I/O and Meta-Data Storage Conventions in MOAB

Timothy J. Tautges

INTRODUCTION

The Mesh-Oriented datABase (MOAB) is a library for representing finite element and other types of mesh data [1]. Various types of meta-data are often used in conjunction with a mesh. Examples include boundary condition groupings, material types, and provenance information for the mesh. Because the data model used in MOAB is so abstract, conventions are useful for describing how meta-data is stored into that data model. This document describes those conventions for several types of data commonly found in meshes stored in MOAB. Because the data models used by MOAB and iMesh, the ITAPS mesh interface [2], are so similar, the conventions described here apply almost unmodified to iMesh as well as to MOAB.

The meshes represented in MOAB originate in a variety of forms, including mesh read from files of various formats (e.g. CUBIT ".cub" file, VTK, etc.) as well as mesh written into MOAB directly by various software libraries (e.g. MeshKit). Although there is no standard for naming or storing meta-data with a mesh, there is a great deal of commonality in the types of meta-data typically found with mesh data. This document describes conventions that have been established for commonly encountered meta-data. Various mesh readers implemented in MOAB attempt to read meta-data from a file and write it into the MOAB data model using these conventions. Although there is no requirement to store a given type of meta-data in the form described here, a number of services have been written to handle meta-data using these conventions, no matter the source of the meta-data being processed.

Several specific tools are often used in concert with MOAB and bear special mention here. The CUBIT toolkit generates finite element meshes, and saves them to a native save file (referred to as a ".cub" file) which MOAB is able to read. Reading CUBIT meshes into MOAB through the .cub file format is preferred over other formats, since most other mesh formats written by CUBIT do not save most meta-data. The MeshKit library also generates mesh using CGM and MOAB, and uses the same conventions for storing meshes into MOAB. Finally, MOAB includes a CGM reader which can read a geometric model into a faceted representation in MOAB. Meta-data from all these tools are stored in MOAB using the conventions described here.

The MOAB data model consists of the following basic types:

- **Entity:** The basic elements of topology, e.g. vertex, edge, triangle, tetrahedron, etc. MOAB represents all types in the finite element zoo, plus polygons and polyhedra.
- **Entity Set:** An arbitrary collection of entities and other sets. Sets can have parent/child relations with other sets, and these relations are distinct from "contains" relations.
- **Interface:** The interface object through which other entities are accessed, in the sense of object-oriented-programming. iMesh refers to the interface as the "root" set.
- **Tag:** A piece of data that can be assigned a distinct value to each entity and entity set, and to the interface itself. Tags have a prescribed name, size in bytes, and data type; allowed data types are integer, double, entity handle, and byte or opaque.

The following section describes each meta-data tag convention in detail; these conventions are also summarized in Table 1.

Meta-Data Conventions

Meta-data is stored in MOAB and iMesh in the form of tags applied to either entities or entity sets. For meta-data represented as entity sets, the contents of those sets are determined by the convention, with tags on those sets identifying them with the convention and adding any other semantic data.

Each meta-data convention is described in a subsection below. Each convention begins with a short description of:

- Whether tags associated with the convention are assigned to entities or entity sets
- The tag(s) associated with the convention; information for each tag includes the name, the data type (I=integer, D=double, C=character, H=handle), and the tag length. Tag lengths are specified after an asterisk (*); for example, C*32 implies a tag with character type and length 32. Unspecified lengths correspond to length one.

Name

(Data: Entity sets, entities; Tag(s): NAME/C*32)

Character strings are used in many different contexts in applications. MOAB uses the "NAME" tag to store character strings used to name entities. This tag is of byte-type and is of length 32 bytes. *Note that the string stored in this tag may or may not be terminated with a NULL character. It is always prudent account for missing NULL terminator, to avoid buffer overflow errors in the application.* Applications are free to define their own version of the NAME tag with a longer length, though this definition may conflict with other services attempting to use this tag with the conventional size. Applications needing a string tag with a longer or variable length can also use MOAB's variable-length tag type, though this will not be compatible with iMesh.

Title

(Data: Entity sets (file or instance); Tag(s): TITLE/C*strlen)

The title tag is meant to hold the overall identifier of a mesh, written at generation time or read from a file generated with a non-MOAB tool. The tag length is variable, and is set by the application directly (by calling the tag_create function) or indirectly (by embedding the title in a file read by MOAB).

Global Identifier

(Data: Entity sets, entities; Tag(s): GLOBAL_ID/I)

Global identifiers are used in many different contexts in applications. Geometric model entities are identified by dimension and id, e.g. "Volume 1". Mesh vertices and elements are identified similarly in mesh generation codes. Boundary conditions and material types are identified similarly. This tag is used to store such information. This tag is currently stored in a 32-byte integer, though this may change in the future.

Geometric Model Information

(Data: Entity sets; Tag(s): GEOM_DIMENSION/I, GLOBAL_ID/I, NAME/C*32, CATEGORY/C*32, GEOM_SENSE_2(EH[2]), GEOM_SENSE_N_ENTS(EH*N), GEOM_SENSE_N_SENSES(I*N))

Mesh generation is often performed starting from a geometric model, represented in some form of CAD engine. Many of the meshes used by MOAB are generated based on the CGM library.

Geometric models contain both topological information (the topological entities in the geometric model) and shape information (the geometric shape of those entities), as well as other meta-data written to the entities in a model. When a mesh is read from a CUBIT .cub file, meta-data from the geometric model is read and represented in the MOAB data model, as described below. **Note that although MOAB reads and represents meta-data associated with the geometric model, it does not represent the geometric model itself.** Therefore, shape-related information, e.g. the arc length of an edge or surface normal at a given point, can be retrieved only from the model represented in CGM or another geometric modeling engine.

The information contained in a geometric model, read into and represented in MOAB, consists of:

- Model entities (vertex, edge, face, volume)
- Topological relationships between model entities
- Groups of model entities
- Model entity/group ids
- Model entity/group names

The storage of this information into MOAB's data model is described for each type is described below.

Entities

Entities in the geometric model (VERTEX, EDGE, FACE, VOLUME) are each represented by an entity set[1]. These sets are tagged with the "GEOM_DIMENSION" tag, with integer value equal to the topological dimension of the entity (VERTEX = 0, EDGE = 1, etc.) These sets *contain* the mesh *owned* by the corresponding entity in the geometric model. Note this does not include mesh owned by bounding entities; thus, the set for a FACE will not contain the mesh vertices owned by bounding EDGEs in the geometric model. These sets may or may not contain mesh entities of intermediate dimension, e.g. mesh edges owned by a FACE or faces owned by a VOLUME, depending on the application generating the mesh or the file from which the mesh was read. These sets are all set-types, i.e. the order of entities in the sets is not significant, except in the case of EDGE sets, where order of the mesh vertices and edges corresponds to the relative order of vertices and edges at the time of mesh generation. In MOAB, these sets are non-tracking by default, i.e. entities do not have knowledge of which geometry sets they are members of.

Topological Relationships

In the geometric model, each FACE is bounded by zero or more EDGEs; other topological relationships between geometric entities exist in a similar manner. These relationships are embedded in the data model using parent/child relations between entity sets. For example, the entity set corresponding to a FACE will have child sets, each corresponding to a bounding EDGE, and parent sets, each corresponding to a VOLUME bounded by that FACE. The relative order of sets in those parent/child lists is not significant, thus, "loops" bounding a FACE cannot reliably be inferred from this data.

Groups

Geometric entities are sometimes assigned to application-specific groups. These groups are represented using entity sets, tagged with a "GROUP" tag whose value equals the group id. Group sets are "set"-type, and are not tracking sets. These sets *contain* the sets corresponding to geometric entities contained in the groups in the geometric model, as well as any mesh entities assigned to the group.

Sense

1 Body-type entities from CUBIT are not explicitly represented in MOAB.

A geometric face has a natural orientation, indicated by the direction of the normal to the face; similarly, edges have a natural orientation determined by the direction of the tangent. When faces bound regions, or edges bound faces, they do so with a sense; if a region includes a face with forward sense, that means the face's natural normal direction points out of the volume. If a face includes an edge with forward sense, that means that if one moves along the edge in the direction of its tangent, the material of the face is on the left hand side. The sense of a face (edge) with respect to a region (face) it bounds is stored using tags on the face (edge).

Most models allow a face to be part of only two regions. Therefore, to store the sense of a face with respect to regions including it, a tag with two values is used. This tag is named GEOM_SENSE_2, and has 2 EntityHandle values. The first value corresponds to the entity set for the region for which that face has a forward sense, and the second to the region for which that face has a reverse sense.

Edges can bound more than two faces. Therefore, two variable-length tags are used, one to store the EntityHandles of the faces the edge bounds, and the other to store the sense with which the edge bounds the corresponding face. These tags are named GEOM_SENSE_N_ENTS and GEOM_SENSE_N_SENSES, respectively. These are stored as variable-length tags; see the MOAB user's guide for information on how to work with tags of this type.

The following sense values are used:

- 0: forward
- 1: reverse
- -1: unnknown

Material Type

(Data: Entity sets; Tag(s): MATERIAL_SET/I)

Most finite element and other PDE-based analysis codes require a material type for each cell or element in the simulation. MOAB uses entity sets to store this information, in the form of entity sets. The MATERIAL_SET tag is used to identify these sets. The value of this tag is conventionally an integer; in most cases this stores a user-assigned identifier associated with that material.

CUBIT assigns material types using what it calls “element blocks”, with each element block given a user-assigned id number and optionally a name. The CUBIT and Exodus file readers in MOAB read element blocks into MATERIAL_SET sets.

In CUBIT, materials are typically assigned by assigning geometric volumes to element blocks. Therefore, *material sets often <u>contain</u> entity sets corresponding to those volumes.* Thus, a materrial set in MOAB is unlikely to contain mesh entities directly; rather, that set contains other sets which contain mesh entities. In these cases, mesh entities can be retrieved by passing a “recursive” flag to the appropriate function (MOAB), or by calling the getEntitiesRec extension function (iMesh) provided by MOAB.

Boundary Conditions (Dirichlet, Neumann)

(Data: Entity sets; Tag(s): DIRICHLET_SET/I, NEUMANN_SET/I)

Boundary conditions are often specified in terms of geometric model entities, similar to material types.

MOAB uses entity sets to store this information as well. The DIRICHLET_SET and NEUMANN_SET tags are used to represent Dirichlet- and Neumann-type boundary condition sets, resp. By convention, Neumann sets usually contain (indirectly) intermediate-dimension entities like edges in a 2D mesh or faces in a 3D mesh, while Dirichlet sets usually contain vertices. In addition, Neumann sets are represented as sets of faces, rather than as sides of elements. Faces can be ordered "forward" or "reverse" with respect to one of the bounding elements, depending on whether the right-hand normal points into or out of the element. Forward-sense faces are added to the Neumann set. Reverse-sense faces are put into a separate set; that set is tagged with the NEUSET_SENSE tag, with value = -1; and that reverse set is added to the Neummann set.

Parallel Mesh Constructs

(Data: Entity sets, entities; Tag(s): PARALLEL_PART/I, PARALLEL_PARTITION/I, PSTATUS/C*1, PARALLEL_SHARED_PROC/I, PARALLEL/SHARED_HANDLE/H, PARALLEL_SHARED_PROCS/I*NP, PARALLEL_SHARED_HANDLES/H*NP)

On a parallel computer, MOAB can represent the mesh on each processor as well as information about entities shared with neighboring processors. Some of this information is also relevant even when the mesh is represented on a serial machine. MOAB uses several tag and set conventions to describe the parallel nature of a mesh. This information is summarized here; for a more complete description of MOAB's parallel mesh representation and functionality, see [ref-moabpar].

Parallel partition, parts

Most parallel mesh applications use a domain decomposition approach, where each processor solves for a subset of the domain. The set of entities solved by a given processor is referred to as a part, and the collection of parts together is called the partition. MOAB stores each part in an entity set, marked with the PARALLEL_PART tag, whose value is the rank of the processor assigned that part; an entity set which contains all part sets is given the PARALLEL_PARTITION tag, whose value is currently meaningless. The MBZoltan tool included as a tool in MOAB can partition a mesh for parallel solution, and writes the partition to the mesh in the form of parts and partitions. Both these types of sets can be accessed in a serial mesh, e.g. for visualization.

Part interfaces

When a partitioned mesh has been loaded on a parallel computer, the part on a given processor may share portions of its boundary with parts on other processors. These shared regions are called part interfaces, and are also represented using entity sets. These sets are marked with the PARALLEL_INTERFACE tag, whose value is currently meaningless.

Shared processor and handle

For entities shared between processors, it is helpful to know locally which other processor shares an entity, and what the entity's handle is on the remote processor. There are two cases which are useful to distinguish, first where an entity is shared with only one other processor (referred to as shared), and second when a processor is shared by more than one other processor (referred to as multi-shared). Shared entities are given the PARALLEL_SHARED_PROC and PARALLEL_SHARED_HANDLE tags, which store the rank of the sharing processor and the handle of the entity on that processor, respectively. Multi-shared entities are marked with the PARALLEL_SHARED_PROCS and PARALLEL_SHARED_HANDLES tags; these tags have a length NP assigned at compile time in MOAB, with default values of -1 for processor rank and zero for handle (which are each invalid values for the corresponding data). The processors/handles sharing a given entity are then written on the front

of the arrays. So, for example, an entity on processor rank 0, shared by processors 1 and 2, would have a PARALLEL_SHARED_PROCS tag whose values would be [1, 2, -1, -1, …], with PARALLEL_SHARED_HANDLES values of [m, n, 0, 0, …], where m and n would be the handles of that entity on processors 1 and 2. The shared versions of these tags are "dense", with default values which denote unshared entities. The multi-shared tags are sparse tags in MOAB, with no default value.

Parallel status

In addition to the tags above, MOAB also defines the PSTATUS tag, whose bits contain information about the parallel status of a given entity. Starting with least significant bit, these bits represent whether an entity is 1) not owned, 2) shared, 3) multi-shared, 4) interface, 5) a ghost entity. The first bit being set indicates "not owned" so that the default value for this tag, of zero, corresponds to an owned, unshared entity, which will be the state of most entities on a given processor.

Structured Mesh Parameters

MOAB has a structured mesh interface for creating structured mesh (see "ScdInterface.hpp" header file in MOAB source code). Along with an internal representation that is more memory-efficient (since it does not need to store connectivity), MOAB also creates and tags entity sets with structured mesh parameters, which can be accessed through the normal tag and set interface. The following tags are used:

- **BOX_DIMS:** This tag stores the ijk coordinates of the lower and upper corner of the structured mesh box(es).
- **GLOBAL_BOX_DIMS:** If specified when the structured mesh is created, a tag with this name stores the global box dimensions (which may be different than the local box dimensions).
- **BOX_PERIODIC:** Stores whether the box is periodic in the i (BOX_PERIODIC[0]) and j (BOX_PERIODIC[1]) directions.
- **__BOX_SET:** Pointer to the ScdBox instance corresponding to this entity set.[2]

Although the structured mesh is not saved as such in HDF5-format files, the entity sets and corresponding tags will be saved and restored.

Spectral Mesh Constructs

The Spectral Element Method (SEM) is a high-order method, using a polynomial Legendre interpolation basis with Gauss-Lobatto quadrature points, in contrast to the Lagrange basis used in (linear) finite elements. A spectral mesh with order O contains quadrilateral or hexahedral elements comprised of $(O+1)^d$ vertices. Spectral meshes are usually represented in one of two ways, either as coarse elements which point to an array of higher-order vertices (and with corner vertices represented in the normal manner), or as linear quads/hexes formed from the higher-order vertices, with each original coarse quad/hex represented by O^d fine quads/hexes. Similarly, the spectral variables, which are normally computed at fine vertex positions, are stored either on those vertices, or in lexicographically-ordered arrays on elements (with tag values repeated on neighboring elements). MOAB can read spectral meshes from a variety of formats (at this time, including CAM-SE, HOMME, and Nek5000). Which of the above two representations are controlled by read options and are indicated by certain tags:

- SPECTRAL_MESH: read option indicating that spectral elements should be represented as coarse linear quads/hexes and each element containing an array of lexicographically-ordered vertex handles
- TAG_SPECTRAL_ELEMENTS: read option; if given, spectral variables are represented as lexicographically-ordered arrays on elements

2 The double-underscore in the tag name implies that this tag will not be saved in a file, in this case because the ScdBox instances are not preserved in a file.

- TAG_SPECTRAL_VERTICES: read option; if given, spectral variables are represented as tags on vertices
- CONN=<filename>: in CAM-SE, the connectivity of the spectral mesh is stored by default in a file named “HommeMapping.nc”; this option can be given to read the connectivity from a different file
- SPECTRAL_VERTICES: tag name for array of vertex handles
- SPECTRAL_ORDER: tag name for spectral order, written to file set or (if no file set given) to interface after a spectral mesh is read

Reader/Writer Options

All mesh file readers and writers in MOAB take an option string as an argument. By default, the semicolon (“;”) delimits individual options in the option string. Options used in multiple readers are described in this section; the options enabled in specific readers/writers are described in the corresponding appendix at the end of this document.

variable=<var_name>[,...]

By default, all field data stored with the mesh is read with the mesh, and stored as tags on the associated mesh entities. This option lists specific variables that should be read along with the mesh (note also the “nomesh” option, described elsewhere in this document). The variable name listed will be read into a tag with the same name. For time-dependent variables, the time step number will be appended to the variable name to form the tag name. If no “timestep” or “timeval” option is given, all time steps will be read, resulting in several tags being created. If the “nomesh” option is given, the application must pass the entity set resulting from the original mesh read in to the function, that this set must contain the mesh read only from that file. The mesh in the file is checked against the mesh in the set to verify that the two correspond. The special name “MOAB_ALL_VARIABLES” can be used to indicate that all variables should be read. Multiple variable names can be specified, separated from each other by commas.

nomesh

Indicates that no mesh should be read from the file. This option is used in conjunction with the “variable=” option, to read variables and assign them as tags to a previously-read mesh. If this option is used, applications should pass an entity set to the read function, which should contain the mesh previously read from the file.

timestep=<step_number>[, ...]

Read the corresponding time step number for the specified variable(s). Tag names for the variable(s) will be formed by appending the time step number to the variable name. Multiple time step numbers can be specified, separated from each other by commas.

timeval=<time_value>[, ...]

Read the time step number whose time value is equal to or greater than the specified time value, for the specified variable(s). Tag names for the variable(s) will be formed by appending the time step number to the variable name. Multiple time step values can be specified, separated from each other by commas.

gather_set[=<rank>]

Create a gather set (associated with tag GATHER_SET) on one processor with the specified rank, to duplicate entities on other processors. If the rank is not specified, it will be rank 0 by default. If an

invalid rank is passed, no gather set will be created. Gather set is specially used by HOMME, MPAS, and any other unstructured grid.

References

[1] T.J. Tautges, R. Meyers, K. Merkley, C. Stimpson, and C. Ernst, *MOAB: A Mesh-Oriented Database*, Sandia National Laboratories, 2004.

[2] L. Diachin, A. Bauer, B. Fix, J. Kraftcheck, K. Jansen, X. Luo, M. Miller, C. Ollivier-Gooch, M.S. Shephard, T. Tautges, and H. Trease, "Interoperable mesh and geometry tools for advanced petascale simulations," *Journal of Physics: Conference Series*, vol. 78, 2007, p. 012015.

Appendix A: Summary

Table 1: Summary of MOAB meta-data conventions.

Convention	Applies to (E=ent, S=set)	Tag(s) (type/length)	Description
Name	E, S	NAME/C*32	
Title	S	TITLE/C*strlen	Title of mesh
Global identifier	E, S	GLOBAL_ID/I	
Geometric topology	S	GEOM_DIMENSION/I, GLOBAL_ID/I, NAME/C*32, CATEGORY/C*32. GEOM_SENSE_2/EH[2], GEOM_SENSE_N_ENTS/EH*N, GEOM_SENSE_N_SENSES/I*N	Sets contain mesh owned by that entity; parent/child links to bounded/bounding entities in geometric model
Material type	S	MATERIAL_SET/I	Set contains entities or sets assigned a common material type
Boundary condition	S	DIRICHLET_SET/I NEUMANN_SET/I	Set contains entities or sets assigned a particular boundary condition; neumann sets usually contain edges (2D) or faces (3D)
Parallel mesh constructs	E, S	PARALLEL_PART/I, PARALLEL_PARTITION/I, PSTATUS/C*1, PARALLEL_SHARED_PROC/I, PARALLEL/SHARED_HANDLE/H, PARALLEL_SHARED_PROCS/I*NP, PARALLEL_SHARED_HANDLES/H*NP	Data which describes parallel mesh
Structured mesh constructs	S	BOX_DIMS/I*6, GLOBAL_BOX_DIMS/I*6,	Data describing structured mesh

		BOX_PERIODIC/2*I, __BOX_SET/O	
Spectral mesh constructs	E, S	SPECTRAL_ORDER/I, SPECTRAL_VERTICES/I*(O+1)^2	Data marking spectral mesh constructs

*Table 2: Summary of MOAB conventional tag names, types, and purposes. Data types are I=integer, D=double, C=character, H=entity handle,O=opaque. Data type with *x denote length of x elements of that data type.*

Tag name	**Data type**	**Applies to (E=entity, S=set)**	**Purpose**
BOX_DIMS	I*6	S	Lower and upper ijk dimensions of box, ordered (ilo, jlo, klo, ihi, jhi, khi)
BOX_PERIODIC	I*2	S	Indicates whether box is periodic in i (BOX_PERIODIC[0]) or j (BOX_PERIODIC[1])
__BOX_SET	O	S	Pointer to corresponding ScdBox instance
CATEGORY	C*32	S	String describing purpose of set; examples include “group”, “vertex”, “edge”, “surface”, “volume”
DIRICHLET_SET	I	S	Entities or sets with common boundary condition
GEOM_ DIMENSION	I	S	Identifies mesh entities resolving a given geometric model entity
GEOM_SENSE_2	EH*2	S	Stored on face-type geometric topology sets, values store regions having forward and reverse sense
GEOM_SENSE_N_ENTS	EH*N	S	Stored on edge-type geometric topology sets, values store faces whose senses are stored in GEOM_SENSE_N_SENSES.
GEOM_SENSE_N_ SENSES	I*N	S	Stored on edge-type geometric topology sets, values store senses of the edge with respect to faces stored in GEOM_SENSE_N_ENTS.
GLOBAL_ID	I	E, S	Application-specific entity id
MATERIAL_SET	I	S	Entities or sets grouped by material type
NAME	C*32	E, S	User-assigned entity name(s); multiple names delimited with ?
NEUMANN_SET	I	S	Entities or sets with common boundary condition
PARALLEL_PART	I	S	Represent a part in a partition
PARALLEL_ PARTITION	I	S	Represents a partition of the mesh for parallel solution, which is a collection of parts

__PARALLEL_SHARED_HANDLE	H	E, S	Handle of this entity/set on sharing processor
__PARALLEL_SHARED_PROC	I	E, S	Rank of other processor sharing this entity/set
__PARALLEL_SHARED_HANDLES	H*NP	E, S	Handles of this entity/set on sharing processors
__PARALLEL_SHARED_PROCS	I*NP	E, S	Ranks of other processors sharing this entity/set
__PARALLEL_STATUS	C*1	E, S	Bit-field indicating various parallel information
SPECTRAL_ORDER	I	S	Order of a spectral mesh
SPECTRAL_VERTICES	H*(O+1)^d	E	Vertices comprising a spectral element, ordered lexicographically; here, O=value of SPECTRAL_ORDER tag.

Appendix B: CCMIO (Star-CD, Star-CCM+) Reader/Writer Conventions

Table 3: Translation between CCMIO options and MOAB tags.

Set Type	**CCMIO Construct**	**MOAB Tag Name, Type**
File set / Interface	Title (option)	“Title” (C*32)
	CreatingProgram	“CreatingProgram” (C*32)
Material sets	Index	MATERIAL_SET
	Label[1]	NAME
	MaterialId	“MaterialId” (I)
	Radiation	“Radiation” (I)
	PorosityId	“PorosityId” (I)
	SpinId	“SpinId” (I)
	GroupId	“GroupId” (I)
	ColorIdx	“ColorIdx” (I)
	ProcessorId	“ProcessorId” (I)
	LightMaterial	“LightMaterial” (I)
	FreeSurfaceMaterial	“FreeSurfaceMaterial” (I)
	Thickness	“Thickness” (F)
	MaterialType	“MaterialType” (C*32)
Neumann sets	Index	NEUMANN_SET

	Label	NEUMANN_SET
	BoundaryName	NAME
	BoundaryType	“BoundaryType” (C*32)
	ProstarRegionNumber	“ProstarRegionNumber” (I)

Notes:

1. If no name is present, labels the material group with “MaterialX”, where X is the index of that group.

Appendix C: ExodusII Reader/Writer Conventions

Table 4: Translation between ExodusII constructs and MOAB tags.

Data Type	**ExodusII Construct**	**MOAB Tag Name, Type**
	QA records	“qaRecord” (C*(v))[2]
Material sets	Block number	MATERIAL_SET
	Block element type	Entity type, # vertices per entity
Dirichlet sets[3]	Nodeset number	DIRICHLET_SET
	Distribution factors	“distFactor” (D*(v))[1]
Neumann sets	Sideset number	NEUMANN_SET
	Distribution factors	“distFactor” (D*(v))[1]
Neumann sets, reverse faces[3]	Sides	NEUSET_SENSE
Nodes, elements	node_num_map, elem_map	GLOBAL_ID on nodes/elements

Notes:

1. Variable-length tag used for distribution factors; length for each set is the number of entities in each set, such that there is one distribution factor for each entity in the set.
2. QA records are stored as variable-length tags on file set specified on read. Tag is a concatenation of QA record strings into a single string, with '\0' used to delimit lines.
3. MOAB represents sidesets as sets of faces, rather than as sides of elements. Faces can be ordered “forward” or “reverse” with respect to one of the bounding elements, depending on whether the right-hand normal points into or out of the element. Forward-sense faces are added to the Neumann set. Reverse-sense faces are put into a separate set; that set is tagged with the NEUSET_SENSE tag, with value = -1; and that reverse set is added to the Neummann set.

Appendix D: NC (Climate Data) Reader/Writer Conventions

The climate data reader in MOAB reads files with the '.nc' filename extension. By default, this reader reads the whole mesh in the file and creates it as structured mesh in MOAB, with the mesh accessible through MOAB's structured mesh interface. By default, all variables and timesteps are read from the file, and written as tags on the mesh vertices from that file. This behavior is controlled by the “variable”, “nomesh”, “timestep”, and “timeval” options described earlier in this document. If MOAB is compiled for parallel execution and configured with a pnetcdf reader, the mesh is read in parallel, with a 1D or 2D decomposition designed to balance read performance and communication interface

size (for details on the partitioning method used, see the src/io/ReadNC.cpp source file).

Mesh is put into the entity set provided to the load_file function. This entity set is also annotated with various tags representing information read from the file. These tags are described in Table 5.

Reading unstructured NC files in the HOMME format is also supported. Currently a trivial element-based partition is the only option for parallel reading. As the data is unstructured, it is necessary to have a connectivity file to define the vertex adjacencies. The default convention is to have a file called HommeMapping.nc in the same directory as the the variable data file. If this convention is not followed, the connectivity file can be specified with the option -O CONN="/path/to/connectivity.nc". An example of mbconvert using the parallel read capability is shown below:

```
    mpiexec -np 2 tools/mbconvert -O TRIVIAL_PARTITION -O DEBUG_IO=1
-o DEBUG_IO=9 -o PARALLEL=WRITE_PART
/nfs2/hayes6/meshlab/homme_data/camrun.cam2.h0.0000-01-01-16200.nc
output.h5m
```

Several other things to note about reading climate data files into MOAB:

- *Time-dependent variables:* MOAB currently has no mechanism for time-dependent tags. Therefore, time-dependent variables are represented using one tag per timestep, with the tag name set as the variable name plus the timestep index. Thus, the first few timesteps for the variable TEMPERATURE would be represented in tags named TEMPERATURE0, TEMPERATURE1, etc.
- *Cell- and face-centered variables:* The climate data reader currently does not do cell- and face-centered variables correctly.

*Table 5: Summary of MOAB conventional tag names, types, and purposes. Data types are I=integer, D=double, C=character, H=entity handle. Data type with *x denote length of x elements of that data type; data type with *var denote variable-length tag. Tag names with two underscores prepended ("__") denote tags not written to a file by MOAB.*

Tag name	**Data type**	**Applies to (E=entity, S=set)**	**Purpose**
__NUM_DIMS	I	S	The number of dimensions in the netcdf file.
__NUM_VARS	I	S	The number of variables in the netcdf file.
__DIM_NAMES	C*var	S	The dimension names, concatenated into a character string, with '\0' terminating each name.
__VAR_NAMES	C*var	S	The variable names, concatenated into a character string, with '\0' terminating each name.
<dim_name>	(I or D)*var	S	For each dimension, the values for the dimension. The data type for this tag corresponds to that in the netcdf file. The length of this tag is the number of values stored for the dimension in the netcdf file.
__<dim_name>_LOC_MINMAX	2*(I or D)	S	The indices (0-based) of the local min and max values of dimension stored locally. For spatial

			dimensions like lon or lat, this will store the minimum and maximum indices in the local partition of the grid. For dimensions like time, where each processor represents the entire dimension, this will likely store 0 and the number of values for that dimension. Only one of __<dim_name>_LOC_VALS and _LOC_MIN_MAX can be used for a given dimension.
__<dim_name>_LOC_VALS	(I or D)*var	S	The indices (0-based) of the dimension stored locally. This tag only makes sense for dimensions that can be read in multiple pieces, such as time. Only one of __<dim_name>_LOC_VALS and _LOC_MIN_MAX can be used for a given dimension.
__<var_name>_DIMS	C*n	S	For each variable, the tag handles for the dimensions defining this variable, in netcdf ordering (last dimension varying fastest). The length of this tag, n, is # dimensions for the variable * sizeof(TagHandle).
<var_name><timestep_ind>	(data type)	E	Values of the variable for timestep <timestep_ind> for vertices. The data type of this tag corresponds to that of the variable from the netcdf file. Timestep index is 0-based.
__GLOBAL_ATTRIBS	C*Var	S	The global attributes, concatenated into a character string, with '\0' terminating each attribute name, ';' separating the data type and value, and ';' separating one name/data type/value from the next.
__GLOBAL_ATTRIBS_LEN	I*Var	S	A vector of integers, marking the end position of each attribute (name/data type/value) in __GLOBAL_ATTRIBS tag.
__<var_name>_ATTRIBS	C*Var	S	The variable attributes, concatenated into a character string, with '\0' terminating each attribute name, ';' separating the data type and value, and ';' separating one name/data type/value from the next.
__<var_name>_ATTRIBS_LEN	I*Var	S	A vector of integers, marking the end position of each attribute (name/data type/value) in __<var_name>_ATTRIBS tags.

Appendix E: Nek5000 Reader/Writer Conventions

Nek5000, or Nek, is a code that uses the spectral element method to model fluid, heat transfer, electromagnetics, and other physics. Nek uses unstructured hexahedral meshes, with each hex element

resolved by a structured grid of “Gauss Lebato Legendre” (GLL) points. Nek can read meshes through MOAB, and can output physics variables and GLL points through MOAB as well.

Since fluid is a single material in Nek, no material sets are needed. Boundary conditions are mapped to Nek's cbc array using Neumann sets and a user-provided “usr_moab2nek” subroutine (for an example of this subroutine, see examples/moab/pipe.usr in the Nek source code). GLL point locations and fluid variables on those points are stored in tags on the hex elements. All hex elements have the same number of GLL points. The number of GLL points in each direction is stored in a tag on the mesh instance. These tags are described in Table 6.

GLL point locations and fluid variables are stored in lexicographic order, similar to their storage order inside the Nek code.

*Table 6: Summary of MOAB conventional tag names, types, and purposes for Nek. Data types are I=integer, D=double, C=character, H=entity handle. Data type with *x denote length of x elements of that data type; data type with *var denote variable-length tag. Tag names with two underscores prepended (“__”) denote tags not written to a file by MOAB.*

Tag name	**Data type**	**Applies to (E=entity, S=set)**	**Purpose**
SEM_DIMS	I*3	S	The dimensions of the GLL mesh in each hex element.
SEM_X	D*nx*ny*nz	E	X position of GLL points (having nx*ny*nz values)
SEM_Y	D*nx*ny*nz	E	Y position of GLL points (having nx*ny*nz values)
SEM_Z	D*nx*ny*nz	E	Z position of GLL points (having nx*ny*nz values)
VEL_X	D*nx*ny*nz	E	Fluid velocities in the x direction for GLL point array (having nx*ny*nz values)
VEL_Y	D*nx*ny*nz	E	Fluid velocities in the y direction for GLL point array (having nx*ny*nz values)
VEL_Z	D*nx*ny*nz	E	Fluid velocities in the z direction for GLL point array (having nx*ny*nz values)
TEMP	D*nx*ny*nz	E	Fluid temperature for GLL point array (having nx*ny*nz values)
PRESS	D*nx*ny*nz	E	Fluid pressure for GLL point array (having nx*ny*nz values)