example in Figure 6, this

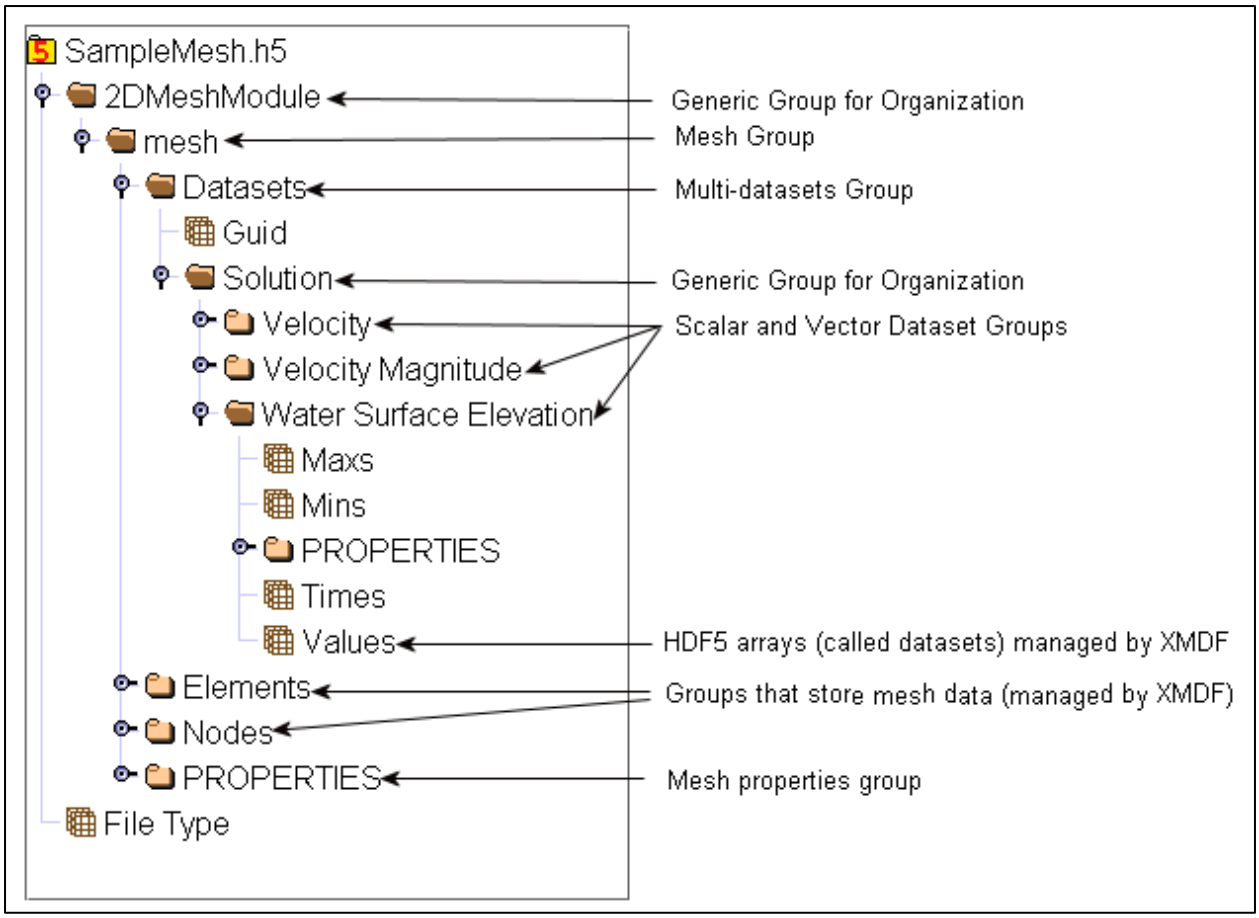


Figure 6. XMDF file containing a mesh created by SMS

path is **2DMeshModule/mesh.** The path is case-sensitive and uses the forward slash similar to a UNIX file path.

3.2.1 Meshes

This section is intended to be a quick overview of reading and writing meshes. For more detailed function descriptions see Section 4.9. The mesh geometry in XMDF files is relatively straightforward. The mesh contains nodes with x-, y-, and z-coordinates. The mesh also contains information about how the nodes are connected to form elements. Each element has an element type and a list of nodes belonging to the element. The nodes are referenced by their location in the node data set. Note: The node locations used in this context are always one-based (i.e., start at 1 rather than 0). The types of elements and number of nodes associated with each type are discussed in 0. The following steps are used to read a mesh:

a. Open the file and the mesh group using the functions **xfOpenFile** and **xfOpenGroup**.

b. Read the number of nodes, number of elements, and the maximum number of nodes in an element. These values are retrieved using the functions **xfGetNumberOfNodes**, **xfGetNumberOfElements**, and **xfGetMaxNodesInElem**.

c. Allocate arrays to store the node locations, element type, and connectivity arrays.

d. Read the node locations, element types, and connectivity arrays. These arrays are read using the functions **xfReadXNodeLocations**, **xfReadYNodeLocations**, **xfReadZNodeLocations**, **xfReadElemTypes**, and **xfReadElemNodeIds**.

e. Close the file and mesh group using **xfCloseFile** and **xfCloseGroup**.

f. Convert the arrays to native model data definitions if necessary.

C++ code

The following sample C++ code illustrates how to read a mesh using XMDF:

```
int fg_nElems;
int fg_nNodes;
int fg_nNodesPerElem;
int *fg_ElemTypes;
double *fg_XNodeLocs, *fg_YNodeLocs, *fg_ZNodeLocs;
int *fg_NodesInElem;

int xfReadMesh(const char *FileName, const char *PathToMesh) {
  xid    xFileId, xGroupId;
  int    nElemType, nNodeId;
  int    status;

  // Open the file and the mesh group
  status = xfOpenFile(FileName, &xFileId, TRUE);
  if (status < 0) {
    return -1;
  }

  if (status >= 0) {
    status = xfOpenGroup(xFileId, PathToMesh, &xGroupId);
  }
  if (status < 0) {
    // group was not opened successfully
    xfCloseFile(xFileId);
    return -1;
  }


  // Get the number of elements, nodes, and Maximum number of nodes per element
  status = xfGetNumberOfElements(xGroupId, &fg_nElems);
  if (status >= 0) {
    status = xfGetNumberOfNodes(xGroupId, &fg_nNodes);
    if (status >= 0) {
      status = xfGetMaxNodesInElem(xGroupId, &fg_nNodesPerElem);
    }
  }
  if (status < 0) {
    return -1;
  }

  fg_ElemTypes = new int[fg_nElems];
```

```
  if (fg_ElemTypes == NULL) {
    printf("Memory Error");
    return -1;
  }
  status = xfReadElemTypes(xGroupId, fg_nElems, fg_ElemTypes);
  if (status < 0) {
    return -1;
  }

  // Nodes in each element
  fg_NodesInElem  = new int[fg_nElems*fg_nNodesPerElem];
  xfReadElemNodeIds(xGroupId, fg_nElems, fg_nNodesPerElem, fg_NodesInElem);

  // NodeLocations
  fg_XNodeLocs = new double[fg_nNodes];
  fg_YNodeLocs = new double[fg_nNodes];
  fg_ZNodeLocs = new double[fg_nNodes];
  if (fg_XNodeLocs == NULL || fg_YNodeLocs == NULL || fg_ZNodeLocs == NULL) {
    if (fg_XNodeLocs != NULL) {
      delete fg_XNodeLocs;
      fg_XNodeLocs = NULL;
    }
    if (fg_YNodeLocs != NULL) {
      delete fg_YNodeLocs;
      fg_YNodeLocs = NULL;
    }
    if (fg_ZNodeLocs != NULL) {
      delete fg_ZNodeLocs;
      fg_ZNodeLocs = NULL;
    }
    printf("Memory Error!");
    return -1;
  }

  status = xfReadXNodeLocations(xGroupId, fg_nNodes, fg_XNodeLocs);
  if (status >= 0) {
    status = xfReadYNodeLocations(xGroupId, fg_nNodes, fg_YNodeLocs);
    if (status >= 0) {
      status = xfReadZNodeLocations(xGroupId, fg_nNodes, fg_ZNodeLocs);
    }
    else {
      return -1;
    }
  }
  else {
    return -1;
  }
}
```

FORTRAN

```
INTEGER   fg_nElems
INTEGER   fg_nNodes
INTEGER   fg_nNodesPerElem
INTEGER, ALLOCATABLE :: fg_ElemTypes
REAL(DOUBLE), ALLOCATABLE :: fg_XNodeLocs(:), fg_YNodeLocs(:), fg_ZNodeLocs(:)
INTEGER, ALLOCATABLE :: fg_NodesInElem(:)

SUBROUTINE READ_MESH (Filename, PathToMesh, error)
CHARACTER(LEN=*), INTENT(IN) :: Filename, PathToMesh
INTEGER, INTENT(OUT)       :: error
INTEGER(HID_T)  xFileId, xGroupId
INTEGER         nElemType, nNodeId
```

```
! Open the file and group
call XF_OPEN_FILE(Filename, XTRUE, xFileId, error)
if (error .LT. 0) then
  return
endif

call XF_OPEN_GROUP(xFileId, PathToMesh, xGroupId, error)
if (error .LT. 0) then
  return
endif

! Get the number of elements, nodes, and Maximum number of nodes per element
call XF_GET_NUMBER_OF_ELEMENTS (xGroupId, fg_nElems, error)
if (error >= 0) then
  call XF_GET_NUMBER_OF_NODES (xGroupId, fg_nNodes, error)
  if (error >= 0) then
    call XF_GET_MAX_NODES_IN_ELEM (xGroupId, fg_nNodesPerElem, error)
  endif
endif

if (error < 0) then
  return
endif

  ! Element types
allocate (fg_ElemTypes(fg_nElems))

call XF_READ_ELEM_TYPES (xGroupId, fg_nElems, fg_ElemTypes, error)
if (error < 0) then
  return
endif

  ! Nodes in each element
allocate (fg_NodesInElem(fg_nElems*fg_nNodesPerElem))
call XF_READ_ELEM_NODE_IDS (xGroupId, fg_nElems, fg_nNodesPerElem, &
                            fg_NodesInElem, error)

  ! NodeLocations
allocate (fg_XNodeLocs(fg_nNodes))
allocate (fg_YNodeLocs(fg_nNodes))
allocate (fg_ZNodeLocs(fg_nNodes))

call XF_READ_X_NODE_LOCATIONS (xGroupId, fg_nNodes, fg_XNodeLocs, error)
if (status >= 0) then
  call XF_READ_Y_NODE_LOCATIONS (xGroupId, fg_nNodes, fg_YNodeLocs, error)
  if (status >= 0) then
    call XF_READ_Z_NODE_LOCATIONS (xGroupId, fg_nNodes, fg_ZNodeLocs, error)
  endif
endif

error = TRUE
return

END SUBROUTINE
```

3.2.2 Grids

The XMDF library supports many different types of grids. Grids can be 2- or 3-D. Two-dimensional grids may be Cartesian or curvilinear. Three-dimensional grids may be Cartesian, curvilinear, or extruded 2-D Cartesian or curvilinear grids. Extruded grids may be sigma-stretch, curvilinear, curvilinear at corners, or curvilinear at midsides. Grid properties include the origin, orientation, dip, and bearing. The way the grid geometry is specified depends upon the type of grid.

For a Cartesian grid, the gridline locations along each axis (I, J, and K if applicable) must be specified. These locations are defined as the distance along the axis from the grid origin. For a curvilinear grid, the locations of each grid corner must be defined. The locations are given in world coordinates.

To read a grid from an XMDF file:

a. Open the file and the grid group using **xfOpenFile** and **xfOpenGroup**.

b. Make sure that the grid type and properties are valid for the particular model. If the model requires a 3-D curvilinear grid, abort if the grid is any other type. Functions that may be used include **xfGetGridType**, **xfGetExtrusionType**, **xfGetNumberOfDimensions**.

c. Get the number of cells in each direction using **xfGetNumberCellsInI**, **xfGetNumberCellsInJ**, and **xfGetNumberCellsInK**.

d. Allocate the arrays for the grid geometry definition. Remember that the size of the arrays depends not only upon the number of cells in the grid but also the type of grid.

e. Read the grid geometry using **xfGetGridCoordsI**, **xfGetGridCoordsJ**, and **xfGetGridCoordsK**.

f. Close the file and grid group using **xfCloseFile** and **xfCloseGroup**.

g. Convert grid information into native model data definitions if necessary.

The following examples have a function to read data for a model that requires a 3-D Cartesian grid for input. The model uses a bearing, but dip and roll values are ignored. The filename and path to the grid are given, and the data are stored in variables accessible outside the function (file globals in C/C++, common blocks in FORTRAN). The function follows the XMDF convention of using negative values to indicate errors.

C/C++

```
int       fg_nCellsI, fg_nCellsJ, fg_nCellsK;
double    fg_Origin[3], fg_Bearing;
double   *fg_CoordI, *fg_CoordJ, *fg_CoordK;

int ReadGrid (const char *a_Filename, const char *a_GridPath)
{
  xid      xFileId, xGroupId;
  int      nGridType = 0, nDims = 0;
  int      nValsI = 0, nValsJ = 0;
  int      nValsK = 0;
  xbool    bDefined = XFALSE;
  int      i = 0, error = 1;

  // open the file and group
  error = xfOpenFile(a_Filename, &xFileId, TRUE);
  if (error < 0) {
    printf("Unable to open file.");
    return -1;
  }

  error = xfOpenGroup(xFileId, a_GridPath, &xGroupId);
```

```
if (error < 0) {
  printf("Unable to open the group.");
  xfCloseFile(xFileId);
  return -1;
}

// Grid type
error = xfGetGridType(a_Id, &nGridType);
if (error < 0) {
  return error;
}
if (nGridType != GRID_TYPE_CARTESIAN) {
  printf("Unsupported grid type. Must be a Cartesian Grid");
  return -1;
}
// Number of dimensions
error = xfGetNumberOfDimensions(a_Id, &nDims);
if (error < 0) {
  return error;
}
if (nDims != 3) {
  printf("Error:  The grid must be a three-dimensional grid.");
  return -1;
}

// Origin
error = xfOriginDefined(a_Id, &bDefined);
if (error < 0) {
  return error;
}
if (bDefined) {
  error = xfGetOrigin(a_Id, &fg_Origin[0], &fg_Origin[1], &fg_Origin[2]);
  if (error < 0) {
    return error;
  }
}

// Bearing
error = xfBearingDefined(a_Id, &bDefined);
if (error < 0) {
  return error;
}
if (bDefined) {
  error = xfGetBearing(a_Id, &fg_Bearing);
  if (error < 0) {
    return error;
  }
}

// number of cells in each direction
error = xfGetNumberCellsInI(a_Id, &nCellsI);
if (error >= 0) {
  error = xfGetNumberCellsInJ(a_Id, &nCellsJ);
  if (error >= 0 && nDims == 3) {
    error = xfGetNumberCellsInK(a_Id, &nCellsK);
  }
}
if (error < 0) {
  return error;
}

nValsI = fg_nCellsI;
nValsJ = fg_nCellsJ;
nValsK = fg_nCellsK;

fg_CoordI = new double[nValsI];
fg_CoordJ = new double[nValsJ];
```

```
  fg_CoordK = new double[nValsK];

  error = xfGetGridCoordsI(a_Id, nValsI, fg_CoordI);
  if (error >= 0) {
    error = xfGetGridCoordsJ(a_Id, nValsJ, fg_CoordJ);
    if (error >= 0 && nDims == 3) {
      error = xfGetGridCoordsK(a_Id, nValsK, fg_CoordK);
    }
  }
  if (error < 0) {
    printf("Error reading coordinates.\n");
    return -1;
  }

  // if we got here, everything read in fine
  return 1;
} // ReadGrid
```

FORTRAN

```
INTEGER      fg_nCellsI, fg_nCellsJ, fg_nCellsK
REAL(DOUBLE), DIMENSION(3) :: fg_Origin
REAL(DOUBLE), ALLOCATABLE :: fg_CoordI(:), fg_CoordJ(:), fg_CoordK(:)
REAL(DOUBLE)              :: fg_Bearing

SUBROUTINE  READ_GRID(a_Filename, a_GridPath, error)
CHARACTER(LEN=*), INTENT(IN) :: a_Filename, a_GridPath
INTEGER, INTENT(OUT)       :: error
INTEGER(XID)   xFileId, xGroupId
INTEGER nGridType, nDims;
INTEGER nValsI, nValsJ, nValsK
INTEGER bDefined

call XF_OPEN_FILE(a_Filename, XTRUE, xFileId, error)
if (error < 0) then
  return
endif

call XF_OPEN_GROUP(xFileId, a_GridPath, xGroupId, error)
if (error < 0) then
  XF_CLOSE_FILE(xFileId, error)
  return
endif

  ! Grid type
call XF_GET_GRID_TYPE(a_Id, nGridType, error)
if (error < 0) then
  return
endif
if (nGridType .EQ. GRID_TYPE_CARTESIAN) then
  Write(*,*) 'Unsupported grid type.  Must be a Cartesian Grid'
  return
endif

  ! Number of dimensions
call XF_GET_NUMBER_OF_DIMENSIONS(a_Id, nDims, error)
if (error .LT. 0) then
  return
endif
if (nDims .NE. 3) then
  WRITE(*,*) 'The grid must be a three-dimensional grid'
  error = -1
  return
endif
```

```
  ! Origin
call XF_ORIGIN_DEFINED(a_Id, bDefined, error)
if (error < 0) then
  return
endif
if (bDefined /= 0) then
  call XF_GET_ORIGIN(a_Id, fg_Origin(1), fg_Origin(2), fg_Origin(3), error)
  if (error < 0) then
    return
  endif
endif

  ! Bearing
call XF_BEARING_DEFINED(a_Id, bDefined, error)
if (error < 0) then
  return
endif
if (bDefined /= 0) then
  call XF_GET_BEARING(a_Id, fg_Bearing, error)
  if (error < 0) then
    return
  endif
endif

  ! number of cells in each direction
call XF_GET_NUMBER_CELLS_IN_I(a_Id, fg_nCellsI, error)
if (error >= 0) then
  call XF_GET_NUMBER_CELLS_IN_J(a_Id, fg_nCellsJ, error)
  if (error > 0) then
    call XF_GET_NUMBER_CELLS_IN_K(a_Id, fg_nCellsK, error)
  endif
endif
if (error < 0) then
  return
endif

nValsI = fg_nCellsI
nValsJ = fg_nCellsJ
nValsK = fg_nCellsK

ALLOCATE(fg_CoordI(nValsI))
ALLOCATE(fg_CoordJ(nValsJ))
ALLOCATE(fg_CoordK(nValsK))

call XF_GET_GRID_COORDS_I(a_Id, nValsI, fg_CoordI, error)
if (error >= 0) then
  call XF_GET_GRID_COORDS_J(a_Id, nValsJ, fg_CoordJ, error)
  if ((error > 0) .AND. (nDims == 3)) then
    call XF_GET_GRID_COORDS_K(a_Id, nValsK, fg_CoordK, error)
  endif
endif
if (error < 0) then
  WRITE(*,*) 'Error reading coordinates'
  error = -1
  return
endif

! Return successful if we got here
error = 1
return
END SUBROUTINE TG_READ_GRID
```

3.3 Writing Data Sets

This section gives a quick overview of reading and writing data sets. For more detailed function descriptions see Section 4.12. Setting up a model to write data sets to an XMDF file requires more information than is needed for reading geometry. Part of the reason more information is required is because more of the file organization is created by the model. Another source of complexity is that data sets can be written to either new or existing files.

Individual data sets are grouped in special folders that hold all the data sets for a specific geometry. These folders are called **multi-data-set** groups. Multi-data-set groups may or may not be in the same file as the geometry. In Figure 6 the **multi-data-set** group for the mesh has the path **2DMeshModule/mesh/Datasets**. Multi-data-set groups store the globally unique identifier (GUID) of the geometry to which they belong. Within a multi-data-set group, generic groups may be used to organize the data sets. In Figure 6 the folder named **Solution** under the multi-data-set group is an example of using generic groups to organize data sets. These generic groups would be especially useful for stochastic simulations and could be used to organize the data sets belonging to individual runs.

When writing data sets using XMDF the following items of information are necessary:

a. The filename to write the data to.

b. The path for multi-data-set group to write the data sets.

c. The GUID for the geometry that the data sets belong to.

d. The path to a generic group inside the multi-data-set group (may be blank).

e. What existing data to clear before saving. If the model is writing to a file that is intended only for solutions for the particular simulation, the entire file should be overwritten. If the data sets are to be written to an existing XMDF file, either the folder to write the data sets should be cleared or only existing data sets with the same name as the data sets that will be written should be cleared. Each of these options is supported.

Inside XMDF there is a function to make it easier for models to set up the paths to begin writing data sets. This function takes for arguments the five items mentioned previously and returns the File ID and the ID of the group to begin writing data sets to. This function should always be used to set up a model to begin writing data sets using XMDF because it ensures that everything is set up regardless of whether or not the file or the required paths exist. The function is named **xfSetupToWriteDatasets**.

Once the File ID and Group ID to start writing data sets has been obtained from **xfSetupToWriteDatasets**, the following steps are used to add each data set to the file:

a. Create the data set using **xfCreateScalarDataset** or **xfCreateVectorDataset**.

b. If desired, use the function **xfDatasetReftime** to specify the reference time for the data set. The reference time is the Julian date for time zero for the data set. Each time-step is treated as an offset from this reference time.

c. For each time-step, set the data using **xfWriteScalarTimestep**. If necessary, use **xfWriteActivityTimestep** to set active element information for each time-step.

d. Close the data set using **xfCloseGroup**.

e. When all data sets are closed, close the group that all the data sets are written to using **xfCloseGroup** and close the file using **xfCloseFile**.

Often steps 1 and 2 will be done together for all of the data sets to be written at the beginning of the model execution. Then when time-steps are computed, step 3 will be performed for each data set. When all the time-steps are completed, steps 4 and 5 are used to close the file and all the resources.

The following example has a function to write a scalar data set to an XMDF file. The arguments to the function are those included in the **xfSetupToWriteDatasets,** a compression level (–1 for none). The number of times, time values, number of values per time-step, and the values for the data set are stored in file globals. The function follows the XMDF convention of using negative values to indicate errors.

C/C++

```
static int    fg_nTimes;
static double *fg_Times;
static int    fg_nValues;
static float  **fg_Values;

int WriteDataset (const char *a_Filename,
                  const char *a_MultiDatasetsGroupPath,
                  const char *a_SpatialDataObjectGuid,
                  int a_OverwriteOption,
                  int a_Compression)
{
  xid      xFileId, xDsetsId, xScalarId;
  int      iTimestep, iActive;
  double   dTime;
  int      status;

  status = xfSetupToWriteDatasets(a_Filename, a_MultiDatasetsGroupPath,
              a_SpatialDataObjectGuid, a_OverwriteOption, &xFileId, &xDsetsId);
  if (status < 0) {
    printf("Error initializing files to write datasets.");
    return status;
  }

  // Create the scalar A dataset group
  status = xfCreateScalarDataset(xDsetsId, "Concentration",
                                 "mg/L", TS_HOURS, a_Compression,
                                 &xScalarId);
  if (status < 0) {
```

```
    printf("Error creating scalar datasets.");
    xfCloseGroup(xDsetsId);
    xfCloseFile(xFileId);
    return FALSE;
  }

  // Loop through timesteps adding them to the file
  for (iTimestep = 0; iTimestep < fg_nTimes; iTimestep++) {
    // We will have an 0.5 hour timestep
    dTime = fg_Times[iTimestep];

    // write the dataset array values
    status = xfWriteScalarTimestep(xScalarAId, dTime, fg_nValues,
                                   fg_Values[iTimestep]);
    if (status < 0) {
      xfCloseGroup(xScalarId);
      xfCloseGroup(xDsetsId);
      xfCloseFile(xFileId);
      return status;
    }
  }

  // close the dataset
  xfCloseGroup(xScalarId);
  xfCloseGroup(xDsetsId);
  xfCloseFile(xFileId);

  return FALSE;
} // tdWriteScalarA
```

FORTRAN

```
INTEGER         fg_nTimes;
REAL(DOUBLE), ALLOCATABLE ::  fg_Times(:)
INTEGER         fg_nValues
REAL, ALLOCATABLE ::  fg_Values(:,:)

SUBROUTINE WRITE_DATASET (a_Filename,
                          a_MultiDatasetsGroupPath,
                          a_SpatialDataOjbectGuid,
                          a_OverwriteOption,
                          a_Compression, error)
  CHARACTER(LEN=*), INTENT(IN) :: a_Filename, a_MultiDatasetsGroupPath
  CHARACTER(LEN=*), INTENT(IN) :: a_SpatialDataObjectGuid
  INTEGER, INTENT(IN)          :: a_OverwriteOption, a_Compression
  INTEGER, INTENT(OUT)         :: error
  INTEGER(HID_T)      xFileId, xDsetsId, xScalarId
  INTEGER      iTimestep, iActive
  REAL(DOUBLE) dTime
  REAL, ALLOCATABLE ::         Values(:)

  call XF_SETUP_TO_WRITE_DATASETS(a_Filename, a_MultiDatasestsGroupPath, &
                                  a_SpatialDataObjectGuid, a_OverwriteOption, &
                                  xFileId, xScalarId, error)

  if (error .LT. 0) then
    WRITE(*,*) 'Error initializing files to write datasets.'
    return
  endif

  ! Create the scalar A dataset group
  call XF_CREATE_SCALAR_DATASET(xDsetsId, 'Concentration', 'mg/L', &
             TS_HOURS, a_Compression, xScalarAd, error)
  if (error .LT. 0) then
      ! close the dataset
```

```
    call XF_CLOSE_GROUP(xScalarId, error)
    call XF_CLOSE_GROUP(xDsetsId, error)
    call XF_CLOSE_FILE(xFileId, error)
    return
  endif

  ! allocate the values for an individual timestep
  allocate (Values(fg_nValues))

  ! Loop through timesteps adding them to the file
  do iTimestep = 1, fg_nTimes
    ! We will have an 0.5 hour timestep
    dTime = fg_Times(iTimestep)

    do iVal = 1, fg_nValues
      Values(iVal) = fg_Values(iTimestep, iVal)
    enddo

    ! write the dataset array values
    call XF_WRITE_SCALAR_TIMESTEP(xScalarId, dTime, fg_nValues, &
                                  Values, error)
    if (error .LT. 0) then
      deallocate (Values)
      call XF_CLOSE_GROUP(xScalarAId, error)
      call XF_CLOSE_GROUP(xDsetsId, error)
      call XF_CLOSE_FILE(xFileId, error)
      return
    endif
  enddo

  deallocate (Values)

  ! close the dataset
  call XF_CLOSE_GROUP(xScalarAId, error)
  call XF_CLOSE_GROUP(xDsetsId, error)
  call XF_CLOSE_FILE(xFileId, error)

  return
END SUBROUTINE
```

4 Implementation Design

The following sections describe the implementation of the API for XMDF.

4.1 XMDF Functions/Subroutines

All function/subroutine names begin with the letters **xf** to avoid likely conflicts with other function names. The memory for all variables must be allocated outside the library. This is done so users can reuse memory, and users are more likely to deallocate memory if they have to allocate it.

4.1.1 C/C++ Interface

All functions return an integer status value. A zero or positive values indicate success and negative values indicate failure.

All arrays in the C library must be contiguous blocks of memory. This means that 2- and 3-D arrays must be allocated with a single allocation with the size necessary to store all the information at one time.

4.1.2 FORTRAN Interface

The entire FORTRAN interface uses subroutines rather than functions. The final argument is always an integer that is set to a negative number if an error occurs. The FORTRAN interface is available as a module to other model codes.

4.2 Compression

All functions that write arrays provide a compression option represented by an integer. A value of –1 for the compression option is no compression. A compression option between 0 and 9 indicates the compression level where 0 is the minimum compression and 9 is the maximum compression. Higher compression levels result in smaller files but are slower.

4.3 Version Number

When a file is opened to write XMDF data, a version number is written out. Future versions of the XMDF API will allow backward compatibility to previous versions. A function exists to retrieve which version of the library is being used (currently linked library). Another function exists to retrieve which version of the library wrote a specific file. Both items are important because although future versions of the library will read files written by older versions of the library, it cannot be guaranteed that older versions of the library will read files generated by future versions of the library correctly.

C/C++

```
int xfGetLibraryVersion(float *Version);
int xfGetLibaryVersionFile(xid File, float *Version);
```

FORTRAN

```
SUBROUTINE XF_GET_LIBRARY_VERSION(Version, Error)
REAL, INTENT(OUT)          ::   Version
INTEGER, INTENT(OUT)       ::   Error
SUBROUTINE XF_GET_LIBRARY_VERSION_FILE(File, Version, Error)
INTEGER(XID), INTENT(IN)  ::     FileId
REAL, INTENT(OUT)          ::   Version
INTEGER, INTENT(OUT)       ::   Error
```

4.4 Creating and Opening Files

The library provides functions to create, open, and close files. Each function uses or fills in a variable of type **xid** that identifies the file to the API.

The **xid** type is the same type as HDF5's **hid_t**. Variables of type **xid** are used to access all files, groups, and arrays stored in XMDF files.

C/C++

```
int xfCreateFile(const char *Name, xid *FileId, xbool Overwrite);
int xfOpenFile(const char *Name, xid *FileId, xbool Readonly);
int xfCloseFile(xid FileId);
```

FORTRAN

```
SUBROUTINE XF_CREATE_FILE(Name, Overwrite, FileId, Error)
CHARACTER(LEN=*), INTENT(IN):: Name
LOGICAL, INTENT(IN)        ::   Overwrite
INTEGER(XID), INTENT(OUT)  ::   FileId
INTEGER, INTENT(OUT)       ::   Error

SUBROUTINE XF_OPEN_FILE(Name, FileId, Error)
CHARACTER(LEN=*), INTENT(IN)  ::  Name
LOGICAL, INTENT(IN)           ::  ReadOnly
```

```
INTEGER(XID), INTENT(OUT)     ::  FileId
INTEGER, INTENT(OUT)          ::  Error

SUBROUTINE XF_CLOSE_FILE(FileId, Error)
INTEGER(XID), INTENT(IN) ::     FileId
INTEGER, INTENT(OUT) ::         Error
```

4.5 Float Variable Types

HDF5 automatically performs necessary conversions between different types of floats. For example, if a file is being read on a UNIX computer that represents floats with big-endian notation and the floats in the file are represented using little-endian notation, HDF5 will convert to big-endian. This conversion takes extra time when reading the data. Since data are likely going to be read more times than it was written (data are written only once), it is generally preferable to write the data in the format that it will be postprocessed in. The XMDF library provides a function that can be used to specify how the floats should be represented in the file (little-endian is the default). Call the function and pass XFALSE to write floating point numbers in big-endian notation.

C/C++

```
int xfWriteFloatsAsLittleEndian(xbool LittleEndian);
```

FORTRAN

```
SUBROUTINE XF_WRITE_FLOATS_AS_LITTLE_ENDIAN(LittleEndian, Error)
CHARACTER(LEN=*), INTENT(IN):: Name
LOGICAL, INTENT(IN)        ::   LittleEndian
INTEGER, INTENT(OUT)       ::   Error
```

4.6 XMDF Groups

As mentioned previously, the file structure for HDF5 files consists of a hierarchical tree of groups. Each group may contain variables or other groups. This section defines the variables that a group must include for a particular type of data (2-D mesh geometry, grid geometry, data sets, etc.). A particular path that these items must be stored in (like **/Meshes/2D/MeshA**) is not specified since these locations may change in the future (in order to include multiple scenarios, etc.). Instead, functions have been created inside the API to determine paths for a group. Every group includes a “Grouptype string” that identifies the type of information contained in the group. The grouptypes and their meanings are given in Table 1.

All functions that read from or write to a group use an identifier to the group. This identifier is of type **xid** and is obtained by creating or opening a group.

Table 1
Grouptypes Defined as Part of XMDF

Grouptype	Description
MESH	Any mesh type including 0D, 1-, 2-, and 3-D elements. Meshes are allowed to contain mixtures of elements of different dimensions
GRID	Any grid including Cartesian, curvilinear, and composites of the two
XSECS	Cross-section information
PROPERTIES	Properties belonging to meshes, grids, elements, nodes, etc.
GENERIC	A group used for organization. Is not tied to a particular type of data

The following API functions are used to create groups for meshes, grids, and cross sections. Creating data set and attribute groups will be discussed later because they are associated with a specific grid or mesh. The path is a string with slashes (/) to specify the groups. The identifier is the file identifier received from the **open file** or **close file** command. The **create group** operation for data sets should be used only when writing data sets for a mesh, grid, or cross sections that are not defined in the file.

C/C++

```
int xfCreateGroupForMesh(xid FileId, const char *Path, xid *GroupId);
int xfCreateGroupForGrid(xid FileId, const char *Path, xid *GroupId);
int xfCreateGroupForXsecs(xid FileId, const char *Path, xid *GroupId);
int xfCreateGenericGroup(xid FileId, const char *Path, xid *GroupId);
```

FORTRAN

```
SUBROUTINE XF_CREATE_GROUP_FOR_MESH(FileId, Path, GroupId, Error)
INTEGER(XID), INTENT(IN) ::    FileId
CHARACTER(LEN=*), INTENT(IN):: Path
INTEGER(XID), INTENT(OUT) ::   GroupId
INTEGER, INTENT(OUT)  ::       Error

SUBROUTINE XF_CREATE_GROUP_FOR_GRID (FileId, Path, GroupId, Error)
INTEGER(XID), INTENT(IN) ::    FileId
CHARACTER(LEN=*), INTENT(IN) :: Path
INTEGER(XID), INTENT(OUT) ::   GroupId
INTEGER, INTENT(OUT)  ::       Error

SUBROUTINE XF_CREATE_GROUP_FOR_XSECS (FileId, Path, GroupId,
Error)
INTEGER(XID), INTENT(IN) ::    FileId
CHARACTER(LEN=*), INTENT(IN):: Path
INTEGER(XID), INTENT(OUT) ::   GroupId
INTEGER, INTENT(OUT)   ::      Error

SUBROUTINE XF_CREATE_GENERIC_GROUP (FileId, Path, GroupId, Error)
INTEGER(XID), INTENT(IN) ::    FileId
CHARACTER(LEN=*), INTENT(IN):: Path
INTEGER(XID), INTENT(OUT) ::   GroupId
INTEGER, INTENT(OUT)   ::      Error
```

The following API function is used to open existing groups for reading/ writing:

C/C++

```
int xfOpenGroup(xid FileId, const char *path, xid *GroupId);
```

FORTRAN

```
SUBROUTINE XF_OPEN_GROUP(FileId, Path, GroupId, Error)
INTEGER(XID), INTENT(IN) ::     FileId
CHARACTER(LEN=*), INTENT(IN) :: Path
INTEGER(XID), INTENT(OUT) ::    GroupId
INTEGER, INTENT(OUT)    ::      Error
```

When the group is no longer needed, the following call must be made to close the group:

C/C++

```
int xfCloseGroup(xid GroupId);
```

FORTRAN

```
SUBROUTINE XF_CLOSE_GROUP(GroupId, Error)
INTEGER(XID), INTENT(IN) :: GroupId
INTEGER, INTENT(OUT)  ::   Error
```

4.7 Determining All Entities in a File

Sometimes the program reading the XMDF file will have the path where data reside given to them. When the user is not looking for a specific entity or does not know the path for the group, the API provides the following functions to get paths to specific groups. These functions traverse the groups in the file and look for the grouptype flags. The first function in each set is used to determine the required size of the character arrays that hold the paths. The path passed into the second function in each set must be allocated to a size Num*MaxSize. Data-set groups beneath meshes or grids will not be returned and must be accessed through the mesh or grid to which they belong.

C/C++

```
int xfGetGroupPathsSizeForMeshes(xid FileId, int *Num, int *MaxSize);
int xfGetAllGroupPathsForMeshes(xid FileId, int Num, int MaxSize, char *path);

int xfGetGroupPathsSizeForGrids(xid FileId, int *Num, int *MaxSize);
int xfGetAllGroupPathsForGrids(xid FileId, int Num, int MaxSize, char *path);
```

```
int xfGetGroupPathsSizeForXsecs(xid FileId, int *Num, int *MaxSize);
int xfGetAllGroupPathsForXsecs(xid FileId, int Num, int MaxSize, char *path);
```

FORTRAN

```
SUBROUTINE XF_GET_ALL_GROUP_PATHS_SIZE_FOR_MESHES(FileId, Num, Maxsize, Error)
INTEGER(XID), INTENT(IN) ::   FileId
INTEGER, INTENT(OUT) ::       Num, Maxsize, Error

SUBROUTINE XF_GET_ALL_GROUP_PATHS_FOR_MESHES(FileId, Num, Size, Paths, Error)
INTEGER(XID), INTENT(IN) ::     FileId
INTEGER, INTENT(IN) ::  Num, Size
CHARACTER(len=Size), DIMENSION(Size, Num), INTENT(INOUT) ::  Paths
INTEGER, INTENT(OUT) :: Error


SUBROUTINE XF_GET_ALL_GROUP_PATHS_SIZE_FOR_GRIDS (FileId, Num, Maxsize, Error)
INTEGER(XID), INTENT(IN) ::   FileId
INTEGER, INTENT(OUT) ::       Num, Maxsize, Error

SUBROUTINE XF_GET_ALL_GROUP_PATHS_FOR_GRIDS (FileId, Num, Size, Paths, Error)
INTEGER(XID), INTENT(IN) ::     FileId
INTEGER, INTENT(IN) ::  Num, Size
CHARACTER(len=Size), DIMENSION(Size, Num), INTENT(INOUT) ::  Paths
INTEGER, INTENT(OUT) :: Error

SUBROUTINE XF_GET_ALL_GROUP_PATHS_SIZE_FOR_XSECS (FileId, Num, Maxsize, Error)
INTEGER(XID), INTENT(IN) ::   FileId
INTEGER, INTENT(OUT) ::       Num, Maxsize, Error

SUBROUTINE XF_GET_ALL_GROUP_PATHS_FOR_XSECS (FileId, Num, Size, Paths, Error)
INTEGER(XID), INTENT(IN) ::     FileId
INTEGER, INTENT(IN) ::  Num, Size
CHARACTER(len=Size), DIMENSION(Size, Num), INTENT(INOUT) ::  Paths
INTEGER, INTENT(OUT) :: Error
```

4.8 Properties

Properties are defined generically and can be assigned to any type of entity. Property groups are created under groups of other types (meshes, elements, nodes, grids, cells, etc). Properties can be integers, floats (single or double precision), bit on/off (stored as unsigned characters), or strings. The length of the array in the property variable can be the number of cells or elements in the group (one value per cell or element), the number of nodes in the group (one value per node), or a single value (one value for the group or the same value for all objects in the group). Properties must be in the same order in the array as the data objects in the group.

Only one property with a given name may exist in a property folder (for a particular group). The name is case-sensitive so be careful. String properties are passed as character arrays along with their length.

4.8.1 Reserved property names

Some property names are reserved for specific meanings. This is to ensure conformity for commonly used properties. The reserved names and their meanings are given in Table 2.

Table 2
Reserved Attribute Names and Descriptions

Property Name	Type	Description
Ids	Integer	Ids belonging to an entity (nodes, elements, etc.)
Material	Integer	Material properties for a list of elements, or cells
Activity	bit (on/off)	Whether an element, cell is part of computation

4.8.2 API functions for properties

Each of the functions returns a negative value for failure and a positive value for success. The following functions will retrieve the entire array of values for a property specified by its name. The id must be the id for the property group. The method to get the property group id will be given later.

4.8.2.1 Writing

The following functions are used to write properties. Property arrays are always one-dimensional arrays but can be any size. The size should be appropriate for the type of data stored. For example, if the property array stores the material property for a data set of elements, the size of the property array should be the same as the number of elements.

C/C++

```
int xfWritePropertyInt(xid GroupId, const char *Name, int Num, int
                       *Property, int Compression);
int xfWritePropertyFloat(xid GroupId, const char *Name, int Num, float
                       *Property, int Compression);
int xfWritePropertyDouble(xid GroupId, const char *Name, int Num, double
                       *Property, int Compression);
int xfWritePropertyString(xid GroupId, const char *Name, int Num, char
                       **Property, int StringLength, int Compression);
```

FORTRAN

```
SUBROUTINE  XF_WRITE_PROPERTY_INT(GroupId, Name, Num, Property, Compression,
Error);
INTEGER(XID), INTENT(IN) :: GroupId
CHARACTER(len=*), INTENT(IN) :: Name
INTEGER, INTENT(IN)      :: Num
INTEGER, INTENT(IN)      :: Property(*)
INTEGER, INTENT(IN)      :: Compression
INTEGER, INTENT(OUT)     :: Error
```

```
SUBROUTINE  XF_WRITE_PROPERTY_FLOAT(GroupId, Name, Num, Property, Compression,
Error);
INTEGER(XID), INTENT(IN) :: GroupId
CHARACTER(len=*), INTENT(IN) :: Name
INTEGER, INTENT(IN)      :: Num
REAL, INTENT(IN)      :: Property(*)
INTEGER, INTENT(IN)      :: Compression
INTEGER, INTENT(OUT)     :: Error


SUBROUTINE  XF_WRITE_PROPERTY_DOUBLE(GroupId, Name, Num, Property, Compression,
Error);
INTEGER(XID), INTENT(IN) :: GroupId
CHARACTER(len=*), INTENT(IN) :: Name
INTEGER, INTENT(IN)      :: Num
REAL(DOUBLE), INTENT(IN) :: Property(*)
INTEGER, INTENT(IN)      :: Compression
INTEGER, INTENT(OUT)     :: Error

SUBROUTINE  XF_WRITE_PROPERTY_STRING(GroupId, Name, Num, Property, Compression,
Error);
INTEGER(XID), INTENT(IN) :: GroupId
CHARACTER(len=*), INTENT(IN) :: Name
INTEGER, INTENT(IN)      :: Num
CHARACTER(len =*), INTENT(IN)    :: Property(*,*)
INTEGER, INTENT(IN)      :: Compression
INTEGER, INTENT(OUT)     :: Error
```

4.8.2.2 Reading

The following functions are used to determine what property names are used in the group or to see whether a property exists. The names of all properties are limited to 256 characters.

C/C++

```
int xfDoesPropertyWithNameExist(xid GroupId, const char *Name, int *Exists);
int xfGetPropertyNumber(xid GroupId, const char *Name, int *Num);
// Names must be a 2D array size NumberOfProperties by 256 (max should be 256)
int xfGetPropertyNames(xid GroupId, char **Names);
```

FORTRAN

```
SUBROUTINE  XF_DOES_PROPERTY_WITH_NAME_EXIST(GroupId, Name, Exists, Error);
INTEGER(XID), INTENT(IN) :: GroupId
CHARACTER(len=*), INTENT(IN) :: Name
LOGICAL, INTENT(OUT)     :: Exists
INTEGER, INTENT(OUT)     :: Error

SUBROUTINE  XF_GET_NUMBER_OF_PROPERTIES(GroupId, Number, Error);
INTEGER(XID), INTENT(IN) :: GroupId
INTEGER, INTENT(OUT)     :: Number, Error

SUBROUTINE  XF_GET_PROPERTY_NAMES(GroupId, Number, Names, Error);
INTEGER(XID), INTENT(IN) :: GroupId
CHARACTER(len=*), INTENT(OUT)  :: Names(*, Number)
INTEGER, INTENT(OUT)   ::    Error
```

There may be times when the type of a property is unknown. The following function allows the user to get the type of data that is stored in a property. The function sets the integer 'type'. The types, associated constants, and associated numbers are given in Table 3.

Table 3
Possible Property Types and the Number Associated With Them

Number	Constant	Type
1	XF_TYPE_INT	integer
2	XF_TYPE_FLOAT	single precision float (4 bytes)
3	XF_TYPE_DOUBLE	double precision float (8 bytes)
4	XF_TYPE_STRING	string
11	XF_TYPE_OTHER	other – not used directly using XMDF. Have to use HDF5 calls to work with data

C/C++

```
int xfGetPropertyType(xid GroupId, const char *Name, int *Type);
```

FORTRAN

```
SUBROUTINE  XF_GET_PROPERTY_TYPE(GroupId, Name, Type, Error);
INTEGER(XID), INTENT(IN) :: GroupId
CHARACTER(len=*), INTENT(IN) :: Name
INTEGER, INTENT(OUT)     :: Type, Error
```

If the property type is a string, the maximum string length used in the array is needed. The following function gets the maximum length for a string data set.

C/C++

```
int xfGetPropertyStringLength(xid GroupId, const char *Name, int *MaxLength);
```

FORTRAN

```
SUBROUTINE  XF_GET_PROPERTY_STRING_LENGTH(GroupId, Name,
MaxLength, Error);
INTEGER(XID), INTENT(IN) :: GroupId
CHARACTER(len=*), INTENT(IN) :: Name
INTEGER, INTENT(OUT)     :: MaxLength, Error
```

All of the functions to retrieve properties must already be allocated to the correct size. The correct size can be determined using the **xfGetPropertyNumber** function.

C/C++

```
// get the number of items in the attribute
int xfGetPropertyNumber (xid GroupId, const char *Name, int *Size);

int xfReadPropertyInt(xid GroupId, const char *Name, int *Property);
int xfReadPropertyFloat(xid GroupId, const char *Name, float *Property);
int xfReadPropertyDouble(xid GroupId, const char *Name, double *Property);
int xfReadPropertyString(xid GroupId, const char *Name, char **Property,
                         int StringLength);
```

FORTRAN

```
SUBROUTINE  XF_GET_PROPERTY_NUMBER(GroupId, Name, Size, Error);
INTEGER(XID), INTENT(IN) :: GroupId
CHARACTER(len=*), INTENT(IN) :: Name
INTEGER, INTENT(OUT)     :: Size, Error

SUBROUTINE  XF_READ_PROPERTY_INT(GroupId, Name, Property, Error);
INTEGER(XID), INTENT(IN) :: GroupId
CHARACTER(len=*), INTENT(IN) :: Name
INTEGER, INTENT(OUT)     :: Property(*), Error

SUBROUTINE  XF_READ_PROPERTY_FLOAT(GroupId, Name, Property, Error);
INTEGER(XID), INTENT(IN) :: GroupId
CHARACTER(len=*), INTENT(IN) :: Name
REAL, INTENT(OUT)     :: Property(*)
INTEGER, INTENT(OUT)     :: Error

SUBROUTINE  XF_READ_PROPERTY_DOUBLE(GroupId, Name, Property, Error);
INTEGER(XID), INTENT(IN) :: GroupId
CHARACTER(len=*), INTENT(IN) :: Name
REAL(DOUBLE), INTENT(OUT)     :: Property(*)
INTEGER, INTENT(OUT)     :: Error

SUBROUTINE  XF_READ_PROPERTY_INT(GroupId, Name, Property, Error);
INTEGER(XID), INTENT(IN) :: GroupId
CHARACTER(len=*), INTENT(IN) :: Name
LOGICAL, INTENT(OUT)     :: Property(*)
INTEGER, INTENT(OUT)     :: Error

SUBROUTINE  XF_READ_PROPERTY_STRING(GroupId, Name, Property, Error);
INTEGER(XID), INTENT(IN) :: GroupId
CHARACTER(len=*), INTENT(IN) :: Name
CHARACTER(len=*), INTENT(OUT)     :: Property(*, *)
INTEGER, INTENT(OUT)     :: Error
```

4.9 Meshes

The mesh group must store the information associated with the nodes and elements. Details of the information stored are contained in the following sections.

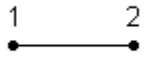
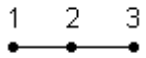
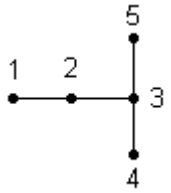
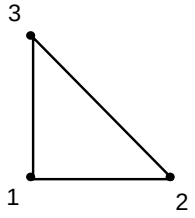
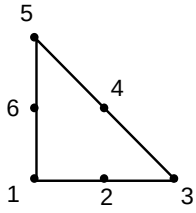
4.9.1 Nodal coordinates

Nodes are defined by an integer element ID and a set of coordinates. For 3-D meshes, each node will have x-, y-, and z-coordinates. For 1-D and 2-D meshes, nodes will always have x- and y-coordinates and in some cases will have z-coordinates as well.

4.9.2 Elements

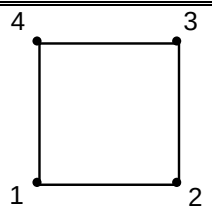
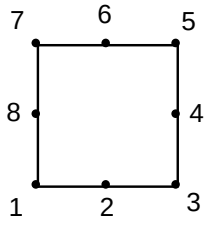
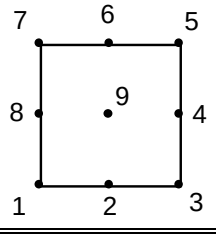
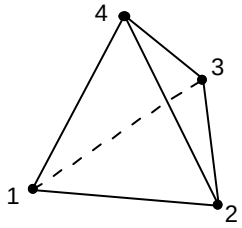
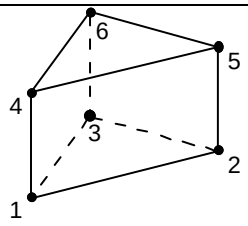
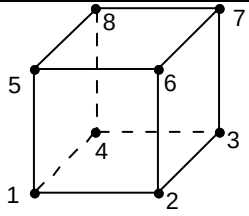
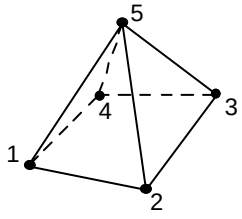
Elements are defined by an integer ID, a type, and the element topology. The topology consists of a list of the nodes that define the element. The element type defines the dimension of the element (1-, 2-, 3-D), the number of nodes, and the shape/structure of the element. The supported element types are shown in Table 4. This list may be extended in the future based on feedback from users.

Table 4
Element Types

Type	Description	# nodes	Topology
003–008	Junction This is a 0D element that joins several 1-D elements. The number of elements is the same as the type. The nodes in a junction element are all at the same position and are attached to other 1-D elements	3–8	Three to eight nodes at the same location connected to 1-D elements.
100	1-D linear	2	
101	1-D quadratic	3	
110	Transition element (1-D-2-D) (RMA2 definition) 1 is connected to a 1-D element 3, 4, 5 are connected to a 2-D element	5	
200	2-D linear triangle	3	
201	2-D quadratic triangle	6	

(Continued)

Table 4 (Concluded)

Type	Description	# nodes	Topology
210	2-D linear quadrilateral	4	
211	2-D quadratic quadrilateral	8	
212	2-D quadratic quadrilateral with center node	9	
300	3-D linear tetrahedron	4	
310	3-D linear prism	6	
320	3-D linear hexahedron	8	
330	3-D linear pyramid	5	

Element activity and material IDs can be stored as properties under the element group. Management of property groups is described in Section 4.8. The reserved names for element activity and material IDs are **Activity** and **Material**.

4.9.3 Coordinate system

The options for storing the coordinate system for a mesh are discussed in Section 4.14.

4.9.4 Group organization

Element group descriptions. The organization of element groups includes the following:

- NumElems – This is a single integer representing the number of elements in the mesh.
- NodeIds – This is a 2-D array of integers that specifies the node indices that make up each element. The node indices are the array indices of the node locations in the nodes group. The array has an attribute, MaxNumnodes, which is the maximum number of nodes in any element. The size of the NodeIds array is NumElems X MaxNumnodes.
- Types – This is the integer ID defining the type of each element as given in Table 4. This may be either an array of size NumElems where each item in the array is the element type for its corresponding element, or a single integer if all the element types are the same.

Node group descriptions. The organization of node groups includes the following:

- NumNodes – This is a single integer representing the number of nodes in the mesh.
- Locations – This is a 2-D array of doubles for the x-, y-, and z-locations of the mesh nodes. The size of the array is 3 X NumNodes.

4.9.5 API Functions

When an API function is called that fills in an array, the array must already be allocated. The path used in all of these functions is the path to the mesh folder, not the path for the element or node folder.

4.9.5.1 Writing

This function is used to set the number of elements for a mesh. Anyone using this API to read the mesh will have to know this in order to properly allocate their arrays.

C/C++

```
int xfSetNumberOfElements(xid GroupId, int NumElems);
```

FORTRAN

```
SUBROUTINE  XF_SET_NUMBER_OF_ELEMENTS(GroupId, NumElems, Error);
INTEGER(XID), INTENT(IN) :: GroupId
INTEGER, INTENT(IN)      :: NumElems
INTEGER, INTENT(OUT)     :: Error
```

Setting the element types. These functions are used to set the element types. If all of the elements have the same type, a single value can be passed rather than an entire array.

C/C++

```
int xfSetAllElemsSameType(xid GroupId, int type);
int xfWriteElemTypes(xid GroupId, int *type, int Compression);
```

FORTRAN

```
SUBROUTINE  XF_SET_ALL_ELEMS_SAME_TYPE(GroupId, Type, Error);
INTEGER(XID), INTENT(IN) :: GroupId
INTEGER, INTENT(IN)      :: Type
INTEGER, INTENT(OUT)     :: Error

SUBROUTINE  XF_WRITE_ELEM_TYPES(GroupId, Types, Error);
INTEGER(XID), INTENT(IN) :: GroupId
INTEGER, INTENT(IN)      :: Types(*)
INTEGER, INTENT(OUT)     :: Error
```

Setting the node IDs for the elements. This function sets the node IDs for the elements (must be numbered consistently with diagrams in Table 4). The “IDs” array must be a contiguous array of size MaxNumNodes X NumElems. The IDs of the nodes start at one, which represents the first array index in the node array. Since C arrays are zero-based, the actual index of the node in the array when using C is decremented by one.

C/C++

```
int xfWriteElemNodeIds(xid GroupId, int MaxNumNodes, int *Ids, int Compression);
```

FORTRAN

```
SUBROUTINE  XF_WRITE_ELEM_NODE_IDS(GroupId, MaxNumNodes, Ids, Compression,
Error);
INTEGER(XID), INTENT(IN) :: GroupId
INTEGER, INTENT(IN)      :: MaxNumNodes, Ids(*), Compression
INTEGER, INTENT(OUT)     :: Error
```

Functions to write the node group data. The functions to write the node group data are as follows:

C/C++

```
int xfSetNumberOfNodes(xid GroupId, int Num);
int xfWriteXNodeLocations(xid GroupId, double *Locs, int Compression);
int xfWriteYNodeLocations(xid GroupId, double *Locs);
int xfWriteZNodeLocations(xid GroupId, double *Locs);
```

FORTRAN

```
SUBROUTINE  XF_SET_NUMBER_OF_NODES(GroupId, Num, Error)
INTEGER(XID), INTENT(IN) :: GroupId
INTEGER, INTENT(IN)      :: Num, Compression
INTEGER, INTENT(OUT)     :: Error

SUBROUTINE  XF_WRITE_X_NODE_LOCATIONS(GroupId, XLocs, Compression, Error)
INTEGER(XID), INTENT(IN) :: GroupId
REAL(DOUBLE), INTENT(IN) :: Locs(*)
INTEGER, INTENT(IN)      :: Compression
INTEGER, INTENT(OUT)     :: Error

SUBROUTINE  XF_WRITE_Y_NODE_LOCATIONS(GroupId, YLocs, Error)
INTEGER(XID), INTENT(IN) :: GroupId
REAL(DOUBLE), INTENT(IN) :: Locs(*)
INTEGER, INTENT(IN)      :: Compression
INTEGER, INTENT(OUT)     :: Error

SUBROUTINE  XF_WRITE_Z_NODE_LOCATIONS(GroupId, ZLocs, Error)
INTEGER(XID), INTENT(IN) :: GroupId
REAL(DOUBLE), INTENT(IN) :: Locs(*)
INTEGER, INTENT(IN)      :: Compression
INTEGER, INTENT(OUT)     :: Error

// The locations can also be written as a single 2D array indexed by
// component then node index
SUBROUTINE  XF_WRITE_LOCATIONS(GroupId, Locs, Compression, Error)
INTEGER(XID), INTENT(IN) :: GroupId
REAL(DOUBLE), INTENT(IN) :: Locs(3,*)
INTEGER, INTENT(IN)      :: Compression
INTEGER, INTENT(OUT)     :: Error
```

4.9.5.2 Reading

Several functions are provided to read the element group data.

The following function is used to get the number of elements in a mesh:

C/C++

```
int xfGetNumberOfElements(xid GroupId, int *NumElems);
```

FORTRAN

```
SUBROUTINE  XF_GET_NUMBER_OF_ELEMENTS(GroupId, NumElems, Error)
INTEGER(XID), INTENT(IN) :: GroupId
INTEGER, INTENT(OUT)      :: NumElems, Error
```

Retrieving the element types. These functions are used to get the element types. If all of the elements have the same type, a single value can be retrieved rather than an entire array.

C/C++

```
int xfAreAllElemsSameType(xid GroupId, xbool &Same);
int xfReadElemTypeSingleValue(xid GroupId, int *Type);
int xfReadElemTypes(xid GroupId, int *Types);
```

FORTRAN

```
SUBROUTINE  XF_ARE_ALL_ELEMS_SAME_TYPE(GroupId, Same, Error)
INTEGER(XID), INTENT(IN) :: GroupId
LOGICAL, INTENT(OUT)     :: Same
INTEGER, INTENT(OUT)      :: Error

SUBROUTINE  XF_READ_ELEM_TYPE_SINGLE_VALUE(GroupId, Type, Error)
INTEGER(XID), INTENT(IN) :: GroupId
INTEGER, INTENT(OUT)     :: Type
INTEGER, INTENT(OUT)      :: Error

SUBROUTINE  XF_READ_ELEM_TYPES(GroupId, Types, Error)
INTEGER(XID), INTENT(IN) :: GroupId
INTEGER, INTENT(OUT)     :: Types(*)
INTEGER, INTENT(OUT)      :: Error
```

Retrieving element connectivity information. These functions are used to get the element connectivity information. The first returns the maximum number of nodes in any element in the mesh. This is used to dimension the connectivity array returned by the second function.

C/C++

```
int xfGetMaxNodesInElem(xid GroupId, int *MaxNodes);
int xfReadElemNodeIds(xid GroupId, int *ElemNodes);
```

FORTRAN

```
SUBROUTINE  XF_GET_MAX_NODES_IN_ELEM(GroupId, MaxNodes, Error)
INTEGER(XID), INTENT(IN) :: GroupId
INTEGER, INTENT(OUT)     :: MaxNodes
INTEGER, INTENT(OUT)      :: Error

SUBROUTINE  XF_GET_MAX_NODES_IN_ELEM(GroupId, ElemNodes, Error)
INTEGER(XID), INTENT(IN) :: GroupId
INTEGER, INTENT(OUT)     :: ElemNodes(*)
INTEGER, INTENT(OUT)      :: Error
```

Reading the node group data. The following functions are used to read the node group data:

C/C++

```
int xfGetNumberOfNodes(xid GroupId, int *NumNodes);
int xfReadNodeLocationsX(xid GroupId, double *xLocs);
int xfReadNodeLocationsY(xid GroupId, double *yLocs);
int xfReadNodeLocationsZ(xid GroupId, double *zLocs);
```

FORTRAN

```
SUBROUTINE  XF_GET_NUMBER_OF_NODES(GroupId, NumNodes, Error)
INTEGER(XID), INTENT(IN) :: GroupId
INTEGER, INTENT(OUT)     :: NumNodes
INTEGER, INTENT(OUT)     :: Error

SUBROUTINE  XF_READ_NODE_LOCATIONS_X(GroupId, xLocs, Error)
INTEGER(XID), INTENT(IN) :: GroupId
REAL(DOUBLE), INTENT(OUT)     :: xLocs(*)
INTEGER, INTENT(OUT)     :: Error

SUBROUTINE  XF_READ_NODE_LOCATIONS_Y(GroupId, yLocs, Error)
INTEGER(XID), INTENT(IN) :: GroupId
REAL(DOUBLE), INTENT(OUT)     :: yLocs(*)
INTEGER, INTENT(OUT)     :: Error

SUBROUTINE  XF_READ_NODE_LOCATIONS_Y(GroupId, zLocs, Error)
INTEGER(XID), INTENT(IN) :: GroupId
REAL(DOUBLE), INTENT(OUT)     :: zLocs(*)
INTEGER, INTENT(OUT)     :: Error

// To read all the same time similar to writing.
SUBROUTINE XF_READ_NODE_LOCATIONS (GroupId, Locs, Error)
INTEGER(XID), INTENT(IN) :: GroupId
REAL(DOUBLE), INTENT(OUT)     :: Locs(3, *)
INTEGER, INTENT(OUT)     :: Error
```

4.9.6 Properties

Properties can be specified for the mesh, the elements, or the nodes. These properties are created using the methodology described in Section 4.8. The properties group must be opened to read or write property data. The following functions are used to open the property groups associated with a mesh. When it is no longer needed, the property group must be closed using xfCloseGroup.

C/C++

```
int xfGetMeshPropertyGroup(xid MeshGroupId, xid *PropGroupId);
int xfGetMeshElementPropertyGroup(xid MeshGroupId, xid
*PropGroupId);
int xfGetMeshNodePropertyGroup(xid MeshGroupId, xid *PropGroupId);
```

FORTRAN

```
SUBROUTINE  XF_GET_MESH_PROPERY_GROUP(MeshGroupId, PropGroupId, Error)
INTEGER(XID), INTENT(IN) :: MeshGroupId
INTEGER(XID), INTENT(OUT) :: PropGroupId
INTEGER, INTENT(OUT)     :: Error
```

```
SUBROUTINE  XF_GET_MESH_ELEMENT_PROPERY_GROUP(MeshGroupId, PropGroupId, Error)
INTEGER(XID), INTENT(IN) :: MeshGroupId
INTEGER(XID), INTENT(OUT) :: PropGroupId
INTEGER, INTENT(OUT)     :: Error

SUBROUTINE  XF_GET_MESH_NODE_PROPERY_GROUP(MeshGroupId, PropGroupId, Error)
INTEGER(XID), INTENT(IN) :: MeshGroupId
INTEGER(XID), INTENT(OUT) :: PropGroupId
INTEGER, INTENT(OUT)     :: Error
```

4.10 Grids

The grid group stores information associated with 2- or 3-D grids. This information includes the geometric definition of the grid along with grid properties and data sets associated with the grid.

4.10.1 Grid properties

Several properties, including the ones listed in Table 5, are associated with a grid.

Table 5
Grid Properties

Item	Description	Required?
Dimension	2-D versus 3-D	Yes
Grid type	Cartesian, curvilinear, extruded Cartesian, extruded curvilinear	Yes
Extrusion type	Type of extrusion used for K direction. Options include sigma stretch, Cartesian, curvilinear at corners, curvilinear at midsides. Must be defined for extruded grids.	No
Global coordinate system	Used to position the grid in 3-D space.	No
Origin	The coordinates of the grid origin (I=0, J=0, K=0) in the global coordinate system	No
Orientation	Right-hand or left-hand rule	Yes
Dip	The angle of rotation about the global x-axis (this is 0.0 for plan view 2-D cases and 90 for vertically averaged 2-D cases)	No
Bearing	The angle of rotation about the global z-axis	No
Computational Origin	The geometric corner of the grid that is the computational origin. By default this is the geometric origin (location 1).	No
U Direction	The direction of the u axis that defines the position of the 3-D grid triad on the geometric definition of the grid. This value defaults to either 1 or –1 depending on the computational origin.	No (applies only to 3-D case)
NumI	The number of cells in the I direction	Yes
NumJ	The number of cells in the J direction	Yes
NumK	The number of cells in the K direction (layers)	Yes (3-D Grids only)

4.10.1.1 Cartesian grids

For Cartesian grids the local coordinate location of each row, column, and layer (if 3-D) boundary is specified. The first boundary in each direction is the local origin. Figure 7 shows a 2-D Cartesian grid with the required grid cell boundary locations. The origin is always (0.0, 0.0) and therefore is not specified.

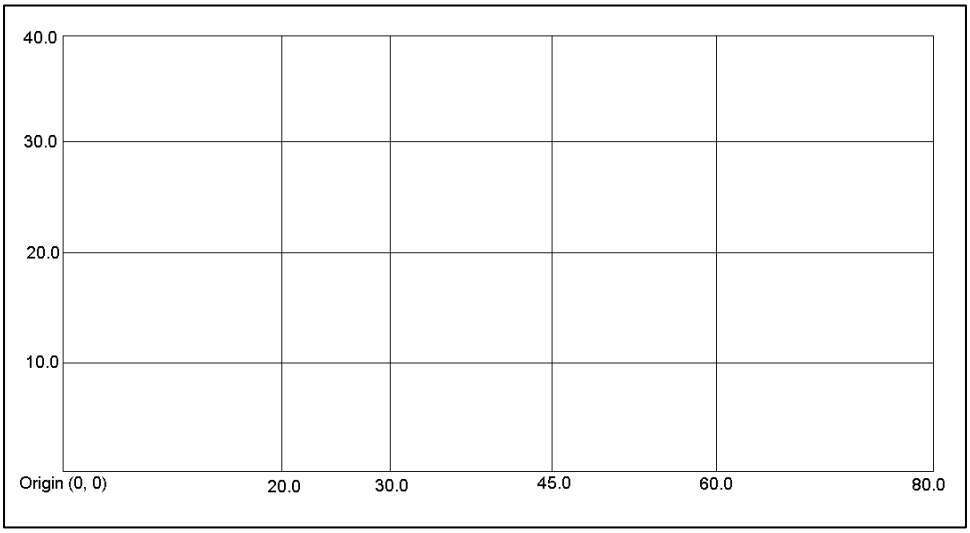


Figure 7. 2-D Cartesian grid with boundary locations

4.10.1.2 Curvilinear grids

Curvilinear grids must have all coordinates defined for each grid corner. For a 2-D grid, there are (NumI + 1) * (NumJ + 1) corners. A 3-D grid has (NumI + 1) * (NumJ + 1) * (NumK + 1) corners. Figure 8 shows a 2-D curvilinear grid.

4.10.1.3 Extruded grids

Some numerical models use grids that are 2-D grids extruded in the K direction to form 3-D grids. Both 2-D curvilinear and 2-D Cartesian grids can be extruded. The methods to extrude 2-D grids include sigma-stretch, Cartesian, curvilinear at corners, and curvilinear at midsides. Sigma-stretch grids have top and bottom K values (elevations) that may vary from column to column. Each layer of a sigma-stretch grid is a constant percentage of the K thickness. Cartesian extruded grids define constant K values for each layer in the grid (this is used only for 2-D curvilinear grids because otherwise it would just be a 3-D Cartesian grid). Curvilinear extrusion grids define the top and bottom K values for each layer and each corner or cell.

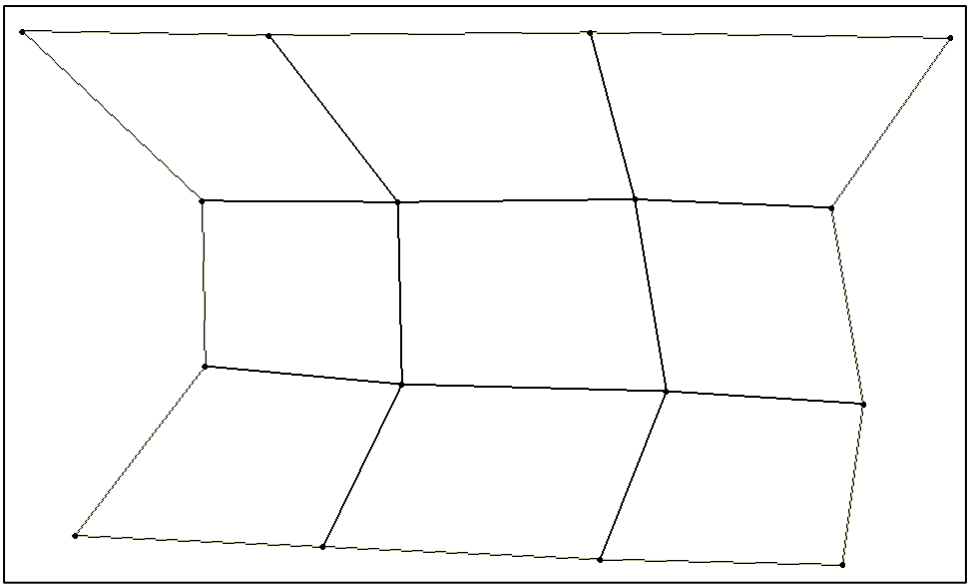


Figure 8. 2-D curvilinear grid

The CH3D hydrodynamic model supports 2-D curvilinear grids extruded either by sigma-stretch or Cartesian. The groundwater model MODFLOW uses a 2-D Cartesian grid extruded curvilinear at cell centers.

4.10.2 API functions

4.10.2.1 Defining the type of grid

The following function is used to define the type of grid. Grid type is one of the following constants:

- GRID_TYPE_CARTESIAN
- GRID_TYPE_CURVILINEAR
- GRID_TYPE_CARTESIAN_EXTRUDED
- GRID_TYPE_CURVILINEAR_EXTRUDED

C/C++

```
int xfSetGridType(xid GroupId, int GridType);
```

FORTRAN

```
SUBROUTINE xfSetGridType(GroupId, GridType, Error)
INTEGER(XID), INTENT(IN) :: GroupId
INTEGER, INTENT(IN) :: GridType
INTEGER, INTENT(OUT)     :: Error
```

The following function is used only for extruded grids to define the type of extrusion taking place. Extrudetype must be one of the following constants:

- EXTRUDE_SIGMA
- EXTRUDE_CARTESIAN
- EXTRUDE_CURV_AT_CORNERS
- EXTRUDE_CURV_AT_CELLS

C/C++

```
int xfSetExtrusionType(xid GroupId, int ExtrudeType);
```

FORTRAN

```
SUBROUTINE XF_SET_EXTRUSION_TYPE(GroupId, ExtrudeType, Error)
INTEGER(XID), INTENT(IN) :: GroupId
INTEGER, INTENT(IN) :: ExtrudeType
INTEGER, INTENT(OUT)     :: Error
```

4.10.2.2 Setting grid attributes

These functions are used to set attributes for a grid. The number of dimensions must be 2 or 3. Orientation must be one of the following constants: ORIENTATION_RIGHT_HAND, ORIENTATION_LEFT_HAND. The computational origin is a number between 1 and 4 for 2-D grids and between 1 and 8 for 3-D grids. The number corresponds to the corner of the grid that will be the origin. Figure 9 shows the corresponding values. The *u* direction must be 1, 2, 3, –1, –2, or –3. This corresponds to the grid direction as labeled in Figure 9. Negative direction values indicate the origin is at the other side of the grid. (For more information on the grid computational origin, see Appendix B, Grid.)

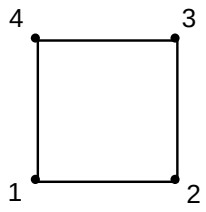
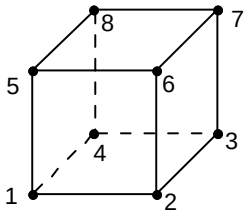
Two-Dimensional	Three-Dimensional
	
Grid direction 1 – location 1 to location 2 Grid direction 2 – location 1 to location 4	Grid direction 1 – location 1 to location 2 Grid direction 2 – location 1 to location 4 Grid direction 3 – location 1 to location 5

Figure 9. Setting attributes for grid

C/C++

```
int xfSetNumberOfDimensions(xid GroupId, int NumDimensions);
int xfSetOrigin(xid GroupId, double x, double y, double z);
int xfSetOrientation(xid GroupId, int Orientation);
int xfSetBearing(xid GroupId, double Bearing);
int xfSetDip(xid GroupId, double Dip);
int xfSetComputationalOrigin(xid GroupId, int Origin);
int xfSetUDirection(xid GroupId, int Direction);
int xfSetNumberCellsInI(xid GroupId, int NumI);
int xfSetNumberCellsInJ(xid GroupId, int NumJ);
int xfSetNumberCellsInK(xid GroupId, int NumK);
```

FORTRAN

```
SUBROUTINE XF_SET_NUMBER_OF_DIMENSIONS(GroupId, NumDimensions, Error)
INTEGER(XID), INTENT(IN) :: GroupId
INTEGER, INTENT(IN) :: NumDimensions
INTEGER, INTENT(OUT)     :: Error

SUBROUTINE  XF_SET_ORIGIN(GroupId, x, y, z, Error)
INTEGER(XID), INTENT(IN) :: GroupId
REAL(DOUBLE), INTENT(IN) :: x, y, z
INTEGER, INTENT(OUT)     :: Error

SUBROUTINE  XF_SET_ORIENTATION(GroupId, Orientation, Error)
INTEGER(XID), INTENT(IN) :: GroupId
INTEGER, INTENT(IN) :: Orientation
INTEGER, INTENT(OUT)     :: Error

SUBROUTINE  XF_SET_BEARING(GroupId, Bearing, Error)
INTEGER(XID), INTENT(IN) :: GroupId
REAL(DOUBLE), INTENT(IN) :: Bearing
INTEGER, INTENT(OUT)     :: Error

SUBROUTINE  XF_SET_DIP(GroupId, Dip, Error)
INTEGER(XID), INTENT(IN) :: GroupId
REAL(DOUBLE), INTENT(IN) :: Dip
INTEGER, INTENT(OUT)     :: Error

SUBROUTINE XF_SET_COMPUTATIONAL_ORIGIN (GroupId, Origin, Error)
INTEGER(XID), INTENT(IN) :: GroupId
INTEGER, INTENT(IN)      :: Origin
INTEGER, INTENT(OUT)     :: Error

SUBROUTINE XF_SET_U_DIRECTION (GroupId, Direction, Error)
INTEGER(XID), INTENT(IN) :: GroupId
INTEGER, INTENT(IN)      :: Direction
INTEGER, INTENT(OUT)     :: Error

SUBROUTINE  XF_SET_NUM_CELLS_I(GroupId, NumCellsI, Error)
INTEGER(XID), INTENT(IN) :: GroupId
REAL(DOUBLE), INTENT(IN) :: NumCellsI
INTEGER, INTENT(OUT)     :: Error

SUBROUTINE  XF_SET_NUM_CELLS_J(GroupId, NumCellsJ, Error)
INTEGER(XID), INTENT(IN) :: GroupId
REAL(DOUBLE), INTENT(IN) :: NumCellsJ
INTEGER, INTENT(OUT)     :: Error

SUBROUTINE  XF_SET_NUM_CELLS_K(GroupId, NumCellsK, Error)
INTEGER(XID), INTENT(IN) :: GroupId
REAL(DOUBLE), INTENT(IN) :: NumCellsK
INTEGER, INTENT(OUT)     :: Error
```

4.10.2.3 Retrieving information about a grid

The following functions are used to retrieve information about a grid. Optional parameters have a function that can be used to determine whether the parameter was set.

C/C++

```
int xfGetGridType(xid GroupId, int *GridType);
int xfGetExtrusionType(xid GroupId, int *ExtrudeType);
int xfGetNumberOfDimensions(xid GroupId, int *NumDimensions);
int xfOriginDefined(xid GroupId, xbool *bDefined);
int xfGetOrigin(xid GroupId, double *x, double *y, double *z);
int xfGetOrientation(xid GroupId, int *Orientation);
int xfBearingDefined(xid GroupId, xbool *bDefined);
int xfGetBearing(xid GroupId, double *bearing);
int xfDipDefined(xid GroupId, xbool *bDefined);
int xfGetDip(xid GroupId, double *dip);
int xfComputationalOriginDefined(xid GroupId, xbool *bDefined);
int xfGetComputationalOrigin (xid GroupId, int *Origin);
int xfGetUDirectionDefined(xid GroupId, xbool *bDefined);
int xfGetUDirection (xid GroupId, double *Direction);
int xfGetNumberCellsInI(xid GroupId, int *NumI);
int xfGetNumberCellsInJ(xid GroupId, int *NumJ);
int xfGetNumberCellsInK(xid GroupId, int *NumK);
```

FORTRAN

```
SUBROUTINE  XF_GET_GRID_TYPE(GroupId, GridType, Error)
INTEGER(XID), INTENT(IN) :: GroupId
INTEGER, INTENT(OUT)     :: GridType
INTEGER, INTENT(OUT)     :: Error

SUBROUTINE  XF_GET_EXTRUSION_TYPE(GroupId, ExtrudeType, Error)
INTEGER(XID), INTENT(IN) :: GroupId
INTEGER, INTENT(OUT)     :: ExtrudeType
INTEGER, INTENT(OUT)     :: Error

SUBROUTINE  XF_GET_NUMBER_OF_DIMENSIONS(GroupId, NumDimensions, Error)
INTEGER(XID), INTENT(IN) :: GroupId
INTEGER, INTENT(OUT)     :: NumDimensions
INTEGER, INTENT(OUT)     :: Error

SUBROUTINE  XF_ORIGIN_DEFINED(GroupId, Defined, Error)
INTEGER(XID), INTENT(IN) :: GroupId
LOGICAL, INTENT(OUT)     :: Defined
INTEGER, INTENT(OUT)     :: Error

SUBROUTINE  XF_GET_ORIGIN(GroupId, x, y, z, Error)
INTEGER(XID), INTENT(IN)  :: GroupId
REAL(DOUBLE), INTENT(OUT) :: x, y, z
INTEGER, INTENT(OUT)      :: Error

SUBROUTINE  XF_GET_ORIENTATION(GroupId, Orientation, Error)
INTEGER(XID), INTENT(IN) :: GroupId
INTEGER, INTENT(OUT)     :: Orientation
INTEGER, INTENT(OUT)     :: Error

SUBROUTINE  XF_BEARING_DEFINED(GroupId, Defined, Error)
INTEGER(XID), INTENT(IN) :: GroupId
LOGICAL, INTENT(OUT)     :: Defined
INTEGER, INTENT(OUT)     :: Error
```

```
SUBROUTINE  XF_GET_BEARING(GroupId, Bearing, Error)
INTEGER(XID), INTENT(IN)  :: GroupId
REAL(DOUBLE), INTENT(OUT) :: Bearing
INTEGER, INTENT(OUT)      :: Error

SUBROUTINE  XF_DIP_DEFINED(GroupId, Defined, Error)
INTEGER(XID), INTENT(IN) :: GroupId
LOGICAL, INTENT(OUT)     :: Defined
INTEGER, INTENT(OUT)     :: Error

SUBROUTINE  XF_GET_DIP(GroupId, Dip, Error)
INTEGER(XID), INTENT(IN)  :: GroupId
REAL(DOUBLE), INTENT(OUT) :: Dip
INTEGER, INTENT(OUT)      :: Error

SUBROUTINE XF_COMPUTATIONAL_ORIGIN_DEFINED (GroupId, Defined, Error)
INTEGER(XID), INTENT(IN) :: GroupId
LOGICAL, INTENT(OUT)     :: Defined
INTEGER, INTENT(OUT)     :: Error

SUBROUTINE XF_GET_COMPUTATIONAL_ORIGIN (GroupId, Origin, Error)
INTEGER(XID), INTENT(IN) :: GroupId
INTEGER, INTENT(OUT)     :: Origin
INTEGER, INTENT(OUT)     :: Error

SUBROUTINE XF_U_DIRECTION_DEFINED (GroupId, Defined, Error)
INTEGER(XID), INTENT(IN) :: GroupId
LOGICAL, INTENT(OUT)     :: Defined
INTEGER, INTENT(OUT)     :: Error

SUBROUTINE XF_GET_U_DIRECTION (GroupId, Direction, Error)
INTEGER(XID), INTENT(IN) :: GroupId
INTEGER, INTENT(OUT)     :: Direction
INTEGER, INTENT(OUT)     :: Error

SUBROUTINE  XF_GET_NUMBER_CELLS_IN_I(GroupId, NumI, Error)
INTEGER(XID), INTENT(IN) :: GroupId
INTEGER, INTENT(OUT) :: NumI
INTEGER, INTENT(OUT)     :: Error

SUBROUTINE  XF_GET_NUMBER_CELLS_IN_J(GroupId, NumJ, Error)
INTEGER(XID), INTENT(IN) :: GroupId
INTEGER, INTENT(OUT) :: NumJ
INTEGER, INTENT(OUT)     :: Error

SUBROUTINE  XF_GET_NUMBER_CELLS_IN_K(GroupId, NumK, Error)
INTEGER(XID), INTENT(IN) :: GroupId
INTEGER, INTENT(OUT) :: NumK
INTEGER, INTENT(OUT)     :: Error
```

4.10.3 Grid geometry

The next three functions are used to specify the grid geometry. For Cartesian grids, the grid geometry is the locations of cell boundaries for each row, column, and layer. The first row, column, and layer boundary is always assumed to be the grid origin. When using these functions to set the grid geometry, **NumVals** is equal to the number of cells in the corresponding direction.

For curvilinear grids, each coordinate of the grid must be specified for each corner of every cell in the grid. For a 2-D curvilinear grid **NumVals** is equal to (NumI + 1) * (NumJ + 1) for both **xfSetGridCoordsI** and **xfSetGridCoordsJ**.

For a 3-D curvilinear grid **NumVals** is equal to (NumI + 1) * (NumJ + 1) * (NumK + 1) for all of these functions. The arrays are numbered in I, J, K order.

C/C++

```
int xfSetGridCoordsI(xid GroupId, int NumVals, double *iValues);
int xfSetGridCoordsJ(xid GroupId, int NumVals, double *jValues);
int xfSetGridCoordsK(xid GroupId, int NumVals, double *kValues);
```

FORTRAN

```
SUBROUTINE  XF_SET_GRID_COORDS_I(GroupId, NumVals, iValues, Error)
INTEGER(XID), INTENT(IN) :: GroupId
INTEGER, INTENT(IN) :: NumVals
REAL(DOUBLE), INTENT(IN) :: iValues
INTEGER, INTENT(OUT)     :: Error

SUBROUTINE  XF_SET_GRID_COORDS_J(GroupId, NumVals, jValues, Error)
INTEGER(XID), INTENT(IN) :: GroupId
INTEGER, INTENT(IN) :: NumVals
REAL(DOUBLE), INTENT(IN)  :: jValues
INTEGER, INTENT(OUT)     :: Error

SUBROUTINE  XF_SET_GRID_COORDS_K(GroupId, NumVals, kValues, Error)
INTEGER(XID), INTENT(IN) :: GroupId
INTEGER, INTENT(IN) :: NumVals
REAL(DOUBLE), INTENT(IN)  :: kValues
INTEGER, INTENT(OUT)     :: Error
```

4.10.4 Grid coordinate values

The following functions are used to read the grid coordinate values from the file. The number of values is passed specifying the size allocated for the arrays. If the size is incorrect, the library will return a negative value for the error.

C/C++

```
int xfGetGridCoordsI(xid GroupId, int NumVals, double *iValues);
int xfGetGridCoordsJ(xid GroupId, int NumVals, double *jValues);
int xfGetGridCoordsK(xid GroupId, int NumVals, double *kValues);
```

FORTRAN

```
SUBROUTINE  XF_GET_GRID_COORDS_I(GroupId, NumVals, iValues, Error)
INTEGER(XID), INTENT(IN) :: GroupId
INTEGER, INTENT(IN) :: NumVals
REAL(DOUBLE), INTENT(OUT) :: iValues
INTEGER, INTENT(OUT)     :: Error

SUBROUTINE  XF_GET_GRID_COORDS_J(GroupId, NumVals, jValues, Error)
INTEGER(XID), INTENT(IN) :: GroupId
INTEGER, INTENT(IN) :: NumVals
REAL(DOUBLE), INTENT(OUT)  :: jValues
INTEGER, INTENT(OUT)     :: Error
```

```
SUBROUTINE  XF_GET_GRID_COORDS_K(GroupId, NumVals, kValues, Error)
INTEGER(XID), INTENT(IN) :: GroupId
INTEGER, INTENT(IN) :: NumVals
REAL(DOUBLE), INTENT(OUT)  :: kValues
INTEGER, INTENT(OUT)     :: Error
```

4.10.5 Extruded layers

Using extrusion requires information about how the layers are defined. The following function is used to define the layer information for extruded grids. **NumVals** and **Values** have a different meaning depending upon which extrusion option the grid is using. For this function the top-layer data are passed in first down to the bottom layer of the grid. When using sigma-stretch extrusion the variable **Values** is the percent thickness of each layer of the grid. **NumVals** should correspond to **NumLayers** for sigma-stretch grids. When using Cartesian extrusion the variable **Values** represents the cell face locations perpendicular to the K direction. **NumVals** will be the NumLayers + 1 because each layer has a top and bottom face. When using curvilinear extrusion at corners, the variable **Values** is the K location of every corner of the grid. **NumVals** in this case is (NumI + 1) * (NumJ + 1) * (NumLayers + 1). The values in the array loop on I first, J second, and layers (top–bottom) last. When using curvilinear extrusion at cells the variable **Values** is the K location of every cell face perpendicular to the K direction. **NumVals** in this case is NumI * NumJ * NumLayers. The values in the array loop on I first, J second, and layers (top–bottom) last.

C/C++

```
int xfWriteExtrudeLayerData(xid GroupId, int NumLayers, int NumVals, double
*Values);
```

FORTRAN

```
SUBROUTINE  XF_WRITE_EXTRUDE_LAYER_DATA(GroupId, NumLayers, NumVals, Values,
Error)
INTEGER(XID), INTENT(IN) :: GroupId
INTEGER, INTENT(IN)      :: NumVals
REAL(DOUBLE), INTENT(IN) :: iValues
INTEGER, INTENT(OUT)     :: Error
```

The following functions are used to read the extrude information from the file. The number of values is passed in so that the library can check to see whether the array was allocated to the correct size.

C/C++

```
int xfGetExtrudeNumLayers(xid GroupId, int *NumLayers);
int xfGetExtrudeValues(xid GroupId, int NumVals, *Values);
```

FORTRAN

```
SUBROUTINE  XF_GET_EXTRUDE_NUM_LAYERS(GroupId, NumLayers, Error)
INTEGER(XID), INTENT(IN) :: GroupId
INTEGER, INTENT(OUT)     :: NumLayers
INTEGER, INTENT(OUT)     :: Error

SUBROUTINE  XF_GET_EXTRUDE_VALUES(GroupId, NumValues, Values, Error)
INTEGER(XID), INTENT(IN) :: GroupId
INTEGER, INTENT(IN)     :: NumValues
REAL(DOUBLE), INTENT(OUT) :: Values
INTEGER, INTENT(OUT)     :: Error
```

4.10.6 Cell and node properties

Properties can be specified for the grid, the cells, or the corners (mesh values). These properties are created using the methodology described in Section 4.8. The following functions are used to open the property groups associated with a grid. After finishing, the group IDs must be closed using the function **xfCloseGroup**.

C/C++

```
int xfGetGridPropertyGroup(xid GridGroupId, xid *PropGroupId);
int xfGetGridCellPropertyGroup(xid GridGroupId, xid *PropGroupId);
int xfGetGridNodePropertyGroup(xid GridGroupId, xid *PropGroupId);
```

FORTRAN

```
SUBROUTINE  XF_GET_GRID_PROPERTY_GROUP(GridGroupId, PropGroupId, Error)
INTEGER(XID), INTENT(IN) :: GroupId
INTEGER(XID), INTENT(OUT):: PropGroupId
INTEGER, INTENT(OUT)     :: Error

SUBROUTINE  XF_GET_GRID_CELL_PROPERTY_GROUP(GridGroupId, PropGroupId, Error)
INTEGER(XID), INTENT(IN) :: GroupId
INTEGER(XID), INTENT(OUT):: PropGroupId
INTEGER, INTENT(OUT)     :: Error

SUBROUTINE  XF_GET_GRID_NODE_PROPERTY_GROUP(GridGroupId, PropGroupId, Error)
INTEGER(XID), INTENT(IN) :: GroupId
INTEGER(XID), INTENT(OUT):: PropGroupId
INTEGER, INTENT(OUT)     :: Error
```

4.10.6.1 Activity flags

One of the reserved property names is *activity*. This is an array of on/off values that must be the same size as the number of cells in the grid. The activity array indicates whether every cell is on or off (included in the grid computations). If no activity flags are specified, it will be assumed that all cells are on. Activity can also be defined on a data set level for cells that are computationally active but are inactive at specific times. This occurs in hydraulic studies where grid cells are allowed to go dry (inactive).

4.10.6.2 Null values

Curvilinear grids may have large sections of inactive space. This is especially true in riverine applications. In these cases it is often easier to build the grid without including data in the empty spaces. Sometimes elevation data are unavailable for areas outside the active portions of the grid. Empty areas can be filled in with a null value rather than cell boundary locations. When a null value is used, it is stored as a property of the grid. Figure 10 shows a situation where a null value may be useful. The grid shown is of a river that branches. For this problem the null value would be used above and below the right branch of the river (everything greater than 3 in the I-direction and not between 2 and 5 in the J-direction).

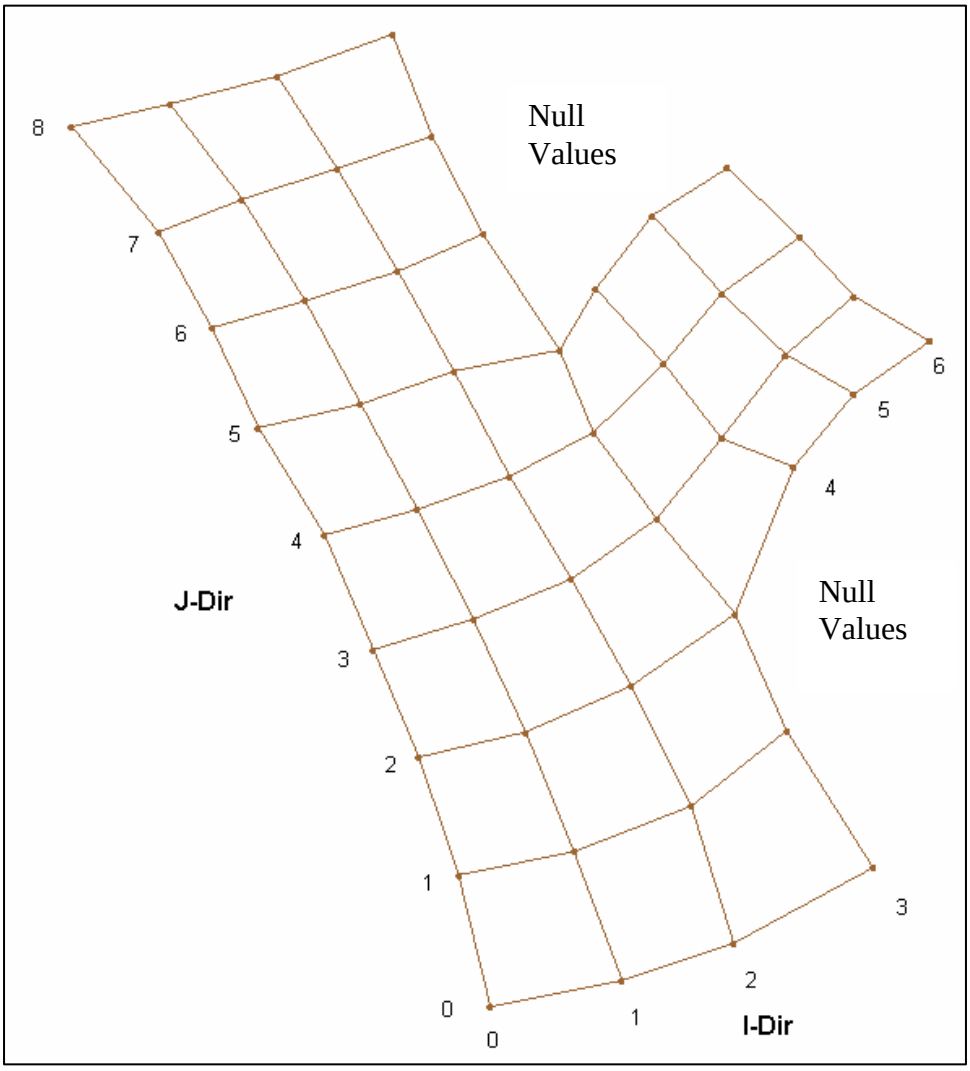


Figure 10. 2-D curvilinear grid that branches

4.11 Cross-section Data

NOTE: CROSS-SECTION DATA ARE NOT SUPPORTED IN VERSION 1.00 OF XMDF

Cross sections will consist of five different groups of data (cross sections, profiles, point properties, line properties, and cross-section geometry), which are associated with one of two different kinds of geometric objects (a point or line).

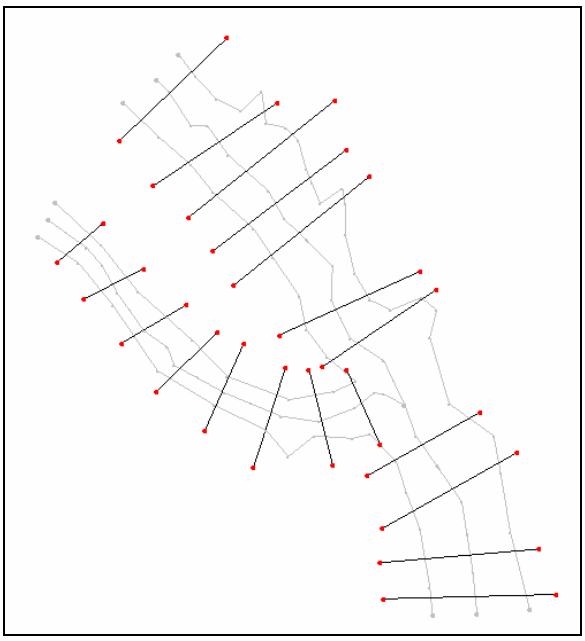


Figure 11. Cross sections

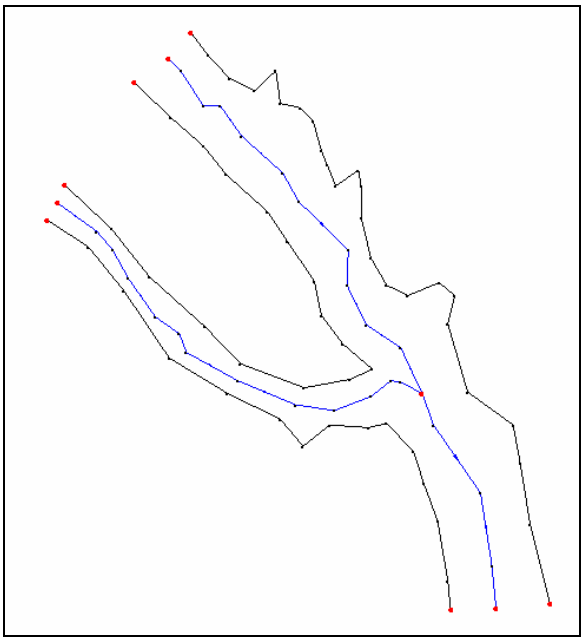


Figure 12. Profile lines for center line and banks

4.11.1 Cross sections

A cross-section group (Figure 11) includes a cross-section entity defined by the river/reach combination in which it is associated. Cross sections are also associated with profile lines by these names. They reference the cross-section geometry (D, x-, y-, and z-values). Cross-section properties include the type of spatial reference (point, 2-D line, or 3-D line) and the spatial referencing entity. Because a cross section may have many point or line properties but each point or line property has only one cross section, they are not referenced directly by the cross sections. Instead, each point and line property references the cross section.

The cross-section geometry is a 3-D array (number of cross sections, by maximum number of d-z pairs, by 2) that hold the d-z pairs for each cross section. The geometry of a cross section defined by a 3-D line could be stored in the cross-section geometry, or it could be left out with the calling application generating the geometry “on the fly,” as needed.

4.11.2 Profiles

Profile lines (Figure 12) define profiles such as center lines or bank lines and can be either 2-D or 3-D. If they are defined as 3-D, then the z-values that are a part of the geometry can be used to derive the bathymetric cross-section geometry. Profile lines are identified by

a river and reach name and can be associated with cross sections based on these names.

4.11.3 Point properties

Point properties represent important locations along the cross-section geometry. They may include such things as the thalweg or left or right banks. Point properties are stored in a continuous list and reference the cross section (by ID) with which they are associated.

Points are defined by their IDs and x-, y-, z-coordinate pairs. Points will also contain attributes according to the type of point being represented (i.e., node, vertex, hydraulic structure, etc.). A point will be referenced for each cross section whose spatial origin definition is given as Point.

4.11.4 Line properties

Line properties represent important lengths along the cross section that can be associated with material properties such as Manning's roughness value. Line properties are stored in a continuous list and reference the cross section with which they are associated by the cross-section ID.

Lines are defined by their IDs and a variable-length list of point IDs. Lines will also contain attributes according to the type of line being represented (2-D cross section, 3-D cross section, profile bank, etc.). A line will be referenced for each cross section whose spatial origin definition is given as either 2- or 3-D, and by all profiles.

4.11.5 Group organization

The XSECS group will store the cross section and profile information. Also shown here are the point and line geometry, even though they will eventually be a part of the map module group. The XSECS group layout is shown in Figure 13. It will include five subgroups, one for cross sections, one for profiles, one for point properties, one for line properties, and one for the cross-section geometry (the point and line subgroups are shown here but will eventually be a part of the map module).

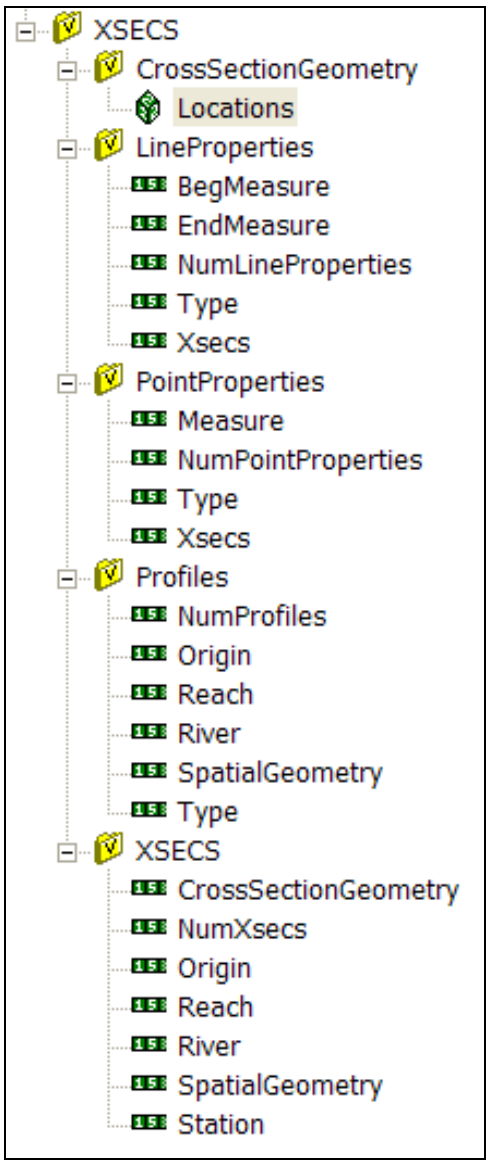


Figure 13. XSECS group layout

4.11.5.1 XSECS subgroup descriptions

The XSECS subgroup contains the following information:

- NumXsecs – This is a single integer representing the number of cross sections.
- River – This is a 1-D array of text strings that store the river name the cross section is associated with. This array is dimensioned by **NumXsecs**.
- Reach – This is a 1-D array of text strings that store the reach name (this would be a subriver name) the cross section is associated with. This array is dimensioned by **NumXsecs**.
- Station – This is a 1-D array of floating point numbers that hold linear stations of the cross section along the defined River and Reach.
- Origin – This is a 1-D array of integer flags with the following possible meanings: 1 – Point, 2 – 2-D Line, and 3 – 3-D Line. This array is dimensioned by **NumXsecs**.
- CrossSectionGeometry – This is a 1-D integer array of IDs that reference the Xsecgeometry subgroup. This array is dimensioned by **NumXsecs**.
- SpatialGeometry – This is a 1-D array of IDs that reference the point or line to which the cross section is tied for spatial referencing. If the Origin flag is defined as Point, then the ID references into the point geometry subgroup; and if it is defined as either a 2-D line or a 3-D line, then it references into the line geometry subgroup. If the origin type is 2-D, then the cross-section geometry must defined; but if the origin is 3-D, then the cross-section geometry can be defined or left blank (the CrossSection geometry in this case would be NULL rather than some ID). This array is dimensioned by **NumXsecs**.

4.11.5.2 Profiles subgroup descriptions

The Profiles subgroup contains the following information:

- NumProfiles – This is a single integer representing the number of profiles.
- River – This is a 1-D array of text strings that store the river name the profile line is associated with. This array is dimensioned by **NumProfiles**.
- Reach – This is a 1-D array of text strings that store the reach name (this would be a subriver name) the profile is associated with. This array is dimensioned by **NumProfiles**.
- Type – This is a 1-D array of integer flags with the following possible meanings: 1 – Center Line, 2 – Left Bank, 3 – Right Bank, 4 – Other. This array is dimensioned by **NumProfiles**.

- Origin – This is a 1-D array of integer flags with the following possible meanings: 2 – 2-D Line, and 3 – 3-D Line. This array is dimensioned by **NumProfiles**.
- SpatialGeometry - This is a 1-D array of IDs that reference the line to which the profile is defined. This array is dimensioned by **NumProfiles**.

4.11.5.3 Point Properties subgroup descriptions

The Point Properties subgroup contains the following information:

- NumPointProperties – This is a single integer representing the number of point properties.
- Xsecs – This is an integer array that references the cross section to which the point property belongs.
- Type – This is an integer array of flags that represent the type of point property. Initially the values will be 1 – thalweg, 2 – left bank, 3 – right bank.
- Measure – This is the distance from the beginning station of the cross section to the point property.

4.11.5.4 Line Properties subgroup descriptions

The Line Properties subgroup contains the following information:

- NumLineProperties – This is a single integer representing the number of line properties.
- Xsecs – This is an integer array that references the cross section to which the line property belongs.
- Type – This is an integer array of flags that represent the type of line property. Initially the values will be 1 – material.
- BegMeasure – This is the distance from the beginning station of the cross section to the beginning of the line property.
- EndMeasure – This is the distance from the beginning station of the cross section to the ending of the line property.

4.11.5.5 CrossSectionGeometry subgroup descriptions

The CrossSectionGeometry subgroup contains the Locations, a 3-D array of doubles for the d- and z-locations of the cross-section points. The size of the array is 2 by **NumXsecs**, by **Maxnumdz**. Actually **Maxnumdz** should be a variable length, which is thought possible and will be looked into further.

4.11.6 API functions

The C and FORTRAN functions/subroutines for cross-section data are as follows:

4.11.6.1 XSecs subgroup

C/C++

```
int xfSetNumberOfXsecs(xid GroupId, int Num);
int xfGetNumberOfXsecs(xid GroupId, int *Num);
int xfSetXsecRiverNames(xid GroupId, const char *Names);
int xfGetXsecRiverNames(xid GroupId, char **Names);
int xfSetXsecReachNames(xid GroupId, const char *Names);
int xfGetXsecReachNames(xid GroupId, char **Names);
int xfSetXsecStations(xid GroupId, double *Stations);
int xfGetXsecStations(xid GroupId, double *stations);
int xfSetXsecOrigins(xid GroupId, int *Origins);
int xfGetXsecOrigins(xid GroupId, int *Origins);
int xfSetXsecGeometryId(xid GroupId, int *IDs);
int xfGetXsecGeometryId(xid GroupId, int *IDs);
```

FORTRAN

```
SUBROUTINE  XF_SET_NUMBER_OF_XSECS(GroupId, Num, Error)
INTEGER(XID), INTENT(IN) :: GroupId
INTEGER, INTENT(IN)      :: Num
INTEGER, INTENT(OUT)     :: Error

SUBROUTINE  XF_GET_NUMBER_OF_XSECS(GroupId, Num, Error)
INTEGER(XID), INTENT(IN) :: GroupId
INTEGER, INTENT(OUT)     :: Num, Error

SUBROUTINE  XF_SET_XSEC_RIVER_NAMES(GroupId, Names, Error)
INTEGER(XID), INTENT(IN) :: GroupId
CHARACTER(len=*), INTENT(IN) :: Names
INTEGER, INTENT(OUT)     :: Error

SUBROUTINE  XF_GET_XSEC_RIVER_NAMES(GroupId, Names, Error)
INTEGER(XID), INTENT(IN) :: GroupId
CHARACTER(len=*), INTENT(INOUT) :: Names
INTEGER, INTENT(OUT)     :: Error

SUBROUTINE  XF_SET_XSEC_REACH_NAMES(GroupId, Names, Error)
INTEGER(XID), INTENT(IN) :: GroupId
CHARACTER(len=*), INTENT(IN) :: Names
INTEGER, INTENT(OUT)     :: Error

SUBROUTINE  XF_GET_XSEC_REACH_NAMES(GroupId, Names, Error)
INTEGER(XID), INTENT(IN) :: GroupId
CHARACTER(len=*), INTENT(INOUT) :: Names
INTEGER, INTENT(OUT)     :: Error

SUBROUTINE  XF_SET_XSEC_STATIONS(GroupId, Stations, Error)
INTEGER(XID), INTENT(IN) :: GroupId
REAL(DOUBLE), INTENT(IN) :: Stations(*)
INTEGER, INTENT(OUT)     :: Error

SUBROUTINE  XF_GET_XSEC_STATIONS(GroupId, Stations, Error)
INTEGER(XID), INTENT(IN) :: GroupId
```

```
REAL(DOUBLE), INTENT(OUT) :: Stations(*)
INTEGER, INTENT(OUT)      :: Error

SUBROUTINE XF_SET_XSEC_ORIGINS (GroupId, Origins, Error)
INTEGER(XID), INTENT(IN) :: GroupId
INTEGER, INTENT(IN)       :: Origins(*)
INTEGER, INTENT(OUT)      :: Error

SUBROUTINE XF_SET_XSEC_ORIGINS (GroupId, Origins, Error)
INTEGER(XID), INTENT(IN) :: GroupId
INTEGER, INTENT(OUT)       :: Origins(*)
INTEGER, INTENT(OUT)      :: Error

SUBROUTINE XF_SET_XSEC_GEOMETRY_ID (GroupId, IDs, Error)
INTEGER(XID), INTENT(IN) :: GroupId
INTEGER, INTENT(IN)       :: IDs(*)
INTEGER, INTENT(OUT)      :: Error

SUBROUTINE XF_GET_XSEC_GEOMETRY_ID (GroupId, IDs, Error)
INTEGER(XID), INTENT(IN) :: GroupId
INTEGER, INTENT(OUT)       :: IDs(*)
INTEGER, INTENT(OUT)      :: Error
```

4.11.6.2 Profile subgroup

C/C++

```
int xfSetNumberOfProfiles(xid GroupId, int Num);
int xfGetNumberOfProfiles(xid GroupId, int *Num);
int xfSetProfileRiverNames(xid GroupId, char **Names);
int xfGetProfileRiverNames(xid GroupId, char **Names);
int xfSetProfileReachNames(xid GroupId, char **Names);
int xfGetProfileReachNames(xid GroupId, char **Names);
int xfSetProfileTypes(xid GroupId, int *Types);
int xfGetProfileTypes(xid GroupId, int *Types);
int xfSetProfileOrigins(xid GroupId, int *Origins);
int xfGetProfileOrigins(xid GroupId, int *Origins);
int xfSetProfileSpatialGeometryId(xid GroupId, int *IDs);
int xfGetProfileSpatialGeometryId(xid GroupId, int *IDs);
```

FORTRAN

```
SUBROUTINE XF_SET_NUMBER_OF_PROFILES (GroupId, Num, Error)
INTEGER(XID), INTENT(IN) :: GroupId
INTEGER, INTENT(IN)       :: Num
INTEGER, INTENT(OUT)      :: Error

SUBROUTINE XF_GET_NUMBER_OF_PROFILES (GroupId, Num, Error)
INTEGER(XID), INTENT(IN) :: GroupId
INTEGER, INTENT(OUT)      :: Num, Error

SUBROUTINE XF_SET_PROFILE_RIVER_NAMES (GroupId, Names, Error)
INTEGER(XID), INTENT(IN) :: GroupId
CHARACTER(len=*), INTENT(IN) :: Names
INTEGER, INTENT(OUT)      :: Error

SUBROUTINE XF_GET_PROFILE_RIVER_NAMES (GroupId, Names, Error)
INTEGER(XID), INTENT(IN) :: GroupId
CHARACTER(len=*), INTENT(INOUT) :: Names
INTEGER, INTENT(OUT)      :: Error

SUBROUTINE XF_SET_PROFILE_REACH_NAMES (GroupId, Names, Error)
```

```
INTEGER(XID), INTENT(IN) :: GroupId
CHARACTER(len=*), INTENT(IN) :: Names
INTEGER, INTENT(OUT)     :: Error

SUBROUTINE XF_GET_PROFILE_REACH_NAMES (GroupId, Names, Error)
INTEGER(XID), INTENT(IN) :: GroupId
CHARACTER(len=*), INTENT(INOUT) :: Names
INTEGER, INTENT(OUT)     :: Error

SUBROUTINE XF_SET_PROFILE_TYPES (GroupId, Types, Error)
INTEGER(XID), INTENT(IN) :: GroupId
INTEGER, INTENT(IN)      :: Types(*)
INTEGER, INTENT(OUT)     :: Error

SUBROUTINE XF_GET_PROFILE_TYPES (GroupId, Types, Error)
INTEGER(XID), INTENT(IN) :: GroupId
INTEGER, INTENT(OUT)      :: Types(*)
INTEGER, INTENT(OUT)     :: Error

SUBROUTINE XF_SET_PROFILE_ORIGINS (GroupId, Origins, Error)
INTEGER(XID), INTENT(IN) :: GroupId
INTEGER, INTENT(IN)      :: Origins(*)
INTEGER, INTENT(OUT)     :: Error

SUBROUTINE XF_GET_PROFILE_ORIGINS (GroupId, Origins, Error)
INTEGER(XID), INTENT(IN) :: GroupId
INTEGER, INTENT(OUT)      :: Origins(*)
INTEGER, INTENT(OUT)     :: Error

SUBROUTINE XF_SET_PROFILE_SPATIAL_GEOMETRY_ID (GroupId, IDs, Error)
INTEGER(XID), INTENT(IN) :: GroupId
INTEGER, INTENT(IN)      :: IDs(*)
INTEGER, INTENT(OUT)     :: Error

SUBROUTINE XF_GET_PROFILE_SPATIAL_GEOMETRY_ID (GroupId, IDs, Error)
INTEGER(XID), INTENT(IN) :: GroupId
INTEGER, INTENT(OUT)      :: IDs(*)
INTEGER, INTENT(OUT)     :: Error
```

4.11.6.3 Point Properties subgroup

C/C++

```
int xfSetNumberOfXsecPointProperties(xid GroupId, int Num);
int xfGetNumberOfXsecPointProperties (xid GroupId, int *Num);
int xfSetXsecPointPropertiesXsecs(xid GroupId, int *Xsecs);
int xfGetXsecPointPropertiesXsecs(xid GroupId, int *Xsecs);
int xfSetXsecPointPropertiesTypes(xid GroupId, int *Types);
int xfGetXsecPointPropertiesTypes(xid GroupId, int *Types);
int xfSetXsecPointPropertiesMeasures(xid GroupId, double *Measures);
int xfGetXsecPointPropertiesMeasures (xid GroupId, double *Measures);
```

FORTRAN

```
SUBROUTINE XF_SET_NUMBER_OF_XSEC_POINT_PROPERTIES (GroupId, Num, Error)
INTEGER(XID), INTENT(IN) :: GroupId
INTEGER, INTENT(IN)      :: Num
INTEGER, INTENT(OUT)     :: Error

SUBROUTINE XF_GET_NUMBER_OF_XSEC_POINT_PROPERTIES (GroupId, Num, Error)
INTEGER(XID), INTENT(IN) :: GroupId
INTEGER, INTENT(OUT)     :: Num, Error
```

```
SUBROUTINE XF_SET_XSEC_POINT_PROPERTIES_XSECS (GroupId, Xsecs, Error)
INTEGER(XID), INTENT(IN) :: GroupId
INTEGER, INTENT(IN)      :: Xsecs(*)
INTEGER, INTENT(OUT)     :: Error

SUBROUTINE XF_GET_XSEC_POINT_PROPERTIES_XSECS (GroupId, Xsecs, Error)
INTEGER(XID), INTENT(IN) :: GroupId
INTEGER, INTENT(OUT)      :: Xsecs(*)
INTEGER, INTENT(OUT)     :: Error

SUBROUTINE XF_SET_XSEC_POINT_PROPERTIES_TYPES (GroupId, Types, Error)
INTEGER(XID), INTENT(IN) :: GroupId
INTEGER, INTENT(IN)      :: Types(*)
INTEGER, INTENT(OUT)     :: Error

SUBROUTINE XF_GET_XSEC_POINT_PROPERTIES_TYPES (GroupId, Types, Error)
INTEGER(XID), INTENT(IN) :: GroupId
INTEGER, INTENT(OUT)      :: Types(*)
INTEGER, INTENT(OUT)     :: Error

SUBROUTINE XF_SET_XSEC_POINT_PROPERTIES_MEASURES (GroupId, Measures,
Error)
INTEGER(XID), INTENT(IN) :: GroupId
REAL(DOUBLE), INTENT(IN) :: Measures(*)
INTEGER, INTENT(OUT)     :: Error

SUBROUTINE XF_GET_XSEC_POINT_PROPERTIES_MEASURES (GroupId, Measures,
Error)
INTEGER(XID), INTENT(IN) :: GroupId
REAL(DOUBLE), INTENT(OUT) :: Measures(*)
INTEGER, INTENT(OUT)     :: Error
```

4.11.6.4 Line Properties subgroup

C/C++

```
int xfSetNumberOfXsecLineProperties(xid GroupId, int Num);
int xfGetNumberOfXsecLineProperties (xid GroupId, int *Num);
int xfSetXsecLinePropertiesXsecs(xid GroupId, int *Xsecs);
int xfGetXsecLinePropertiesXsecs(xid GroupId, int *Xsecs);
int xfSetXsecLinePropertiesTypes(xid GroupId, int *Types);
int xfGetXsecLinePropertiesTypes(xid GroupId, int *Types);
int xfSetXsecLinePropertiesBegMeasures(xid GroupId, double *Measures);
int xfGetXsecLinePropertiesBegMeasures (xid GroupId, double *Measures);
int xfSetXsecLinePropertiesEndMeasures(xid GroupId, double *Measures);
int xfGetXsecLinePropertiesEndMeasures (xid GroupId, double *Measures);
```

FORTRAN

```
SUBROUTINE XF_SET_NUMBER_OF_XSEC_LINE_PROPERTIES (GroupId, Num, Error)
INTEGER(XID), INTENT(IN) :: GroupId
INTEGER, INTENT(IN)      :: Num
INTEGER, INTENT(OUT)     :: Error

SUBROUTINE XF_GET_NUMBER_OF_XSEC_LINE_PROPERTIES (GroupId, Num, Error)
INTEGER(XID), INTENT(IN) :: GroupId
INTEGER, INTENT(OUT)     :: Num, Error

SUBROUTINE XF_SET_XSEC_LINE_PROPERTIES_XSECS (GroupId, Xsecs, Error)
INTEGER(XID), INTENT(IN) :: GroupId
INTEGER, INTENT(IN)      :: Xsecs(*)
```

```
INTEGER, INTENT(OUT)     :: Error

SUBROUTINE XF_GET_XSEC_LINE_PROPERTIES_XSECS (GroupId, Xsecs, Error)
INTEGER(XID), INTENT(IN) :: GroupId
INTEGER, INTENT(OUT)      :: Xsecs(*)
INTEGER, INTENT(OUT)     :: Error

SUBROUTINE XF_SET_XSEC_LINE_PROPERTIES_TYPES (GroupId, Types, Error)
INTEGER(XID), INTENT(IN) :: GroupId
INTEGER, INTENT(IN)      :: Types(*)
INTEGER, INTENT(OUT)     :: Error

SUBROUTINE XF_GET_XSEC_LINE_PROPERTIES_TYPES (GroupId, Types, Error)
INTEGER(XID), INTENT(IN) :: GroupId
INTEGER, INTENT(OUT)      :: Types(*)
INTEGER, INTENT(OUT)     :: Error

SUBROUTINE XF_SET_XSEC_LINE_PROPERTIES_BEG_MEASURES (GroupId, Measures,
Error)
INTEGER(XID), INTENT(IN) :: GroupId
REAL(DOUBLE), INTENT(IN) :: Measures(*)
INTEGER, INTENT(OUT)     :: Error

SUBROUTINE XF_GET_XSEC_LINE_PROPERTIES_BEG_MEASURES (GroupId, Measures,
Error)
INTEGER(XID), INTENT(IN) :: GroupId
REAL(DOUBLE), INTENT(OUT) :: Measures(*)
INTEGER, INTENT(OUT)     :: Error

SUBROUTINE XF_SET_XSEC_LINE_PROPERTIES_END_MEASURES (GroupId, Measures,
Error)
INTEGER(XID), INTENT(IN) :: GroupId
REAL(DOUBLE), INTENT(IN) :: Measures(*)
INTEGER, INTENT(OUT)     :: Error

SUBROUTINE XF_GET_XSEC_LINE_PROPERTIES_END_MEASURES (GroupId, Measures,
Error)
INTEGER(XID), INTENT(IN) :: GroupId
REAL(DOUBLE), INTENT(OUT) :: Measures(*)
INTEGER, INTENT(OUT)     :: Error
```

4.11.6.5 CrossSectionGeometry subgroup

C/C++

```
int xfSetXsecGeometry(xid GroupId, double *Locations);
int xfGetXsecGeometry(xid GroupId, double *Locations);
```

FORTRAN

```
SUBROUTINE XF_SET_XSEC_GEOMETRY (GroupId, Locations, Error)
INTEGER(XID), INTENT(IN) :: GroupId
REAL(DOUBLE), INTENT(IN) :: Locations(*)
INTEGER, INTENT(OUT)     :: Error

SUBROUTINE XF_GET_XSEC_GEOMETRY (GroupId, Locations, Error)
INTEGER(XID), INTENT(IN) :: GroupId
REAL(DOUBLE), INTENT(OUT) :: Locations(*)
INTEGER, INTENT(OUT)     :: Error
```

4.12 Geometric Paths

Geometric paths store coordinate locations of nodes or particles. Also,

- The position of each node (x, y, z) varys over time. This requires that the array used to store these positions become a 3-D array rather than a 2-D array.
- Since the position is moving, it is difficult to determine which entities intersect a specific region. To address this, the Spatial Bins array can be included.
- There is no connectivity of elements, only strings of locations for a specific geometric entity along its path.

Data sets may be associated with a geometric path group. Individual paths may be inactive for portions of the time range represented by the data. This inactive period could be at the beginning (particle does not start at the start of the time range), in the middle of the time range (particle becomes static or inactive for a period of time), or at the end (particle stops or leaves the domain). In these cases, a data set would be used to store the velocity magnitude of each node at each time. If the velocity magnitude is a null data flag, the node is inactive at that time.

In some known applications the number of particles may increase over time. The geometric paths group is set up in such a way that particles may be added after the group is initialized. Data sets associated with these groups will also need to expand. The particle data are written one time-step at a time and are stored in a chunked layout (necessary to allow for growth). It might be advantageous to convert this to a contiguous layout using the repack command-line utility for faster data access (post-processing).

4.12.1 Group organization

The geometric path groups are described as follows:

- NumPaths – This is a single integer representing the number of paths in the group.
- NumTimes – This is the number of times at which points can be saved along a path.
- Times – This is a 1-D array of the time values at which values are stored. The time values are stored as Julian dates.
- Locations – This is a 3-D array of doubles for the x-, y-, and z-location of the points on the geometric paths. The size of the array is 3 × NumPaths × NumTimes.
- Mins, Maxs – Single-dimensional size 3 arrays that store the minimum and maximum x-, y-, and z-values encountered. This may be of interest to users as well as helpful for building spatial grids.

A geometric path group may have associated with it a 2-D array of lists. This is the spatial bins array. Each entry in the 2-D array corresponds to a portion of the domain covered by the paths. That array entry includes a list of path identifiers that cross this spatial bin, along with a range of times when that path is in this bin. The spatial bins functionality may not be implemented until a later date.

4.12.2 API functions

When an API function is called that fills in an array, the array must already be allocated. The path used in all of these functions is the path to the geometric path folder.

4.12.2.1 Writing

These functions are used to store the properties and attributes of a geometric path group. Those using this API to read the geometric path group will have to know this in order to properly allocate their arrays.

The function **xfCreateGeometricPath** group creates the location in the HDF5 file to store the geometric path group. This function requires a GUID because GUIDs are used to match data sets to spatial objects. The NullLoc argument must be an array of size 3 and is used to define the fill value for null particles. If the number of nodes/particles increases in the time range, newly created nodes are added to all time-steps, including previously stored time-steps. Since these particles did not exist at these time values, their location for time-steps before they became active will be stored as the NullLoc.

The function **xfWriteParticleTimestep** is used to write all of the particle information for a single time-step. The function first checks to determine whether the number of particles (nPaths) has not changed (it can be increased but not decreased). If the number of particles has increased, the function will extend the HDF5 data set and fill in the previous time-steps for these elements with the Nullvalue. The time-step dimension is then extended by one. Lastly, the data for the new locations are stored in the HDF5 data set. In C, the Locs array is a single-dimensional double array that represents the x-, y-, and z-values for all of the particles (the array is size number of particles * 3). The x-values are stored in indices 0, 3, 6, etc. Likewise, the y-values are stored in indices 1, 4, 7, etc., and z in 2, 5, 8, etc. In FORTRAN the Locs array is a double-dimensional array 3 x nParticles.

C/C++

```
int xfCreateGeometricPathGroup(xid ParentId, const char *Path,
                               const char *Guid, int Compression,
                               xid *PathGroup, double *NullLoc);
int xfWriteParticleTimestep(xid PathGroup, double Time, int nPaths,
                            double *Locs);
```

4.12.2.2 Reading

These functions are provided to read the properties and attributes of a geometric path group.

Particle path locations can be read for multiple indices at a specific time or for a specific particle at multiple times. In each case the Locs array must already be allocated to the correct size.

C/C++

```
int xfGetNumberOfPaths(xid GroupId, int *NumPaths);
int xfGetNumberOfTimes(xid GroupId, int *NumTimes);
int xfGetPathTimesArray(xid GroupId, int NumTimes, double *Times);
int xfReadPathLocationsAtTime(xid GroupId, int TimeIndex,
                                    int FirstPathIndex,
                                    int NumIndicies, double *Locs);
int xfReadPathLocationsForParticle(xid GroupId, int PathIndex,
                                   int FirstTimeIndex, int
NumTimes,
                                   double *Locs);
int xfReadPathLocationsForParticles(xid GroupId, int NumPaths, int *PathIndices,
                                    int FirstTimeIndex, int NumTimes,
                                    double *Locs);
```

4.12.3 Spatial bins

Spatial queries are commonly performed upon dynamic particle paths. An example would be to report all particles that end within a specific bounding box or polygon. This can easily be determined by looking at the particle locations on the final time-step. More difficult, however, are queries such as “find all particles that pass through a specific polygon at any time.” This type of a query would require looking at every particle location for every time-step. This process can be sped up considerably by storing some extra data.

Because it is desired to take advantage of HDF5 functionality supported only when working with C and when the XMS programs will be doing the queries, the spatial bins functionality will be available only in the C version of the library.

The proposed methodology is to divide the domain into a regular grid of spatial bins, and for each bin store the particles that travel through it and for each particle the time-step indices where the particle enters and leaves the bin. These indices would actually be the time-step before entering the bin and the time-step after leaving the bin so that reading these time-step indices for the particles will have all the segments that pass through any part of the bin. Since a particle may enter and leave a bin several times, there will be a variable number of pairs of entering/exiting time-steps.

This additional information will make it very efficient to perform spatial queries on the particles. At least initially this information will have to be computed outside XMDF but can be stored and queried here. The advantage of saving these data back to the file is that the information can be computed once and used many times over. The extra data are not stored in RAM.

Current implementation will build only 2-D bins. This will be expanded to 3-D bins if necessary. Three-dimensional bins consume enough memory to delay their implementation if possible.

The minimum and maximum values for the grid (which should correspond to the minimum and maximum values for the particle paths) are used to define the bins.

Particles on the boundary of bins should have entries placed in the bins on both sides of the boundary. Likewise particles on a corner should have entries for all bins touching the corner.

A bin may reference a particle even if none of the time-step values are in a cell. This would happen if the straight line path between successive time-steps crosses over the cell.

C/C++

```
int xfDefineGrid(xid PartGroup, int nXBin, int nYBins, int *nPointsInBin,
                 int **PointsInBin)
int xfWriteParticleBins(xid PartGroup, int nTimesteps, int
*BinsTraversed)
```

These functions are used to read the spatial bins for making spatial inquiries on the geometric path group. In every case, the memory should already be allocated before calling the functions.

C/C++

```
int xfIsSpatialGridDefined(xid PartGroup, xbool *bDefined)
int xfGetNumCellsInXandY(xid PartGroup, int *nXCells, int *nYCells)

int xfGetNumParticlesInBin(xid PartGroup, int Bin, int *nParticles)
int xfGetParticlesInBin(xid PartGroup, int Bin, int nParticles,int
                        *PartIndicies)

int xfReadBinsParticleVisits(xid PartGroup, int PartIndex, int nTimesteps,
                             int *BinsTraversed)
```

4.13 Data Sets

Data sets are subgroups associated with another group (a mesh, grid, or geometric path group). They are stored in a special folder (group) (Figure 14) that contains subfolders for individual data sets. Data sets can be either scalar (a single value per geometric entity) or vector (two or three values per geometric entity).

Scalar data sets are stored as a 2-D array where the first index is the time-step index and the second index is the node or cell index.

Vector data sets are stored as a 3-D array. The first index is for the time-step, the second index is for the node or cell, and the third index is for the individual components. The first index (0 in C, 1 in FORTRAN) corresponds to the x-component, the second index corresponds to the y-component, and the third index corresponds to the z-component. For grids the components may be in i-, j-, k-coordinates rather than x, y, z.

The data locations for meshes are always at mesh nodes. Data locations for grids may be at centers, nodes, all faces, or only on faces in a particular direction (one value per cell). The data locations for vector data sets may be different for each component. For example, the 2-D hydraulic model M2D computes and reports velocities at cell faces. Velocity in the i-direction is given at cell faces perpendicular to the i-direction. Likewise, velocity in the j-direction is given at cell faces perpendicular to the j-direction.

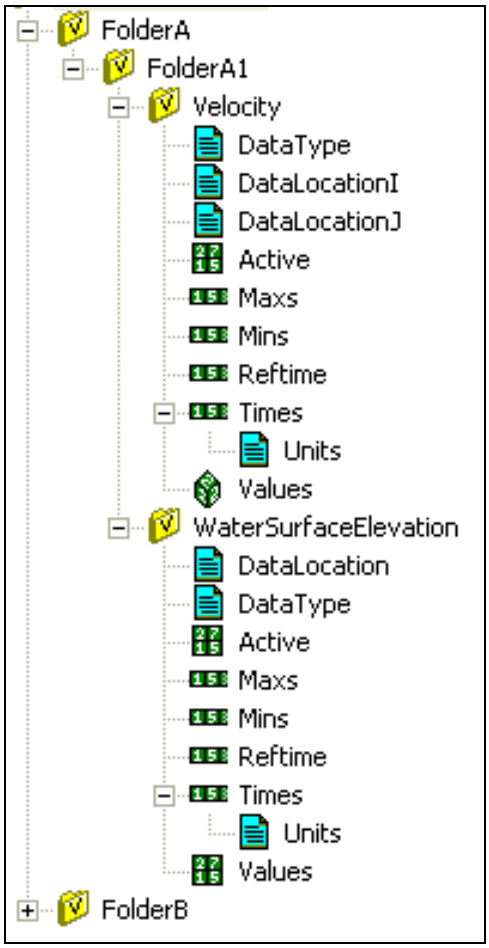


Figure 14. Schematic including data set folders

Particular elements or cells may have values at one time-step but not have values at another. This happens frequently in hydraulic models. When flow rates are decreasing, areas that are inundated can dry, leaving an area of the mesh dry that was previously wet. These situations are handled by an optional activity array. This activity array is a 2-D array of on/off values indicating whether a specific element or cell is active for a given time-step and element or cell. Activity arrays are always done on a cell-centered basis regardless of whether the data are mesh or cell-centered. The activity array is a 2-D array where the first index is the time-step index and the second index is the cell or element index.

Data sets stored using XMDF include the minimum and maximum values for each time-step. Although this information is not necessary for a data set, it is useful for visualization packages. The minimum and maximum values are automatically determined when data sets are written using the XMDF API.

Because data sets can take up large amounts of disk space, XMDF allows data sets to be compressed. Compression is performed using the default compression algorithm in HDF5.

4.13.1 API functions

4.13.1.1 Multiple data sets groups

A multiple data sets group stores data sets for a specific spatial data object (mesh or grid). The mesh or grid that the data sets are to be used with is identified by a global unique identifier or GUID. The following functions are used to create multiple data sets groups, retrieve paths to multiple data sets groups, and retrieve the GUID for the spatial data object. The GUID string should be of size XF_GUID_STRINGLENGTH as defined in the XMDF source code.

C/C++

```
int xfCreateMultiDatasetsGroup(xid ParentId, const char *Path,
                               const char *Guid);
int xfGetGroupPathsSizeForMultiDatasets(xid FileId, int *Num, int *MaxSize);
int xfGetAllGroupPathsForMultiDatasets(xid FileId, int Num, int Maxsize,
                                       char *path);
int xfGetDatasetsSdoGuid(xid MultiDatasetsGroup, char *GUID);
```

FORTRAN

```
SUBROUTINE  XF_CREATE_MULTI_DATASETS_GROUP(ParentId, Path, Guid, Error)
INTEGER(XID), INTENT(IN) :: ParentId
CHARACTER(LEN=*), INTENT(IN) :: Path, Guid
INTEGER, INTENT(OUT)     :: Error

SUBROUTINE XF_GET_GROUP_PATHS_SIZE_FOR_MULTI_DATASETS (FileId, Num,
Maxsize, Error)
INTEGER(XID), INTENT(IN) ::   FileId
INTEGER, INTENT(OUT) ::       Num, Maxsize, Error

SUBROUTINE XF_GET_ALL_GROUP_PATHS_FOR_MULTI_DATASETS (FileId, Num, Size,
Paths, Error)
INTEGER(XID), INTENT(IN) ::     FileId
INTEGER, INTENT(IN) ::  Num, Size
CHARACTER(len=Size), DIMENSION(Size, Num), INTENT(INOUT) ::  Paths
INTEGER, INTENT(OUT) :: Error

SUBROUTINE  XF_GET_DATASETS_SDO_GUID(MultiDatasetsGroup, Guid)
INTEGER(XID), INTENT(IN) :: ParentId
CHARACTER(LEN=XF_GUID_STRINGLENGTH), INTENT(OUT) :: Guid
INTEGER, INTENT(OUT)     :: Error
```

The multiple data sets group for a spatial data object is created automatically upon first use. The following function is used to open the data sets group associated with a spatial data object. Note this is the only case where a GUID is not needed for a multiple data sets group because the spatial data object that the data sets belong to is obvious.

C/C++

```
int xfOpenMultiDatasetsGroup(xid GridGroupId, xid *DatasetsGroupId);
```

FORTRAN

```
SUBROUTINE  XF_OPEN_MULTI_DATASETS_GROUP(GridGroupId, DatasetsGroupId, Error)
INTEGER(XID), INTENT(IN) :: GroupId
INTEGER(XID), INTENT(OUT):: DatasetsGroupId
INTEGER, INTENT(OUT)     :: Error
```

4.13.1.2 Shortcut to setup writing to data sets

A shortcut is provided in XMDF to allow model developers to easily set up the files and groups to write data sets. This shortcut is the function **xfSetupToWriteDatasets**. This function should always be used by model developers because it ensures that all models used with XMDF have a consistent and easy-to-use method to begin writing data sets.

The arguments for **xfSetupToWriteDatasets** should be given to the model using an external text file or command line arguments. The arguments include the filename to save the data sets, the path to a special group called the multi-data-sets group path, the path to start writing data sets in this path, the GUID for the spatial data object that all the data sets will be tied to, and the overwrite options. The overwrite options are listed by number and the corresponding C/FORTRAN constant (either the number or the constant may be used):

- XF_OVERWRITE_CLEAR_FILE – File is intended only for the data sets being written by this model. If a file exists with the filename, delete the file.
- XF_OVERWRITE_CLEAR_DATASET_GROUP – Leave existing file if one exists but clear any existing data sets or groups currently in the path where data sets will be written (PathInMultiDatasetsGroup).
- XF_OVERWRITE_NONE – Leave existing file and all data within the file. When the data sets are created they will overwrite any data sets with the same name and path.

C/C++

```
int xfSetupToWriteDatasets(const char *Filename,
                           const char *MultiDatasetsGroupPath,
                           const char *PathInMultiDatasetsGroup,
                           const char *SpatialDataObjectGuid,
                           int OverwriteOptions,
                           xid *FileId,
                           xid *GroupId);
```

FORTRAN

```
SUBROUTINE  XF_SETUP_TO_WRITE_DATASETS (Filename,
MultiDatasetsGroupPath,
                PathInMultiDatasetsGroup, SpatialDataObjectGuid,
                FileId, GroupId, Error)
INTEGER(XID), INTENT(IN)     :: GroupId
CHARACTER(len=*), INTENT(IN) :: Filename, MultiDatasetsGroupPath
CHARACTER(len=*), INTENT(IN) :: PathInMultiDatasetsGroup, SpatialDataObjectGuid
INTEGER(XID), INTENT(OUT)    :: FileId, GroupId
```

```
INTEGER, INTENT(OUT)          :: Error
```

4.13.1.3 Creating and writing data sets

The following functions are used to create data sets to write to HDF5. The units for the data set are specified when the data set is created. The maximum length for the units is 100 characters.

Valid time units are identified using the constants TS_DAYS, TS_HOURS, TS_MINUTES, TS_SECONDS, or TS_NOTAPPLICABLE. The compression value is applied to all arrays in the data set.

C/C++

```
int xfCreateScalarDataset(xid DatasetGroupId, const char *Path, const char *Units,
                          const char *TimeUnits, int Compression, xid *DatasetId);
int xfCreateVectorDataset(xid DatasetGroupId, const char *Path, const char *Units,
                          const char *TimeUnits, int Compression, xid *DatasetId);
```

FORTRAN

```
SUBROUTINE  XF_CREATE_SCALAR_DATASET(DatasetsGroupId, Path, Units, TimeUnits,
                                     Compression, DatasetId, Error)
INTEGER(XID), INTENT(IN)  :: DatasetsGroupId
CHARACTER(len=*), INTENT(IN) :: Path, Units, TimeUnits
INTEGER, INTENT(IN) :: Compression
INTEGER(XID), INTENT(OUT) :: DatasetId
INTEGER, INTENT(OUT)      :: Error


SUBROUTINE  XF_CREATE_VECTOR_DATASET(DatasetsGroupId, Path, Units, TimeUnits,
                                     Compression, DatasetId, Error)
INTEGER(XID), INTENT(IN)  :: DatasetsGroupId
CHARACTER(len=*), INTENT(IN) :: Path, Units, TimeUnits
INTEGER, INTENT(IN) :: Compression
INTEGER(XID), INTENT(OUT) :: DatasetId
INTEGER, INTENT(OUT)      :: Error
```

All date/time values stored in XMDF should be stored as Julian dates for consistency. A Julian day is the absolute count of days that have elapsed since Noon Universal Time on January 1, 4713 BCE on the Julian calendar (*http://aa.usno.navy.mil/data/docs/JulianDate.html*). While Julian dates are a good standard date format to follow, they are not very meaningful to most people. XMDF contains functions to convert calendar days to Julian days and Julian days to calendar days. The conversion algorithms were adapted from code on a Web site maintained by the U.S. Naval Observatory (*http://aa.usno.navy.mil/data/docs/JulianDate.html*).

C/C++

```
int xfCalendarToJulian (xbool a_bBceEra, int a_yr, int a_mo, int a_day,
                        int a_hr, int a_min, int a_sec, double *a_julian);
int xfJulianToCalendar (xbool *a_bBceEra, int *a_yr, int *a_mo, int *a_day,
                         int *a_hr, int *a_min, int *a_sec, double a_julian);
```

FORTRAN

```
SUBROUTINE XF_CALENDAR_TO_JULIAN (era, yr, mo, day, hr, &
                                  min, sec, julian, error)
  INTEGER, INTENT(IN)        :: era
  INTEGER, INTENT(IN)        :: yr
  INTEGER, INTENT(IN)        :: mo
  INTEGER, INTENT(IN)        :: day
  INTEGER, INTENT(IN)        :: hr
  INTEGER, INTENT(IN)        :: min
  INTEGER, INTENT(IN)        :: sec
  REAL(DOUBLE), INTENT(OUT) :: julian
  INTEGER, INTENT(OUT)       :: error

SUBROUTINE XF_JULIAN_TO_CALENDAR (era, yr, mo, day, hr, &
                                  min, sec, julian, error)
  INTEGER, INTENT(OUT) :: era
  INTEGER, INTENT(OUT) :: yr
  INTEGER, INTENT(OUT) :: mo
  INTEGER, INTENT(OUT) :: day
  INTEGER, INTENT(OUT) :: hr
  INTEGER, INTENT(OUT) :: min
  INTEGER, INTENT(OUT) :: sec
  REAL(DOUBLE), INTENT(IN) :: julian
  INTEGER, INTENT(OUT) :: error
```

The following function is used to store a reference time for a data set. This reference time should be a double precision float specifying the Julian day of time zero for the simulation. The data set times are an array of offset values from the reference time.

C/C++

```
int xfDatasetReftime(xid DatasetId, double Reftime);
```

FORTRAN

```
SUBROUTINE  XF_DATASET_REFTIME(DatasetId, Reftime, Error)
INTEGER(XID), INTENT(IN)  :: DatasetsGroupId
REAL(DOUBLE), INTENT(IN)  :: Reftime
INTEGER, INTENT(OUT)      :: Error
```

Data sets are written as single precision floats by default. When data set values are appended, the number of values must match the number of values previously written. The time-steps must be written in chronological order. Scalar value arrays are 2-D arrays of size **NumValues**. Vector value arrays are 3-D arrays of size **NumValues X NumComponents** (reversed for FORTRAN). For the C library, all arrays must be contiguous arrays of the correct size.

XMDF stores the minimum and maximum values for each time-step in the data set. The minimum and maximum values can be specified or they are determined automatically from the values arrays. For vector data sets the minimum and maximum values are the minimum and maximum values for vector magnitudes.

C/C++

```
int xfWriteScalarTimestep(xid DatasetId, double Time, int NumValues,
                          float *Values);
int xfWriteScalarTimestepMinMax(xid DatasetId, double Time, int NumValues,
                                  float *Values, float Min, float Max);

int xfWriteVectorTimestep(xid DatasetId, double Time, int NumValues,
                          int NumComponents, float *Values);
int xfWriteVectorTimestepMinMax(xid DatasetId, double Time, int NumValues,
                                  int NumComponents, float *Values, float Min,
                                  float Max);
```

FORTRAN

```
SUBROUTINE  XF_WRITE_SCALAR_TIMESTEP(DatasetId, Time, NumValues, Values, Error)
INTEGER(XID), INTENT(IN)  :: DatasetId
REAL(DOUBLE), INTENT(IN)  :: Time
INTEGER, INTENT(IN)       :: NumValues
REAL, DIMESNION(NumValues), INTENT(IN)       :: Values
INTEGER, INTENT(OUT)      :: Error

SUBROUTINE  XF_WRITE_SCALAR_TIMESTEP_WITH_MIN_MAX(DatasetId, Time, NumValues,
Values,
                                          Min, Max, Error)
INTEGER(XID), INTENT(IN)  :: DatasetId
REAL(DOUBLE), INTENT(IN)  :: Time
INTEGER, INTENT(IN)       :: NumValues
REAL, DIMENSION(NumValues), INTENT(IN)       :: Values
REAL, INTENT(IN)          :: Min, Max
INTEGER, INTENT(OUT)      :: Error

SUBROUTINE  XF_WRITE_VECTOR_TIMESTEP(DatasetId, Time, NumValues, Values, Error)
INTEGER(XID), INTENT(IN)  :: DatasetId
REAL(DOUBLE), INTENT(IN)  :: Time
INTEGER, INTENT(IN)       :: NumValues
REAL, DIMESNION(NumComponents, NumValues), INTENT(IN)       :: Values
INTEGER, INTENT(OUT)      :: Error

SUBROUTINE  XF_WRITE_VECTOR_TIMESTEP_WITH_MIN_MAX(DatasetId, Time, NumValues,
Values, Min, Max, Error)
INTEGER(XID), INTENT(IN)  :: DatasetId
REAL(DOUBLE), INTENT(IN)  :: Time
INTEGER, INTENT(IN)       :: NumValues
REAL, DIMENSION(NumComponents, NumValues), INTENT(IN)       :: Values
REAL, INTENT(IN)          :: Min, Max
INTEGER, INTENT(OUT)      :: Error
```

4.13.1.4 Activity information

Activity/Inactive data set values can be specified in two ways. With the first method, a null value is defined for a data set. Any time a data set value is the null value, any elements or cells containing the data value will be considered inactive for that time-step. When null values are used, they must be set before writing the data set values or else the minimum and maximum values may be incorrect. A null value is set by writing a float property to the data-set group using the defined name PROP_NULL_VALUE. The second method to storing active/inactive data set locations is by using an activity array. For C, activity arrays are 2-D arrays of unsigned characters of size number of times X the number of active values. For

FORTRAN, the activity arrays are read/written using integers. The number of active values is based upon the number of cells or elements in the grid or mesh.

C/C++

```
int xfWriteActivityTimestep(xid DatasetId, int NumActiveVals, xbool *Active);
```

FORTRAN

```
SUBROUTINE  XF_WRITE_ACTIVITY_TIMESTEP(DatasetId, NumActiveVals, ActiveVals,
Error)
INTEGER(XID), INTENT(IN)  :: DatasetId
INTEGER, INTENT(IN)       :: NumActiveVals
INTEGER, DIMENSION(NumActiveVals), INTENT(IN)   :: ActiveVals
INTEGER, INTENT(OUT)      :: Error
```

4.13.1.5 Reading data sets

To read data sets from an XMDF file, first open the data-set group and get a group ID. If the file contains only data sets, this group ID is the file ID. The following function is used to get the data-set group ID from a mesh or grid. The variable **EntityId** is the group ID for the mesh or grid.

C/C++

```
int xfGetDatasetGroupId(xid EntityId, xid *DatasetsGroupId);
```

FORTRAN

```
SUBROUTINE  XF_GET_DATASET_GROUP_ID(EntityId, DatasetsGroupId, Error)
INTEGER(XID), INTENT(IN)  :: EntityId
INTEGER(XID), INTENT(OUT) :: DatasetsGroupId
INTEGER, INTENT(OUT)      :: Error
```

The following function is called to get the number of scalar data sets and the maximum path length for scalar data sets inside a data sets folder. This information is necessary to allocate the arrays necessary to retrieve the scalar data set paths.

C/C++

```
int xfGetScalarDatasetsInfo(xid DatasetsId, int *Number,
                            int *MaxPathLength);
```

FORTRAN

```
SUBROUTINE  XF_GET_SCALAR_DATASETS_INFO(DatasetsId, Number, MaxPathLength, Error)
INTEGER(XID), INTENT(IN)  :: DatasetsId
INTEGER, INTENT(OUT)      :: Number, MaxPathLength
INTEGER, INTENT(OUT)      :: Error
```

The next function is used to get the paths to all scalar data sets inside a folder of data sets. These paths are the subsequent group paths to get to a data set divided by backward slashes "\". Paths must be allocated to a size of **NumDataSets X MaximumPathLength** as obtained from **GetScalarDatasetsInfo**.

C/C++

```
int xfGetScalarDatasetPaths(xid DatasetsId, NumDatasets, MaxPathLength,
                            char *Paths);
```

FORTRAN

```
SUBROUTINE  XF_GET_SCALAR_DATASET_PATHS(DatasetsId, NumDatasets, MaxPathLength,
                                        Paths, Error)
INTEGER(XID), INTENT(IN)  :: DatasetsId
INTEGER, INTENT(IN)       :: NumDatasets, MaxPathLength
CHARACTER(len=MaxPathLength), DIMENSION(NumDatasets) :: Paths
INTEGER, INTENT(OUT)      :: Error
```

The following functions are used to obtain paths to vector data sets and work just like their scalar data set counterparts.

C/C++

```
int xfGetVectorDatasetsInfo(xid DatasetsId, int *Number,
                            int *MaxPathLength);
int xfGetVectorDatasetPaths(xid DatasetsId, int NumDatasets, int MaxPathLength,
                            char *Paths);
```

FORTRAN

```
SUBROUTINE  XF_GET_VECTOR_DATASETS_INFO(DatasetsId, Number, MaxPathLength, Error)
INTEGER(XID), INTENT(IN)  :: DatasetsId
INTEGER, INTENT(OUT)      :: Number, MaxPathLength
INTEGER, INTENT(OUT)      :: Error

SUBROUTINE  XF_GET_VECTOR_DATASET_PATHS(DatasetsId, NumDatasets, MaxPathLength,
                                        Paths, Error)
INTEGER(XID), INTENT(IN)  :: DatasetsId
INTEGER, INTENT(IN)       :: NumDatasets, MaxPathLength
CHARACTER(len=MaxPathLength), DIMENSION(NumDatasets) :: Paths
INTEGER, INTENT(OUT)      :: Error
```

The variable units must be a string of length at least 100.

C/C++

```
int xfGetDatasetUnits(xid DatasetId, char *Units);
```

FORTRAN

```
SUBROUTINE  XF_GET_DATASET_UNITS(DatasetsId, Units, Error)
INTEGER(XID), INTENT(IN)  :: DatasetsId
CHARACTER(len=100), INTENT(OUT) :: Units
INTEGER, INTENT(OUT)      :: Error
```

The following functions are used to read time values.

C/C++

```
int xfGetDatasetReftime(xid DatasetId, double *Reftime);
int xfGetDatasetNumTimes(xid DatasetId, int *NumTimes);
int xfGetDatasetTimeUnits(xid DatasetId, UnitType *Units);
```

FORTRAN

```
SUBROUTINE  XF_GET_DATASET_REFTIME(DatasetsId, Reftime, Error)
INTEGER(XID), INTENT(IN)  :: DatasetsId
REAL(DOUBLE), INTENT(OUT) :: Reftime
INTEGER, INTENT(OUT)      :: Error

SUBROUTINE  XF_GET_DATASET_NUMTIMES(DatasetsId, NumTimes, Error)
INTEGER(XID), INTENT(IN)  :: DatasetsId
INTEGER), INTENT(OUT)     :: NumTimes
INTEGER, INTENT(OUT)      :: Error

SUBROUTINE  XF_GET_DATASET_REFTIME(DatasetsId, Units, Error)
INTEGER(XID), INTENT(IN)  :: DatasetsId
CHARACTER(len=100), INTENT(OUT) :: Units
INTEGER, INTENT(OUT)      :: Error
```

This function is used to retrieve the time offsets for every time-step in the data set. The variable Times must already be allocated to the number of time-steps.

C/C++

```
int xfReadDatasetTimes(xid DatasetId, int NumTimes, double *Times);
```

FORTRAN

```
SUBROUTINE  XF_READ_DATASET_TIMES(DatasetsId, NumTimes, Times, Error)
INTEGER(XID), INTENT(IN)  :: DatasetsId
INTEGER, INTENT(IN)       :: NumTimes
REAL(DOUBLE), DIMENSION(NumTimes), INTENT(OUT) :: Times
INTEGER, INTENT(OUT)      :: Error
```

Minimum and maximum values at each time are written out automatically when a data set is stored. This information is used to develop contour intervals in visualization packages. The following functions are used to retrieve the minimum and maximum values for the time-steps. The arrays must already be allocated to the number of time-steps.

C/C++

```
int xfGetDatasetMins(xid DatasetId, float *Mins);
int xfGetDatasetMaxs(xid DatasetId, float *Maxs);
```

FORTRAN

```
SUBROUTINE  XF_GET_DATASET_MINS(DatasetsId, NumTimes, Mins, Error)
INTEGER(XID), INTENT(IN)  :: DatasetsId
INTEGER, INTENT(IN)       :: NumTimes
REAL(DOUBLE), DIMENSION(NumTimes), INTENT(OUT) :: Mins
INTEGER, INTENT(OUT)      :: Error

SUBROUTINE  XF_GET_DATASET_MINS(DatasetsId, NumTimes, Maxs, Error)
INTEGER(XID), INTENT(IN)  :: DatasetsId
INTEGER, INTENT(IN)       :: NumTimes
REAL(DOUBLE), DIMENSION(NumTimes), INTENT(OUT) :: Maxs
INTEGER, INTENT(OUT)      :: Error
```

The following function is used to determine the number of values in each time-step of a data set.

C/C++

```
int xfGetDatasetNumVals(xid DatasetId, int *NumVals);
```

FORTRAN

```
SUBROUTINE  XF_GET_DATASET_NUM_VALS(DatasetsId, NumVals, Error)
INTEGER(XID), INTENT(IN)  :: DatasetsId
INTEGER, INTENT(OUT)      :: NumVals
INTEGER, INTENT(OUT)      :: Error
```

The number of values in an activity array is not necessarily the same as the number of values in a data set. This function is used to determine the number of values in the activity array.

C/C++

```
int xfGetDatasetNumActiveVals(xid DatasetId, int *NumActiveVals);
```

FORTRAN

```
SUBROUTINE  XF_GET_DATASET_NUM_ACTIVE_VALS(DatasetsId,
NumActiveVals, Error)
INTEGER(XID), INTENT(IN)  :: DatasetsId
INTEGER, INTENT(OUT)      :: NumActiveVals
INTEGER, INTENT(OUT)      :: Error
```

The following function is used to retrieve the activity status for the data set. The variable Active must already be allocated to hold the number of active values.

C/C++

```
int xfReadActivityTimestep(xid DatasetId, int Timestep, int NumActive,
                           xbool *Active);
```

FORTRAN

```
SUBROUTINE  XF_READ_ACTIVITY_TIMESTEP(DatasetsId, Timestep, NumActive, Active,
Error)
INTEGER(XID), INTENT(IN)  :: DatasetsId
INTEGER, INTENT(IN)      :: Timestep, NumActive
INTEGER, DIMENSION(NumActive), INTENT(OUT) :: Active
INTEGER, INTENT(OUT)      :: Error
```

The next two functions are used to retrieve values for a particular time-step. The values arrays must be allocated at least as big as the variable **NumVals**.

C/C++

```
int xfReadScalarValuesTimestep(xid DatasetId, int Timestep, int NumVals,
                               float *Values);
int xfReadVectorValuesTimestep(xid DatasetId, int Timestep, int NumVals,
                               int NumComponents, float *Values);
```

FORTRAN

```
SUBROUTINE  XF_READ_SCALAR_VALUES_TIMESTEP(DatasetsId, Timestep, NumVals, Vals,
Error)
INTEGER(XID), INTENT(IN)  :: DatasetsId
INTEGER, INTENT(IN)      :: Timestep, NumVals
REAL, DIMENSION(NumVals), INTENT(OUT) :: Vals
INTEGER, INTENT(OUT)      :: Error

SUBROUTINE  XF_READ_VECTOR_VALUES_TIMESTEP(DatasetsId, Timestep, NumVals,
NumComponents,
                                      Vals, Error)
INTEGER(XID), INTENT(IN)  :: DatasetsId
INTEGER, INTENT(IN)      :: Timestep, NumVals, NumComponents
REAL, DIMENSION(NumVals), INTENT(OUT) :: Vals
INTEGER, INTENT(OUT)      :: Error
```

The next function is used to retrieve values for a data set index for one or more time-steps. The array Values must be a one-dimensional array of size at least as great as NumTimesteps.

C/C++

```
int xfReadScalarValuesAtIndex(xid DatasetId, int Index, int FirstTimestep,
                              int NumTimesteps, float *Values);
```

```
int xfReadVectorValuesAtIndex(xid DatasetId, int Index, int FirstTimestep,
                              int NumTimesteps, int NumComponents,
float *Values);
```

FORTRAN

```
SUBROUTINE  XF_READ_SCALAR_VALUES_AT_INDEX(DatasetsId, Index, FirstTimestep,
                                    NumTimesteps, Values, Error)
INTEGER(XID), INTENT(IN)  :: DatasetsId
INTEGER, INTENT(IN)       :: Index, FirstTimestep, NumTimesteps
REAL, DIMENSION(NumTimesteps), INTENT(OUT) :: Values
INTEGER, INTENT(OUT)      :: Error

SUBROUTINE  XF_READ_VECTOR_VALUES_AT_INDEX(DatasetsId, Index, FirstTimestep,
                                    NumTimesteps, NumComponents,
Values, Error)
INTEGER(XID), INTENT(IN)  :: DatasetsId
INTEGER, INTENT(IN)       :: Index, FirstTimestep, NumTimesteps
REAL, DIMENSION(NumTimesteps, NumComponents), INTENT(OUT) :: Values
INTEGER, INTENT(OUT)      :: Error
```

One can also retrieve activity values at specific locations just like data set values. These functions work just like their data set value counterparts. Remember that activity indices are based upon elements and cells and may not be the same as the locations where the data set values exist.

C/C++

```
int xfReadActivityValuesAtIndex(xid DatasetId, int Index, int
FirstTimestep,
                                int NumTimesteps, xbool
*Activity);
```

FORTRAN

```
SUBROUTINE  XF_READ_ACTIVITY_VALUES_AT_INDEX(DatasetsId, Index,
FirstTimestep,
                                    NumTimesteps, Activity,
Error)
INTEGER(XID), INTENT(IN)  :: DatasetsId
INTEGER, INTENT(IN)       :: Index, FirstTimestep, NumTimesteps
INTEGER, DIMENSION(NumTimesteps), INTENT(OUT) :: Activity
INTEGER, INTENT(OUT)      :: Error
```

4.13.2 Properties

There are a few “reserved” properties when dealing with data sets: Datatype, Datalocation, DatalocationI, DatalocationJ, DatalocationK, and ComponentsIJK. The datatype attribute specifies whether the data set is for scalar data, 2-D vector data, or 3-D vector data, and is stored automatically when the data set is created.

Data sets on grids may be at various locations. The Datalocation attributes are used to indicate where the data are stored for each cell. These include the grid centers, corners, faces, or on faces in a particular direction. Also, a 3-D or Extruded 2-D grid may have a data set for each column rather than for each cell

(for example, CH3D uses a 2-D water surface elevation for each column). For vector data sets, each component may have a different data location. The following functions are used to set the data locations for a data set. The following locations are supported for data sets GRID_LOC_CENTER, GRID_LOC_CORNER, GRID_LOC_FACES, GRID_LOC_FACE_I, GRID_LOC_FACE_J, GRID_LOC_FACE_K, and GRID_LOC_COLUMN.

C/C++

```
int xfScalarDataLocation(xid DatasetId, int DataLocation);
int xfVector2DDataLocations(xid DatasetId, int DataLocationI, int
DataLocationJ);
int xfVector3DDataLocations(xid DatasetId, int DataLocationI, int DataLocationJ,
                            int DataLocationK);
```

FORTRAN

```
SUBROUTINE  XF_SCALAR_DATA_LOCATION(DatasetsId, DataLocation, Error)
INTEGER(XID), INTENT(IN)  :: DatasetsId
INTEGER, INTENT(IN)       :: DataLocation
INTEGER, INTENT(OUT)      :: Error

SUBROUTINE  XF_VECTOR_2D_DATA_LOCATIONS(DatasetsId, DataLocationI, DataLocationJ,
Error)
INTEGER(XID), INTENT(IN)  :: DatasetsId
INTEGER, INTENT(IN)       :: DataLocationI, DataLocationJ
INTEGER, INTENT(OUT)      :: Error

SUBROUTINE  XF_VECTOR_3D_DATA_LOCATIONS(DatasetsId, DataLocationI, DataLocationJ,
                                  DataLocationK, Error)
INTEGER(XID), INTENT(IN)  :: DatasetsId
INTEGER, INTENT(IN)       :: DataLocationI, DataLocationJ, DataLocationK
INTEGER, INTENT(OUT)      :: Error
```

These functions get the data locations for a data set.

C/C++

```
int xfGetScalarDataLocation(xid DatasetId, int *DataLocation);
int xfGetVector2DDataLocations(xid DatasetId, int *DataLocationI,
                               int *DataLocationJ);
int xfGetVector3DDataLocations(xid DatasetId, int *DataLocationI,
                               int *DataLocationJ, int *DataLocationK);
```

FORTRAN

```
SUBROUTINE  XF_GET_SCALAR_DATA_LOCATION(DatasetsId, DataLocation, Error)
INTEGER(XID), INTENT(IN)  :: DatasetsId
INTEGER, INTENT(OUT)      :: DataLocation
INTEGER, INTENT(OUT)      :: Error

SUBROUTINE  XF_GET_VECTOR_2D_DATA_LOCATIONS(DatasetsId, DataLocationI,
DataLocationJ,
                                     Error)
INTEGER(XID), INTENT(IN)  :: DatasetsId
INTEGER, INTENT(OUT)      :: DataLocationI, DataLocationJ
```

```
INTEGER, INTENT(OUT)      :: Error

SUBROUTINE XF_GET_VECTOR_3D_DATA_LOCATIONS(DatasetsId, DataLocationI,
DataLocationJ,
                                          DataLocationK, Error)
INTEGER(XID), INTENT(IN)  :: DatasetsId
INTEGER, INTENT(OUT)      :: DataLocationI, DataLocationJ, DataLocationK
INTEGER, INTENT(OUT)      :: Error
```

Vector data sets on Cartesian grids may have components in local i-, j-, and k-coordinates rather than global x, y, and z. This function should be called for all vector data sets that use local coordinates.

C/C++

```
int xfVectorsInLocalCoords(xid DatasetId);
```

FORTRAN

```
SUBROUTINE  XF_VECTORS_IN_LOCAL_COORDS(DatasetsId, Error)
INTEGER(XID), INTENT(IN)  :: DatasetsId
INTEGER, INTENT(OUT)      :: Error
```

This function is used to determine whether vector data are given in local i-, j-, k-coordinates or the default x-, y-, z-coordinates.

C/C++

```
int xfAreVectorsInLocalCoords(xid DatasetId, int *LocalCoords);
```

FORTRAN

```
SUBROUTINE  XF_ARE_VECTORS_IN_LOCAL_COORDS(DatasetsId, LocalCoords, Error)
INTEGER(XID), INTENT(IN)  :: DatasetsId
LOGICAL, INTENT(OUT)      :: LocalCoords
INTEGER, INTENT(OUT)      :: Error
```

4.14 Coordinate Systems

Coordinate system groups can be associated with mesh, grid, or cross-section groups to define their relative locations.

The values for each of the group variables are listed in Appendix A. Each variable is defined briefly:

- Horizontal datum: The horizontal coordinated system (Geographic, UTM, State Plane…)
- Horizontal units: The units of the horizontal system (ft, m…)

- Vertical datum: The vertical coordinate system (NGVD 29, NGVD 88…)
- Vertical units: The units of the vertical system (ft, m…)
- Latitude: A flag to indicate whether coordinates are entered as north or south latitude.
- Longitude: A flag to indicate whether coordinates are entered as east or west longitude.
- Universal Transverse Mercator (UTM) zone: The UTM zone of the coordinates.
- State Plane Coordinate (SPC) zone: The state plane zone of the coordinates.
- HPGN: The high-precision geodetic network (HPGN) area (Alabama, Arizona, California,…)
- CPP latitude/longitude: Factors used to convert from a latitude/ longitude coordinate system to a CPP (Carte Parallelo-Grammatique Projection) system.
- Ellipse: The ellipsoid for non-North American Datum (NAD)/HPGN coordinate systems.
- MajorR: The major radius for a user-defined ellipsoid.
- MinorR: The minor radius for a user-defined ellipsoid.

If a member of the Coordinate group is not defined, then a default value, listed in Table 6, will be substituted.

Table 6
Default Values for Coordinate Systems

Variable	Type	Default Value
Horizontal datum	int	0
Horizontal units	int	0
Vertical datum	int	0
Vertical units	int	0
Latitude	int	0
Longitude	int	0
UTM zone	int	0
SPC zone	int	0101 (Alabama)
HPGN	int	0
CPP latitude	double	0
CPP longitude	double	0
Ellipse	int	5 (Clark 1866)
MajorR	double	6378206.4 (Clark 1866)
MinorR	double	6356583.8 (Clark 1866)

A list of acceptable values for the Coordinate group can be found in Appendix A. In order to specify a coordinate system, the horizontal datum must be specified. The following must be specified for different coordinate systems.

- Latitude/longitude: Horizontal datum, horizontal units, latitude, longitude
- UTM: Horizontal datum, horizontal units, UTM zone
- State plane: Horizontal datum, horizontal units, SPC zone
- HPGN: Horizontal datum, horizontal units, HPGN area
- CPP: Horizontal datum, horizontal units, CPP latitude, CPP longitude

Ellipse is used with non-NAD geographic and UTM systems as well as user-defined projections. MajorR and MinorR are used with user-defined projections.

The coordinate system information may be stored with each spatial data object within an XMDF file. The following functions are used to create or retrieve the coordinate system group ID for a spatial data object. This group ID is used for the rest of the coordinate system functions to set or get the information for the coordinate system.

C/C++

```
int     xfCreateCoordGroup(xid SpatialId, xid *CoordId);
int     xfOpenCoordGroup(xid SpatialId, xid *CoordId);
```

FORTRAN

```
SUBROUTINE  XF_CREATE_COORD_GROUP(SpatialId, CoordId, Error)
INTEGER(XID), INTENT(IN)  :: SpatialId
INTEGER(XID), INTENT(OUT) :: CoordId
INTEGER, INTENT(OUT)      :: Error

SUBROUTINE  XF_OPEN_COORD_GROUP(SpatialId, CoordId, Error)
INTEGER(XID), INTENT(IN)  :: SpatialId
INTEGER(XID), INTENT(OUT) :: CoordId
INTEGER, INTENT(OUT)      :: Error
```

The API will provide functions to get and set the various parts of the coordinate group.

C/C++

```
int     xfGetHorizDatum(xid CoordId, int *val);
int     xfGetHorizUnits(xid CoordId, int *val);
int     xfGetVertDatum(xid CoordId, int *val);
int     xfGetVertUnits(xid CoordId, int *val);
int     xfGetLat(xid CoordId, int *val);
int     xfGetLon(xid CoordId, int *val);
int     xfGetUTMZone(xid CoordId, int *val);
int     xfGetSPCZone(xid CoordId, int *val);
int     xfGetHPGNArea(xid CoordId, int *val);
int     xfGetCPPLat(xid CoordId, double *val);
int     xfGetCPPLon(xid CoordId, double *val);
int     xfGetEllipse(xid CoordId, int *val);
int     xfGetMajorR(xid CoordId, double *val);
int     xfGetMinorR(xid CoordId, double *val);

int     xfSetHorizDatum(xid CoordId, int val);
int     xfSetHorizUnits(xid CoordId, int val);
int     xfSetVertDatum(xid CoordId, int val);
int     xfSetVertUnits(xid CoordId, int val);
int     xfSetLat(xid CoordId, int val);
int     xfSetLon(xid CoordId, int val);
int     xfSetUTMZone(xid CoordId, int val);
int     xfSetSPCZone(xid CoordId, int val);
int     xfSetHPGNArea(xid CoordId, int val);
int     xfSetCPPLat(xid CoordId, double val);
int     xfSetCPPLon(xid CoordId, double val);
```

```
int     xfSetEllipse(xid CoordId, int val);
int     xfSetMajorR(xid CoordId, double val);
int     xfSetMinorR(xid CoordId, double val);
```

FORTRAN

```
SUBROUTINE  XF_GET_HORIZ_DATUM(CoordId, Val, Error)
INTEGER(XID), INTENT(IN)  :: CoordId
INTEGER, INTENT(OUT)      :: Val
INTEGER, INTENT(OUT)      :: Error

SUBROUTINE  XF_GET_HORIZ_UNITS(CoordId, Val, Error)
INTEGER(XID), INTENT(IN)  :: CoordId
INTEGER, INTENT(OUT)      :: Val
INTEGER, INTENT(OUT)      :: Error

SUBROUTINE  XF_GET_VERT_DATUM(CoordId, Val, Error)
INTEGER(XID), INTENT(IN)  :: CoordId
INTEGER, INTENT(OUT)      :: Val
INTEGER, INTENT(OUT)      :: Error

SUBROUTINE  XF_GET_VERT_UNITS(CoordId, Val, Error)
INTEGER(XID), INTENT(IN)  :: CoordId
INTEGER, INTENT(OUT)      :: Val
INTEGER, INTENT(OUT)      :: Error

SUBROUTINE  XF_GET_LAT(CoordId, Val, Error)
INTEGER(XID), INTENT(IN)  :: CoordId
INTEGER, INTENT(OUT)      :: Val
INTEGER, INTENT(OUT)      :: Error

SUBROUTINE  XF_GET_LON(CoordId, Val, Error)
INTEGER(XID), INTENT(IN)  :: CoordId
INTEGER, INTENT(OUT)      :: Val
INTEGER, INTENT(OUT)      :: Error

SUBROUTINE  XF_GET_UTM_ZONE(CoordId, Val, Error)
INTEGER(XID), INTENT(IN)  :: CoordId
INTEGER, INTENT(OUT)      :: Val
INTEGER, INTENT(OUT)      :: Error

SUBROUTINE  XF_GET_SPC_ZONE(CoordId, Val, Error)
INTEGER(XID), INTENT(IN)  :: CoordId
INTEGER, INTENT(OUT)      :: Val
INTEGER, INTENT(OUT)      :: Error

SUBROUTINE  XF_GET_HPGN_AREA(CoordId, Val, Error)
INTEGER(XID), INTENT(IN)  :: CoordId
INTEGER, INTENT(OUT)      :: Val
INTEGER, INTENT(OUT)      :: Error

SUBROUTINE  XF_GET_CPP_LAT(CoordId, Val, Error)
INTEGER(XID), INTENT(IN)  :: CoordId
REAL(DOUBLE), INTENT(OUT)      :: Val
INTEGER, INTENT(OUT)      :: Error

SUBROUTINE  XF_GET_CPP_LON(CoordId, Val, Error)
INTEGER(XID), INTENT(IN)  :: CoordId
REAL(DOUBLE), INTENT(OUT)      :: Val
INTEGER, INTENT(OUT)      :: Error

SUBROUTINE  XF_GET_ELLIPSE(CoordId, Val, Error)
INTEGER(XID), INTENT(IN)  :: CoordId
INTEGER, INTENT(OUT)      :: Val
INTEGER, INTENT(OUT)      :: Error
```

```
SUBROUTINE  XF_GET_MAJOR_R(CoordId, Val, Error)
INTEGER(XID), INTENT(IN)  :: CoordId
REAL(DOUBLE), INTENT(OUT)      :: Val
INTEGER, INTENT(OUT)      :: Error

SUBROUTINE  XF_GET_MINOR_R(CoordId, Val, Error)
INTEGER(XID), INTENT(IN)  :: CoordId
REAL(DOUBLE), INTENT(OUT)      :: Val
INTEGER, INTENT(OUT)      :: Error

SUBROUTINE  XF_SET_HORIZ_DATUM(CoordId, Val, Error)
INTEGER(XID), INTENT(IN)  :: CoordId
INTEGER, INTENT(IN)       :: Val
INTEGER, INTENT(OUT)      :: Error

SUBROUTINE  XF_SET_HORIZ_UNITS(CoordId, Val, Error)
INTEGER(XID), INTENT(IN)  :: CoordId
INTEGER, INTENT(IN)       :: Val
INTEGER, INTENT(OUT)      :: Error

SUBROUTINE  XF_SET_VERT_DATUM(CoordId, Val, Error)
INTEGER(XID), INTENT(IN)  :: CoordId
INTEGER, INTENT(IN)       :: Val
INTEGER, INTENT(OUT)      :: Error

SUBROUTINE  XF_SET_VERT_UNITS(CoordId, Val, Error)
INTEGER(XID), INTENT(IN)  :: CoordId
INTEGER, INTENT(IN)       :: Val
INTEGER, INTENT(OUT)      :: Error

SUBROUTINE  XF_SET_LAT(CoordId, Val, Error)
INTEGER(XID), INTENT(IN)  :: CoordId
INTEGER, INTENT(IN)       :: Val
INTEGER, INTENT(OUT)      :: Error

SUBROUTINE  XF_SET_LON(CoordId, Val, Error)
INTEGER(XID), INTENT(IN)  :: CoordId
INTEGER, INTENT(IN)       :: Val
INTEGER, INTENT(OUT)      :: Error

SUBROUTINE  XF_SET_UTM_ZONE(CoordId, Val, Error)
INTEGER(XID), INTENT(IN)  :: CoordId
INTEGER, INTENT(IN)       :: Val
INTEGER, INTENT(OUT)      :: Error

SUBROUTINE  XF_SET_SPC_ZONE(CoordId, Val, Error)
INTEGER(XID), INTENT(IN)  :: CoordId
INTEGER, INTENT(IN)       :: Val
INTEGER, INTENT(OUT)      :: Error

SUBROUTINE  XF_SET_HPGN_AREA(CoordId, Val, Error)
INTEGER(XID), INTENT(IN)  :: CoordId
INTEGER, INTENT(IN)       :: Val
INTEGER, INTENT(OUT)      :: Error

SUBROUTINE  XF_SET_CPP_LAT(CoordId, Val, Error)
INTEGER(XID), INTENT(IN)  :: CoordId
REAL(DOUBLE), INTENT(IN)       :: Val
INTEGER, INTENT(OUT)      :: Error

SUBROUTINE  XF_SET_CPP_LON(CoordId, Val, Error)
INTEGER(XID), INTENT(IN)  :: CoordId
REAL(DOUBLE), INTENT(IN)       :: Val
INTEGER, INTENT(OUT)      :: Error

SUBROUTINE  XF_SET_ELLIPSE(CoordId, Val, Error)
INTEGER(XID), INTENT(IN)  :: CoordId
```

```
INTEGER, INTENT(IN)       :: Val
INTEGER, INTENT(OUT)       :: Error

SUBROUTINE  XF_SET_MAJOR_R(CoordId, Val, Error)
INTEGER(XID), INTENT(IN)  :: CoordId
REAL(DOUBLE), INTENT(IN)       :: Val
INTEGER, INTENT(OUT)      :: Error

SUBROUTINE  XF_SET_MINOR_R(CoordId, Val, Error)
INTEGER(XID), INTENT(IN)  :: CoordId
REAL(DOUBLE), INTENT(IN)       :: Val
INTEGER, INTENT(OUT)      :: Error
```

Appendix A
Coord Group

This appendix includes the constants that are used to store coordinate system information inside an XMDF file as discussed in Section 4.14.

Horizontal datum:

```
 0 – Local
 1 – Geographic
 2 – Geographic NAD27 (US)
 3 – Geographic NAD83 (US)
 4 – Geographic HPGN (US)
 5 – UTM
 6 – UTM NAD27 (US)
 7 – UTM NAD83 (US)
 8 – UTM HPGN (US)
 9 – State Plane NAD27 (US)
10 – State Plane NAD83 (US)
11 – State Plane HPGN (US)
12 – CPP (Carte Parallelo-Grammatique Projection)
```

Horizontal units:

```
0 – US survey feet
1 – international feet
2 - meters
```

Vertical datum:

```
0 – Local
1 – NGVD 29
2 – NGVD 88
```

Vertical units:

```
0 – US survey feet
1 – international feet
2 - meters
```

Latitude:

```
0 – North
1 - South
```

Longitude

```
0 – East
1 – West
```

UTM zone:

```
Value from 1 to 60
```

SPC Zone:
NAD27 only
NAD83 only
Both

Alabama East - 0101
Alabama West - 0102
Arizona East - 0201
Arizona Central - 0202
Arizona West - 0203
Arkansas North - 0301
Arkansas South - 0302
California 1 – 0401
California 2 – 0402
California 3 – 0403
California 4 – 0404
California 5 – 0405
California 6 - 0406
California 7 - 0407
Colorado North - 0501
Colorado Central - 0502
Colorado South - 0503
Connecticut - 0600
Delaware - 0700
District of Columbia - 1900
Florida East - 0901
Florida West - 0902
Florida North - 0903
Georgia East - 1001
Georgia West - 1002
Idaho East - 1101
Idaho Central - 1102
Idaho West - 1103
Illinois East - 1201
Illinois West - 1202
Indiana East - 1301
Indiana West - 1302
Iowa North - 1401
Iowa South - 1402
Kansas North - 1501
Kansas South - 1502
Kentucky North - 1601
Kentucky South - 1602
Louisiana North - 1701
Louisiana South - 1702
Louisiana Offshore - 1703
Maine East - 1801
Maine West - 1802
Maryland - 1900
Massachusetts Mainland - 2001
Massachusetts Island – 2002
Michigan East - 2101
Michigan Central - 2102
Michigan West - 2103
Michigan North - 2111
Michigan Central - 2112
Michigan South - 2113
Minnesota North - 2201
Minnesota Central - 2202
Minnesota South - 2203
Mississippi East - 2301
Mississippi West - 2302
Missouri East - 2401
Missouri Central - 2402
Missouri West - 2403
Montana - 2500
Montana North - 2501
Montana Central - 2502
Montana South - 2503
Nebraska - 2600
Nebraska North - 2601
Nebraska South - 2602
Nevada East - 2701
Nevada Central - 2702
Nevada West - 2703
New Hampshire - 2800
New Jersey - 2900
New Mexico East - 3001
New Mexico Central - 3002
New Mexico West - 3003
New York East - 3101
New York Central - 3102
New York West - 3103
New York Long Island - 3104
North Carolina - 3200
North Dakota North - 3301
North Dakota South - 3302
Ohio North - 3401
Ohio South - 3402
Oklahoma North - 3501
Oklahoma South - 3502
Oregon North - 3601
Oregon South - 3602
Pennsylvania North - 3701
Pennsylvania South - 3702
Rhode Island – 3800
South Carolina - 3900
South Carolina North - 3901
South Carolina South - 3902
South Dakota North - 4001
South Dakota South - 4002
Tennessee - 4100
Texas North - 4201
Texas North Central - 4202
Texas Central - 4203
Texas South Central - 4204
Texas South - 4205
Utah North - 4301
Utah Central - 4302
Utah South - 4303
Vermont - 4400
Virginia North - 4501
Virginia South - 4502
Washington North - 4601
Washington South - 4602
West Virginia North - 4701
West Virginia South - 4702
Wisconsin North - 4801
Wisconsin Central - 4802
Wisconsin South - 4803
Wyoming I - 4901
Wyoming II - 4902
Wyoming III - 4903
Wyoming IV - 4904
Wyoming East - 4901
Wyoming East Central - 4902
Wyoming West Central - 4903
Wyoming West - 4904
Alaska 1 - 5001
Alaska 2 - 5002
Alaska 3 - 5003
Alaska 4 - 5004
Alaska 5 - 5005
Alaska 6 - 5006
Alaska 7 - 5007
Alaska 8 - 5008
Alaska 9 - 5009
Alaska 10 - 5010
Hawaii 1 - 5101
Hawaii 2 - 5102
Hawaii 3 - 5103
Hawaii 4 - 5104
Hawaii 5 - 5105
Puerto Rico/Virgin Islands - 5200
Puerto Rico/Virgin Islands - 5201
St. Croix – 5202
American Samoa - 5300
Guam Island – 5400

HPGN:

0 - Alabama
1 - Arizona
2 - California (Northern)
3 - California (Southern)
4 - Colorado
5 - Florida
6 - Georgia
7 - Hawaii
8 - Idaho/Montana (Eastern)
9 - Idaho/Montana (Western)
10 - Kansas
11 - Kentucky
12 - Louisiana
13 - Maine
14 - Maryland/Delaware
15 - Michigan
16 - Mississippi
17 - Nebraska
18 - New England (CT,MA,NH,VT)
19 - New Mexico
20 - New York
21 - North Dakota
22 - Ohio
23 - Oklahoma
24 - Puerto Rico/Virgin Islands
25 - South Dakota
26 - Tennessee
27 - Texas (East)
28 - Texas (West)
29 - Utah
30 - Virginia
31 - Washington/Oregon
32 - West Virginia
33 - Wisconsin
34 - Wyoming

CPP Latitude, CPP Longitude:

Carte Parallelo Grammatique Projection
Factor for converting the latitude/longitude

Ellipse:

0 - Airy 1830
1 - Airy Modified 1849
2 - Australian National 1965
3 - Bessel 1841
4 - Bessel (Namibia) 1841
5 - Clarke 1866
6 - Clarke 1880
7 - Everest 1830
8 - Everest (India) 1956
9 - Everest (Malaysia) 1969
10 - Everest (Malay & Singapr) 1948
11 - Everest (Pakistan
12 - Everest (Sabah & Sarawak)
13 - Everest Modified 1830
14 - Fischer Modified 1960
15 - GRS 1980
16 - Hayford 1909
17 - Helmert 1906
18 - Hough 1909
19 - Indonesian National 1974
20 - International 1909
21 - International 1924
22 - International 1967
23 - Krassovsky 1940
24 - Mercury 1960
25 - Mercury Modified 1968
26 - Southeast Asia
27 - South American 1969
28 - WGS 1960
29 - WGS 1966
30 - WGS 1972
31 - WGS 1984
32 - User defined

MajorR:

User-defined ellipse major radius

MinorR:

User-defined ellipse minor radius

Appendix B
API Types and Functions

This appendix summarizes all the types and functions defined for the XMDF API.

Type	Description
`xid`	An identifier. Can point to a file or a group. Equivalent to HDF5's hid_t type.

Function	Description
Version Information	
`xfGetLibraryVersion`	Get the XMDF library version for linked API
`xfGetLibraryVersionFile`	Get the XMDF library version that wrote a file
Creating and Opening Files	
`xfCreateFile`	Create and open an XMDF file
`xfOpenFile`	Open an existing XMDF file
`xfCloseFile`	Close and open XMDF file
Groups	
`xfCreateGroupForMesh`	Create a group to store a mesh
`xfCreateGroupForGrid`	Create a group to store a grid
`xfCreateGroupForXsecs`	Create a group to store a set of cross sections
`xfOpenGroup`	Open a group (returns the group path)
`xfCloseGroup`	Close a group
Browsing Entities	
`xfGetGroupPathsSizeForMeshes`	Get the number of mesh groups in a file
`xfGetAllGroupPathsForMeshes`	Get the paths to mesh groups in a file
`xfGetGroupPathsSizeForGrids`	Get the number of grid groups in a file
`xfGetAllGroupPathsForGrids`	Get the paths to grid groups in a file
`xfGetGroupPathsSizeForXsecs`	Get the number of cross-section groups in a file
`xfGetAllGroupPathsForXsecs`	Get the paths to cross-section groups in a file

Function	Description
Properties	
xfWritePropertyInt	Create an integer property with a specified name in a group
xfWritePropertyFloat	Create a float property with a specified name in a group
xfWritePropertyDouble	Create a double property with a specified name in a group
xfWritePropertyString	Create a string property with a specified name in a group
xfDoesPropertyeWithNameExist	Find out if a property with a specified name exists
xfGetNumberOfProperties	Get the number of properties defined for a group
xfGetPropertyNames	Get the names of all the properties for a group
xfGetPropertyType	Get the type of a property
xfGetPropertyStringLength	Get the length of the longest string in a property
xfGetPropertyNumber	Get the number of entries in a property array
xfGetPropertyInt	Get integer property value(s)
xfGetPropertyFloat	Get float property value(s)
xfGetPropertyDouble	Get double property value(s)
xfGetPropertyString	Get String property value(s)
Meshes	
xfGetNumberOfElements	Get the number of elements in a mesh group
xfAreAllElemsSameType	Determine whether all elements in a mesh are the same type
xfGetElemTypeSingleValue	Get the type of all elements if they are the same type
xfGetElemTypes	Get an array of element types
xfGetMaxNodesInElem	Get the maximum number of nodes in any element
xfGetElemNodeIds	Get the element connectivity arrays
xfGetNumberOfNodes	Get the number of nodes in a mesh
xfGetXNodeLocations	Get an array of node X locations
xfGetYNodeLocations	Get an array of node Y locations
xfGetZNodeLocations	Get an array of node Z locations
xfSetNumberOfElements	Store the number of elements in a mesh
xfSetAllElemsSameType	Store the type for all elements in a mesh
xfSetElemTypes	Store the types for all elements in a mesh
xfSetElemNodeIds	Store the element connectivity arrays
xfSetNumberOfNodes	Store the number of nodes in a mesh
xfSetXNodeLocations	Store the X locations of the nodes

Function	Description
xfSetYNodeLocations	Store the Y locations of the nodes
xfSetZNodeLocations	Store the Z locations of the nodes
xfGetMeshPropertyGroup	Get the group ID for the mesh property group
xfGetElementPropertyGroup	Get the group ID for the mesh element property group
xfGetNodePropertyGroup	Get the group ID for the mesh node property group
Grids	
xfSetGridType	Set the grid type
xfSetExtusionTYpe	Set up a grid to use some type of extrusion
xfSetNumberOfDimensions	Set the number of dimensions for a grid (2 or 3)
xfSetOrigin	Set the grid origin
xfSetBearing	Set the angle to rotate about the z-axis
xfSetDip	Set the angle to rotate about the x-axis
xfSetComputationalOrigin	Set the grid computational origin
xfSetUDirection	Set the u-direction
xfSetNumberOfCellsInI	Set the number of cells along the i-axis of the grid
xfSetNumberOfCellsInJ	Set the number of cells along the j-axis of the grid
xfSetNumberOfCellsInK	Set the number of cells along the k-axis of the grid
xfGetGridType	Set the grid type
xfGetExtusionTYpe	Set up a grid to use some type of extrusion
xfGetNumberOfDimensions	Set the number of dimensions for a grid (2 or 3)
xfGetOrigin	Set the grid origin
xfGetBearing	Set the angle to rotate about the z-axis
xfGetDip	Get the angle to rotate about the x-axis
xfGetComputationalOrigin	Get the grid computational origin
xfGetUDirection	Get the u-direction
xfGetNumberOfCellsInI	Get the number of cells along the i-axis of the grid
xfGetNumberOfCellsInJ	Get the number of cells along the j-axis of the grid
xfGetNumberOfCellsInK	Get the number of cells along the k-axis of the grid
xfSetGridCoordsI	Set the grid geometry information for i-axis

Function	Description
xfSetGridCoordsJ	Set the grid geometry information for j-axis
xfSetGridCoordsK	Set the grid geometry information for k-axis
xfGetGridCoordsI	Get the grid geometry information for i-axis
xfGetGridCoordsJ	Get the grid geometry information for j-axis
xfGetGridCoordsK	Get the grid geometry information for k-axis
xfWriteExtrudeLayerData	Set the layer extrusion data for a 2-D extruded grid
xfGetExtrudeNumLayers	Get the number of extruded layers for an extruded grid
xfGetExtrudeValues	Get the extruded grid values; the meaning depends upon the type of grid
xfGetGridPropertyGroup	Get the property group associated with a grid as a whole
xfGetGridCellPropertyGroup	Get the property group associated with the cells of a grid
xfGetGridNodePropertyGroup	Get the property group with properties that are associated with the nodes (corners) of a grid.
Data Sets	
xfCreateMultiDatasetsGroup	Create a folder to contain data sets for a specific spatial data object
xfGetGroupPathsSizeForMultiDatasets	Find the paths to all multi-data-set folders within a file or group
xfGetAllGroupPathsForMultiDatasets	Read the paths for all the multi-data set folders within a file or group
xfGetDatasetsSdoGuid	Get the GUID for the associated spatial data object for a multi-data set folder
xfOpenMultiDatasetsGroup	Open the multi-data sets group associated with a mesh or grid
xfSetupToWriteDatasets	Shortcut function to set everything up to start writing data sets
xfCreateScalarDataset	Create a data set to write scalar values to
xfCreateVectorDataset	Create a data set to write vector values to
xfCalendarToJulian	Convert a calendar date to a Julian date
xfJulianToCalendar	Convert a Julian date to a calendar date
xfDatasetReftime	Set the reference time for a data set. Time-steps are offsets from this time

Function	Description
xfWriteScalarTimestep	Write a scalar time-step data to a data set
xfWriteVectorTimestep	Write a vector time-step data to a data set
xfWriteActivityTimestep	Write the activity data for a time-step
xfGetDatasetGroupId	Open the data sets folder for a mesh or grid
xfGetScalarDatasetsInfo	Determine the number and maximum path length for scalar data sets below a group
xfGetScalarDatasetPaths	Get the paths to all scalar data sets below a group
xfGetVectorDatasetsInfo	Determine the number and maximum path length for vector data sets below a group
xfGetVectorDatasetPaths	Get the paths to all vector data sets below a group
xfGetDatasetUnits	Get the units used in a data set
xfGetDatasetReftime	Get the reference time for a data set
xfGetDatasetNumTimes	Get the number of times in a data set
xfGetDatasetTimeUnits	Get the time units for a data set
xfGetDatasetTimes	Get the times for a data set
xfGetDatasetMins	Get the minimum values for each time-step
xfGetDatasetMaxs	Get the maximum values for each time-step
xfGetDatasetNumVals	Get the number of values in a data set
xfGetDatasetNumActiveVals	Get the number of active values (elements or cells) associated with a data set
xfReadActivityTimestep	Read the activity values for a specific time-step
xfReadScalarValuesTimestep	Read the scalar values for a specific time-step
xfReadVectorValuesTimestep	Read the vector values for a specific time-step
xfReadActivityValuesAtIndex	Read the activity values for a specific index for one or more time-steps
xfReadScalarValuesAtIndex	Read the scalar values for a specific index for one or more time-steps
xfReadVectorValuesAtIndex	Read the vector values for a specific index for one or more time-steps
xfScalarDataLocation	Define the location that a scalar data set is assigned to (Grid centers, corners, etc.)
xfVector2DDataLocations	Define the locations for each component of a 2-D vector data set

Function	Description
xfVector3DDataLocations	Define the locations for each component of a 3-D vector data set
xfGetScalarDataLocation	Get the location that a scalar data set is assigned to (Grid centers, corners, etc.)
xfGetVector2DDataLocations	Get the locations for each component of a 2-D vector data set
xfGetvector3DDataLocations	Get the locations for each component of a 3-D vector data set
xfVectorsInLocalCoords	Identifies that a vector data set is in local grid coordinates (Cartesian grids only)
Coordinate Systems	
xfGetHorizDatum	Horizontal coordinate system (Geographic, UTM, …)
xfGetHorizUnits	Horizontal units (ft, m)
xfGetVertDatum	Vertical datum (NGVD 29, NGVD 88, …)
xfGetVertUnits	Vertical units (ft, m)
xfGetLat	Indicate whether coordinates are north or south latitude
xfGetLon	Indicate whether coordinates are east or west longitude
xfGetUTMZone	UTM zone
xfGetSPCZone	State plane zone
xfGetHPGNArea	High precision geodetic network area (Alabama, Arizona, California, etc.)
xfGetCPPLat	Factors to convert from a latitude/longitude system to a Carte Parallelo-Grammatique Projection system
xfGetCPPLon	Factors to convert from a latitude/longitude system to a Carte Parallelo-Grammatique Projection system
xfGetEllipse	Ellipsoid for non-NAD/HPGN coordinate systems
xfGetMajorR	Major radius for a user-defined ellipsoid
xfGetMinorR	Minor radius for a user-defined ellipsoid
xfSetHorizDatum	Horizontal coordinate system (Geographic, UTM, …)
xfSetHorizUnits	Horizontal units (ft, m)
xfSetVertDatum	Vertical datum (NGVD 29, NGVD 88, ..)
xfSetVertUnits	Vertical units (ft, m)

Function	Description
xfSetLat	Indicate whether coordinates are north or south latitude
xfSetLon	Indicate whether coordinates are east or west longitude
xfSetUTMZone	UTM zone
xfSetSPCZone	State plane zone
xfSetHPGNArea	High precision geodetic network area (Alabama, Arizona, California, etc.)
xfSetCPPLat	Factors to convert from a latitude/longitude system to a Carte Parallelo-Grammatique Projection system
xfSetCPPLon	Factors to convert from a latitude/longitude system to a Carte Parallelo-Grammatique Projection system
xfSetEllipse	Ellipsoid for non-NAD/HPGN coordinate systems
xfSetMajorR	Major radius for a user-defined ellipsoid
xfSetMinorR	Minor radius for a user-defined ellipsoid

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	**2. REPORT TYPE**	**3. DATES COVERED** *(From - To)*
January 2007	Final report	

4. TITLE AND SUBTITLE

eXtensible Model Data Format (XMDF)

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Cary D. Butler, David R. Richards, Robert M. Wallace, Norman L. Jones, Russell Jones

5d. PROJECT NUMBER
122401 and 122425

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

U.S. Army Engineer Research and Development Center
Information Technology Laboratory and Coastal and Hydraulics Laboratory
3909 Halls Ferry Road, Vicksburg, MS 39180-6199;
Environmental Modeling Research Laboratory, 242 Clyde Building
Brigham Young University, Provo, Utah 84602

8. PERFORMING ORGANIZATION REPORT NUMBER

ERDC SR-07-1

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

U.S. Army Corps of Engineers
Washington, DC 20314-1000

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

The U.S. Army Engineer Research and Development Center, in conjunction with the Environmental Modeling Research Laboratory (EMRL) at Brigham Young University (BYU), is developing an efficient Application Programming Interface (API) for handling multi-dimensional data produced for water resource computational modeling. This API, in conjunction with a corresponding data standard, is being implemented within ERDC computational models to facilitate rapid data access, enhanced data compression and data sharing, and cross-platform independence. The API and data standard are known as the eXtensible Model Data Format (XMDF), and version 1.0 is available for public use and free dissemination. This report presents the purpose and architecture of the XMDF API and data format.

15. SUBJECT TERMS

Databases
Finite difference method
Finite element method
Three-dimensional methods
Two-dimensional methods

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			**19b. TELEPHONE NUMBER** *(include area code)*
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED		98	