

SAND2008-xxxx  
DRAFT  
Draft Date: June 3, 2019

# EXODUS: A Finite Element Data Model

Gregory D. Sjaardema  
Larry A. Schoof  
Victor R. Yarberry  
Computational Mechanics and Visualization Department  
Sandia National Laboratories  
Albuquerque, NM 87185

## Abstract

EXODUS is a model developed to store and retrieve data for finite element analyses. It is used for preprocessing (problem definition), postprocessing (results visualization), as well as code to code data transfer. An EXODUS data file is a random access, machine independent, binary file that is written and read via C, C++, or Fortran library routines which comprise the Application Programming Interface (API).

See also the doxygen-generated documentation at <http://gsjaardema.github.io/seacas/html/index.html>

# Contents

<b>1</b>	<b>Introduction</b>	<b>6</b>
1.1	License and Availability . . . . .	6
<b>2</b>	<b>Changes Since First Printing</b>	<b>8</b>
<b>3</b>	<b>Development of EXODUS</b>	<b>9</b>
<b>4</b>	<b>Description of Data Objects</b>	<b>10</b>
4.1	Global Parameters . . . . .	11
4.2	Quality Assurance Data . . . . .	12
4.3	Information Data . . . . .	12
4.4	Nodal Coordinates . . . . .	12
4.4.1	Coordinate Names . . . . .	12
4.5	Node Number Map . . . . .	12
4.6	Element Number Map . . . . .	13
4.7	Optimized Element Order Map . . . . .	14
4.8	Element Blocks . . . . .	14
4.8.1	Element Block Parameters . . . . .	14
4.8.2	Element Connectivity . . . . .	15
4.8.3	Element Attributes . . . . .	21
4.9	Node Sets . . . . .	21
4.9.1	Node Set Parameters . . . . .	24
4.9.2	Node Set Node List . . . . .	24
4.9.3	Node Set Distribution Factors . . . . .	26
4.10	Concatenated Node Sets . . . . .	26
4.11	Side Sets . . . . .	27
4.11.1	Side Set Parameters . . . . .	27

4.11.2	Side Set Element List . . . . .	27
4.11.3	Side Set Side List . . . . .	27
4.11.4	Side Set Node List . . . . .	27
4.11.5	Side Set Node Count List . . . . .	29
4.11.6	Side Set Distribution Factors . . . . .	29
4.12	Concatenated Side Sets . . . . .	31
4.12.1	Object Properties . . . . .	31
4.12.2	Property Values . . . . .	32
4.13	Results Parameters . . . . .	32
4.13.1	Results Names . . . . .	32
4.14	Results Data . . . . .	33
4.14.1	Time Values . . . . .	33
4.14.2	Global Results . . . . .	33
4.14.3	Nodal Results . . . . .	33
4.14.4	Element Results . . . . .	33
4.15	Element Variable Truth Table . . . . .	34
<b>5</b>	<b>Application Programming Interface (API)</b>	<b>35</b>
5.1	Data File Utilities . . . . .	36
5.1.1	Create EXODUS File . . . . .	36
5.1.2	Open EXODUS File . . . . .	38
5.1.3	Close EXODUS File . . . . .	39
5.1.4	Write Initialization Parameters . . . . .	39
5.1.5	Read Initialization Parameters . . . . .	41
5.1.6	Write Quality Assurance (QA) Records . . . . .	42
5.1.7	Read Quality Assurance (QA) Records . . . . .	43
5.1.8	Write Information Records . . . . .	44
5.1.9	Read Information Records . . . . .	45
5.1.10	Inquire EXODUS Parameters . . . . .	45
5.1.11	Inquire EXODUS Integer Parameters . . . . .	48
5.1.12	Error Reporting . . . . .	50
5.1.13	Set Error Reporting Level . . . . .	50
5.2	Model Description . . . . .	51
5.2.1	Write Nodal Coordinates . . . . .	51
5.2.2	Read Nodal Coordinates . . . . .	52

5.2.3	Write Coordinate Names . . . . .	53
5.2.4	Read Coordinate Names . . . . .	54
5.2.5	Write Node Number Map . . . . .	55
5.2.6	Read Node Number Map . . . . .	55
5.2.7	Write Element Number Map . . . . .	56
5.2.8	Read Element Number Map . . . . .	57
5.2.9	Write Element Order Map . . . . .	57
5.2.10	Read Element Order Map . . . . .	58
5.2.11	Write Element Block Parameters . . . . .	59
5.2.12	Read Element Block Parameters . . . . .	60
5.2.13	Read Element Blocks IDs . . . . .	62
5.2.14	Write Element Block Connectivity . . . . .	62
5.2.15	Read Element Block Connectivity . . . . .	63
5.2.16	Write Element Block Attributes . . . . .	63
5.2.17	Read Element Block Attributes . . . . .	64
5.2.18	Write Node Set Parameters . . . . .	65
5.2.19	Read Node Set Parameters . . . . .	66
5.2.20	Write Node Set . . . . .	67
5.2.21	Write Node Set Distribution Factors . . . . .	68
5.2.22	Read Node Set Distribution Factors . . . . .	68
5.2.23	Read Node Sets IDs . . . . .	69
5.2.24	Write Concatenated Node Sets . . . . .	70
5.2.25	Read Concatenated Node Sets . . . . .	72
5.2.26	Write Side Set Parameters . . . . .	73
5.2.27	Read Side Set Parameters . . . . .	74
5.2.28	Write Side Set . . . . .	76
5.2.29	Read Side Set . . . . .	76
5.2.30	Write Side Set Distribution Factors . . . . .	77
5.2.31	Read Side Set Distribution Factors . . . . .	78
5.2.32	Read Side Sets IDs . . . . .	79
5.2.33	Read Side Set Node List . . . . .	79
5.2.34	Write Concatenated Side Sets . . . . .	80
5.2.35	Read Concatenated Side Sets . . . . .	82
5.2.36	Convert Side Set Nodes to Sides . . . . .	84
5.2.37	Write Property Arrays Names . . . . .	86

5.2.38	Read Property Arrays Names . . . . .	88
5.2.39	Write Object Property . . . . .	89
5.2.40	Read Object Property . . . . .	90
5.2.41	Write Object Property Array . . . . .	91
5.2.42	Read Object Property Array . . . . .	92
5.3	Results Data . . . . .	94
5.3.1	Write Results Variables Parameters . . . . .	94
5.3.2	Read Results Variables Parameters . . . . .	95
5.3.3	Write Results Variables Names . . . . .	96
5.3.4	Read Results Variable Names . . . . .	97
5.3.5	Write Time Value for a Time Step . . . . .	98
5.3.6	Read Time Value for a Time Step . . . . .	99
5.3.7	Read All Time Values . . . . .	99
5.3.8	Write Element Variable Truth Table . . . . .	100
5.3.9	Read Element Variable Truth Table . . . . .	101
5.3.10	Write Element Variable Values at a Time Step . . . . .	102
5.3.11	Read Element Variable Values at a Time Step . . . . .	104
5.3.12	Read Element Variable Values through Time . . . . .	105
5.3.13	Write Global Variables Values at a Time Step . . . . .	106
5.3.14	Read Global Variables Values at a Time Step . . . . .	107
5.3.15	Read Global Variable Values through Time . . . . .	108
5.3.16	Write Nodal Variable Values at a Time Step . . . . .	109
5.3.17	Read Nodal Variable Values at a Time Step . . . . .	111
5.3.18	Read Nodal Variable Values through Time . . . . .	112
<b>6</b>	<b>References</b>	<b>114</b>
<b>A</b>	<b>Implementation of EXODUS with NetCDF</b>	<b>116</b>
A.1	Description . . . . .	116
A.2	Efficiency Issues . . . . .	116
<b>B</b>	<b>Deprecated Functions</b>	<b>117</b>
<b>C</b>	<b>Sample Code</b>	<b>123</b>
C.1	Write Example Code . . . . .	123
C.2	Read Example Code . . . . .	136

Intentionally Left Blank

# Chapter 1

## Introduction

EXODUS is the successor of the widely used finite element (FE) data file format EXODUS [1] (henceforth referred to as EXODUS I) developed by Mills-Curran and Flanagan. It continues the concept of a common database for multiple application codes (mesh generators, analysis codes, visualization software, etc.) rather than code-specific utilities, affording flexibility and robustness for both the application code developer and application code user. By using the EXODUS data model, a user inherits the flexibility of using a large array of application codes (including vendor-supplied codes) which access this common data file directly or via translators.

The uses of the EXODUS data model include the following:

- problem definition – mesh generation, specification of locations of boundary conditions and load application, specification of material types.
- simulation – model input and results output.
- visualization – model verification, results postprocessing, data interrogation, and analysis tracking.

### 1.1 License and Availability

The EXODUS library is licensed under the BSD open source license.

Copyright © 2005 National Technology & Engineering Solutions of Sandia, LLC (NT-ESS). Under the terms of Contract DE-NA0003525 with NTESS, the U.S. Government retains certain rights in this software.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

- Neither the name of NTESS nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

The EXODUS library source code is available on GitHub <https://github.com/gsjardema/seacas.git>

For bug reports, documentation errors, and enhancement suggestions, contact:

- Gregory D. Sjaardema
- PHONE: (505) 844-2701
- EMAIL: <mailto:gdsjaar@sandia.gov>
- EMAIL: <mailto:gsjaardema@gmail.com>



## Chapter 2

# Changes Since First Printing

There have been several changes to the EXODUS API in the years since the original EXODUS report was published. The main changes are:

- Addition of Coordinate Frames.
- Addition of node set and side set results variables.
- Addition of element block, node set, side set, element map, and node map names.
- Support for very large model.
- Efficient replication of the model definition “genesis” portion of the database.
- Multiple, optional named node and element maps which can be used for any purpose.
- Support for “meshes” with no node or elements; or nodes, but no elements.

There have also been some functions added to make it easier to write an EXODUS database efficiently. These include:

- API function to write concatenated element block information, and
- API function to defined all results data with one function call.

## Chapter 3

# Development of EXODUS

The evolution of the EXODUS() data model has been steered by FE application code developers who desire the advantages of a common data format. The EXODUS model has been designed to overcome deficiencies in the EXODUS I file format and meet the following functional requirements as specified by these developers:

- random read/write access.
- application programming interface (API) – provide routines callable from FORTRAN, C, and C++ application codes.
- extensible – allow new data objects to be added without modifying the application programs that use the file format.
- machine independent – data should be independent of the machine which generated it.
- real time access during analysis – allow access to the data in a file while the file is being created.

To address these requirements, the public domain database library netCDF [3] was selected to handle the low-level data storage. The EXODUS library functions provide the mapping between FE data objects and netCDF dimensions, attributes, and variables. (These mappings are documented in Appendix A.) Thus, the code developer interacts with the data model using the vocabulary of an FE analyst (element connectivity, nodal coordinates, etc.) and is relieved of the details of the data access mechanism. To provide machine independency, the netCDF library stores data in eXternal Data Representation (XDR) [4] format.

Because an EXODUS file is a netCDF file, an application program can access data via the EXODUS API or via netCDF API function calls directly. Although the latter two methods require more in-depth understanding of netCDF, this capability is a powerful feature that allows the development of auxiliary libraries of special purpose functions not offered in the standard EXODUS() library. For example, if an application required access to the coordinates of a single node (the standard library function returns the coordinates for all of the nodes in the model), a simple function could be written that calls netCDF routines directly to read the data of interest.

## Chapter 4

# Description of Data Objects

entity	block	set	map
node		nodeset	nodemap
edge	edgeblock	edgeset	edgemap
face	faceblock	faceset	facemap
element	element block	elementset	elementmap

The data in EXODUS files can be divided into three primary categories: initialization data, model data, and results data.

Initialization data includes sizing parameters (number of nodes, number of elements, etc.), optional quality assurance information (names of codes that have operated on the data), and optional informational text.

The model is described by data which are static (do not change through time). This data includes nodal coordinates, element connectivity (node lists for each element), element attributes, and node sets and side sets (used to aid in applying loading conditions and boundary constraints).

The results are optional and include five types of variables – nodal, element, nodeset, sideset, and global – each of which is stored through time. Nodal results are output (at each time step) for all the nodes in the model. An example of a nodal variable is displacement in the X direction. Element, nodeset, and sideset results are output (at each time step) for all entities (elements, nodes, sides) in one or more entity block. For example, stress may be an element variable. Another use of element variables is to record element status (a binary flag indicating whether each element is "alive" or "dead") through time. Global results are output (at each time step) for a single element or node, or for a single property. Linear momentum of a structure and the acceleration at a particular point are both examples of global variables. Although these examples correspond to typical FE applications, the data format is flexible enough to accommodate a spectrum of uses.

A few conventions and limitations must be cited:

- There are no restrictions on the frequency of results output except that the time value associated with each successive time step must increase monotonically.
- To output results at different frequencies (i.e., variable A at every simulation time step, variable B at every other time step) multiple EXODUS files must be used.
- There are no limits to the number of each type of results, but once declared, the number cannot change.

- If the mesh geometry changes in time (i.e., number of nodes increases, connectivity changes), the new geometry must be output to a new EXODUS file.

The following sections describe the data objects that can be stored in an EXODUS file. API functions that read / write the particular objects are included for reference. API routines for the C binding are in lower case. Refer to Section 4 on page 21 for a detailed description of each API function.

## 4.1 Global Parameters

*API Functions:* `ex_put_init`, `ex_get_init`

Every EXODUS file is initialized with the following parameters:

- Title – data file title of length `MAX_LINE_LENGTH`. Refer to discussion below for definition of `MAX_LINE_LENGTH`.
- Number of nodes – the total number of nodes in the model.
- Problem dimensionality – the number of spatial coordinates per node (1, 2, or 3).
- Number of elements – the total number of elements of all types in the file.
- Number of element blocks – within the EXODUS data model, elements are grouped together into blocks. Refer to Section 3.8 on page 8 for a description of element blocks.
- Number of node sets – node sets are a convenient method for referring to groups of nodes. Refer to Section 3.9 on page 11 for a description of node sets.
- Number of side sets – side sets are used to identify elements (and their sides) for specific purposes. Refer to Section 3.11 on page 12 for a description of side sets.
- Database version number – the version of the data objects stored in the file. This document describes database version is 4.72.
- API version number – the version of the EXODUS library functions which stored the data in the file. The API version can change without changing the database version and vice versa. This document describes API version 4.72.
- I/O word size – indicates the precision of the floating point data stored in the file. Currently, four- or eight-byte floating point numbers are supported. It is not necessary that an application code be written to handle the same precision as the data stored in the file. If required, the routines in the EXODUS library perform automatic conversion between four- and eight-byte numbers.
- Length of character strings – all character data stored in an EXODUS file is either of length `MAX_STR_LENGTH` or `MAX_LINE_LENGTH`. These two constants are defined in the file *exodusII.h*. Current values are 32 and 80, respectively.
- Length of character lines – see description above for length of character strings.

## 4.2 Quality Assurance Data

*API Functions:* `ex_put_qa`, `ex_get_qa`

Quality assurance (QA) data is optional information that can be included to indicate which application codes have operated on the data in the file. Any number of QA records can be included, with each record containing four character strings of length `MAX_STR_LENGTH`. The four character strings are the following (in order):

**Code name** indicates the application code that has operated on the EXODUS file.

**Code QA descriptor** provides a location for a version identifier of the application code.

**Date** the date on which the application code was executed; should be in the format 20080331.

**Time** the 24-hour time at which the application code was executed; should be in the format hours:minutes:seconds, such as 16:30:15.

## 4.3 Information Data

*API Functions:* `ex_put_info`, `ex_get_info`

This is for storage of optional supplementary text. Each text record is of length `MAX_LINE_LENGTH`; there is no limit to the number of text records.

## 4.4 Nodal Coordinates

*API Functions:* `ex_put_coord`, `ex_get_coord`

The nodal coordinates are the floating point spatial coordinates of all the nodes in the model. The number of nodes and the problem dimension define the length of this array. The node index cycles faster than the dimension index, thus the X coordinates for all the nodes is written before any Y coordinate data are written. Internal node numbers (beginning with 1) are implied from a node's place in the nodal coordinates record. See Section 4.5 for a discussion of internal node numbers.

### 4.4.1 Coordinate Names

*API Functions:* `ex_put_coord_names`, `ex_get_coord_names`

The coordinate names are character strings of length `MAX_STR_LENGTH` which name the spatial coordinates. There is one string for each dimension in the model, thus there are one to three strings.

## 4.5 Node Number Map

*API Functions:* `ex_put_node_num_map`, `ex_get_node_num_map`

Within the data model, internal node IDs are indices into the nodal coordinate array and internal element IDs are indices into the element connectivity array. Thus, internal node and element numbers (IDs) are contiguous (i.e., 1...number of nodes and 1...number of elements, respectively). Optional node and element number maps can be stored to relate user-defined node and element IDs to these internal node and element numbers. The length of these maps are number of nodes and number of elements, respectively. As an example, suppose a database contains exactly one QUAD element with four nodes. The user desires the element ID to be 100 and the node IDs to be 10, 20, 30, and 40 as shown in Figure 4.1.

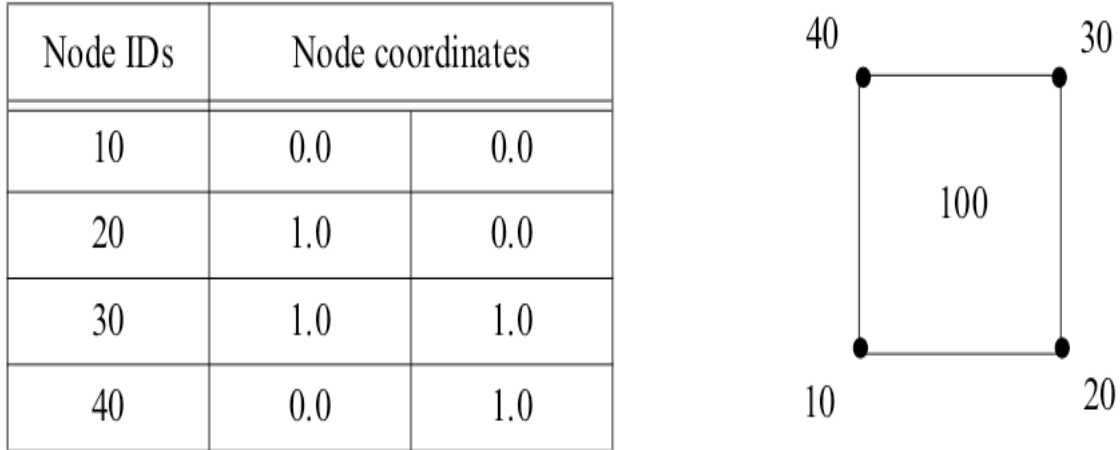


Figure 4.1: User-defined Node and Element IDs.

The internal data structures representing the above model would be the following:

- nodal coordinate array: (0.0, 1.0, 1.0, 0.0, 0.0, 0.0, 1.0, 1.0)
- connectivity array: (1, 2, 3, 4)
- node number map: (10, 20, 30, 40)
- element number map: (100)

Internal (contiguously numbered) node and element IDs must be used for all data structures that contain node or element numbers (IDs), including node set node lists, side set element lists, and element connectivity. Additionally, to inquire the value(s) of node or element results variables, an application code must pass the internal node or element number for the node or element of interest.

## 4.6 Element Number Map

*API Functions:* `ex_put_elem_num_map`, `ex_get_elem_num_map`

Refer to Section 3.5 for a discussion of the optional element number map.

## 4.7 Optimized Element Order Map

*API Functions:* `ex_put_map`, `ex_get_map`

The optional element order map defines the element order in which a solver (e.g., a wavefront solver) should process the elements. For example, the first entry is the number of the element which should be processed first by the solver. The length of this map is the total number of elements in the model.

## 4.8 Element Blocks

For efficient storage and to minimize I/O, elements are grouped into element blocks. Within an element block, all elements are of the same type (basic geometry and number of nodes). This definition does not preclude multiple element blocks containing the same element type (i.e., “QUAD” elements may be in more than one element block); only that each element block may contain only one element type.

The internal number of an element numbering is defined implicitly by the order in which it appears in the file. Elements are numbered internally (beginning with 1) consecutively across all element blocks. See Section 4.6 for a discussion of internal element numbering.

### 4.8.1 Element Block Parameters

*API Functions:* `ex_put_elem_block`, `ex_get_elem_block`, `ex_get_elem_blk_ids`

The following parameters are defined for each element block:

- element block ID – an arbitrary, unique, positive integer which identifies the particular element block. This ID is used as a “handle” into the database that allows users to specify a group of elements to the application code without having to know the order in which element blocks are stored in the file.
- element type Element type – a character string of length `MAX_STR_LENGTH` to distinguish element types. All elements within the element block are of this type. Refer to Table 4.1 on page 21 for a list of names that are currently accepted. It should be noted that the EXODUS library routines do not verify element type names against a standard list; the interpretation of the element type is left to the application codes which read or write the data. In general, the first three characters uniquely identify the element type. Application codes can append characters to the element type string (up to the maximum length allowed) to further classify the element for specific purposes.
- Number of elements – the number of elements in the element block.
- Nodes per element – the number of nodes per element for the element block.
- Number of attributes – the number of attributes per element in the element block. See below for a discussion of element attributes.

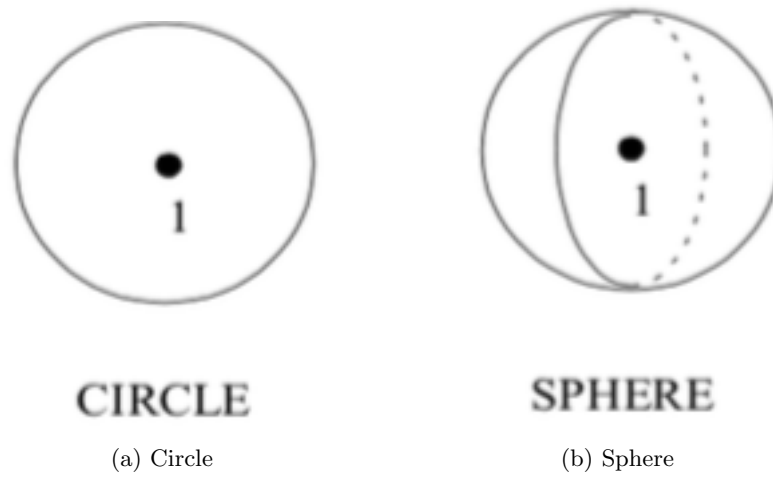


Figure 4.2: Node Ordering for Circle and Sphere Elements

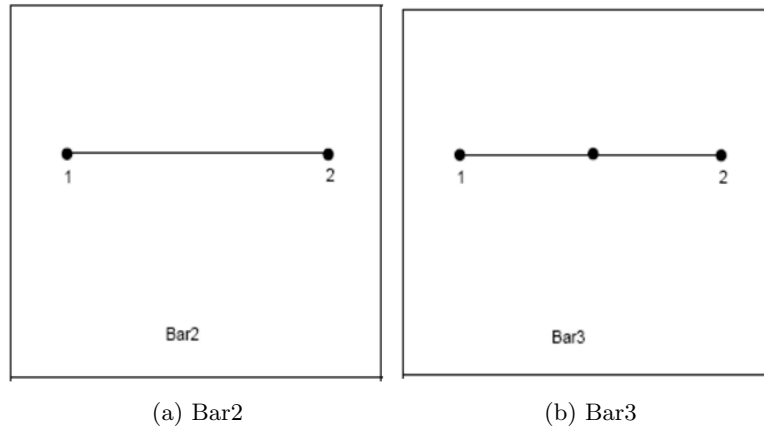


Figure 4.3: Node Ordering for Bar/Truss/Beam Elements

### 4.8.2 Element Connectivity

*API Functions:* `ex_put_elem_conn`, `ex_get_elem_conn`

The element connectivity contains the list of nodes (internal node IDs; see Section 4.6 for a discussion of node IDs) which define each element in the element block. The length of this list is the product of the number of elements and the number of nodes per element as specified in the element block parameters. The node index cycles faster than the element index. Node ordering follows the conventions illustrated in Figures 4.2 through 4.14. The node ordering conventions follow the element topology used in PATRAN [?]. Thus, for higher-order elements than those illustrated, use the ordering prescribed in the PATRAN User Manual [http://web.mscsoftware.com/training\\_videos/patran/reverb3/index.html#page/Finite%2520Element%2520Modeling/element\\_lib\\_topics.16.1.html#ww33606](http://web.mscsoftware.com/training_videos/patran/reverb3/index.html#page/Finite%2520Element%2520Modeling/element_lib_topics.16.1.html#ww33606). For elements of type CIRCLE or SPHERE, the topology is one node at the center of the circle or sphere element.



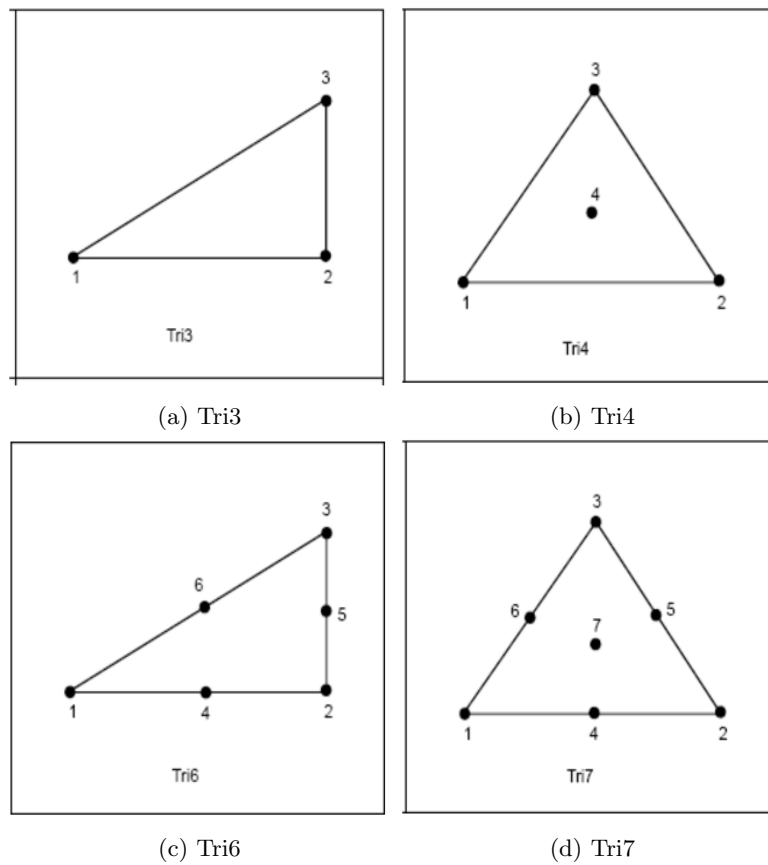


Figure 4.4: Node Ordering for Triangular Elements.

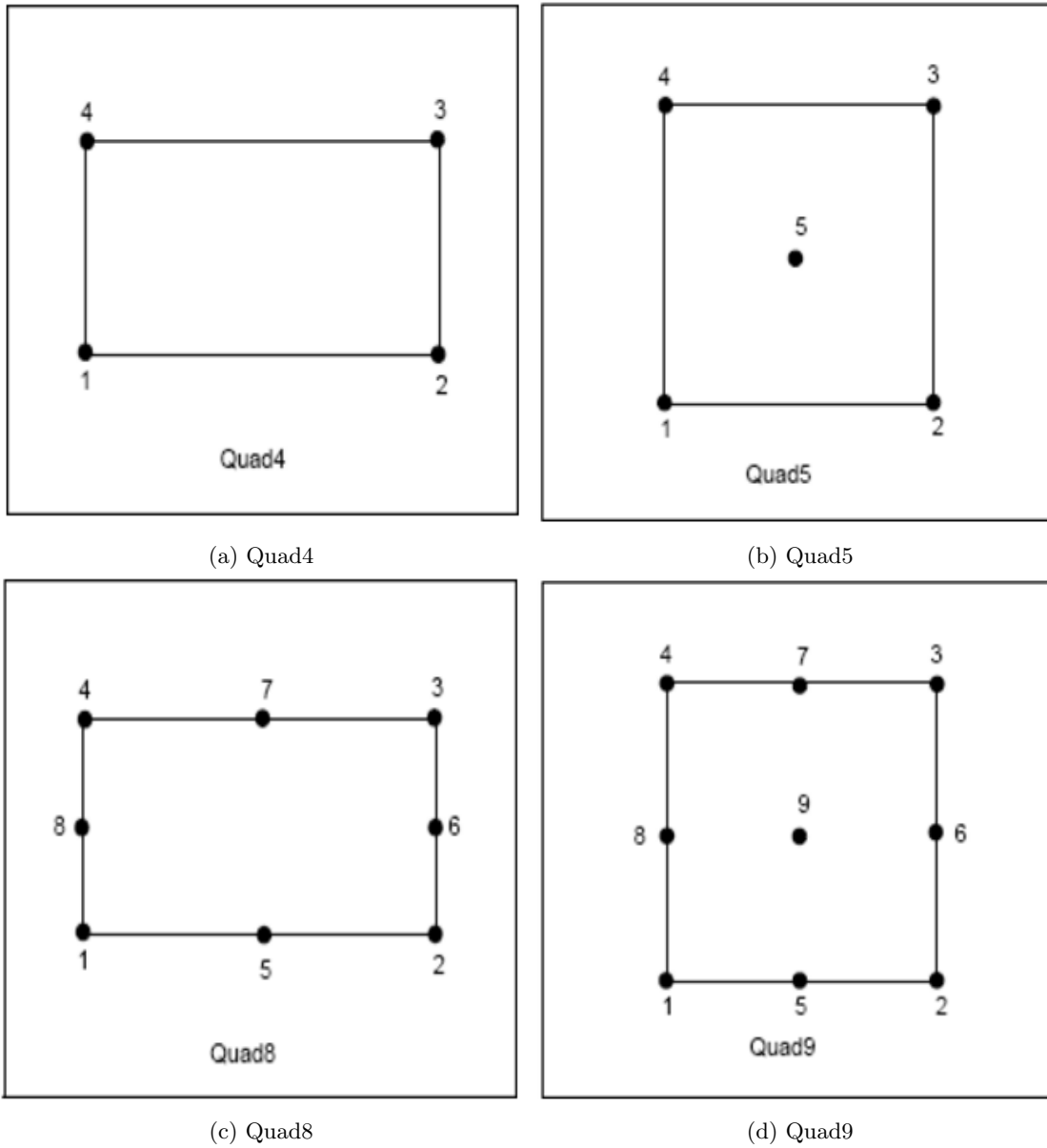


Figure 4.5: Node Ordering for Quadrilateral Elements.

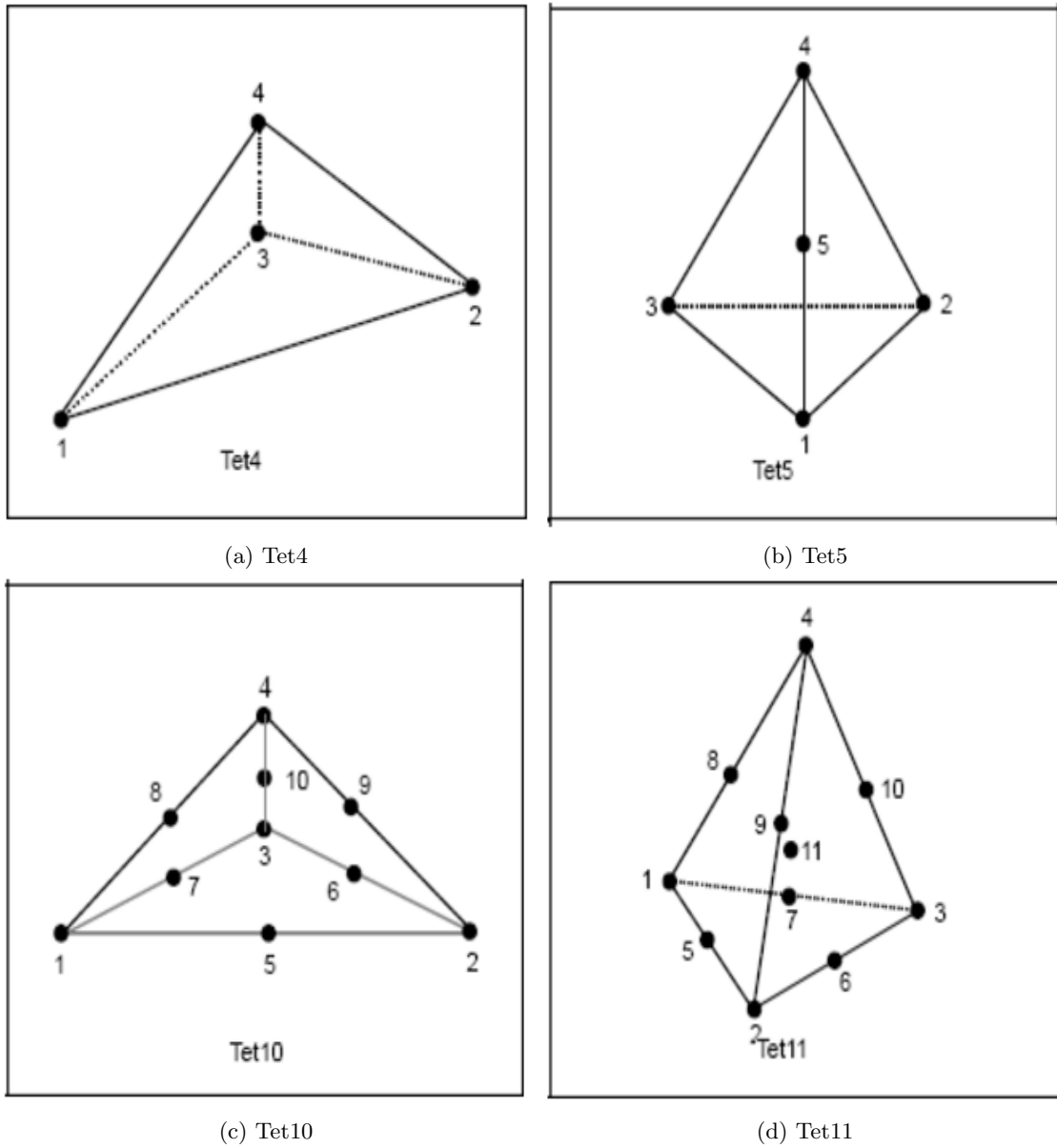


Figure 4.6: Node Ordering for Tetrahedral Elements.

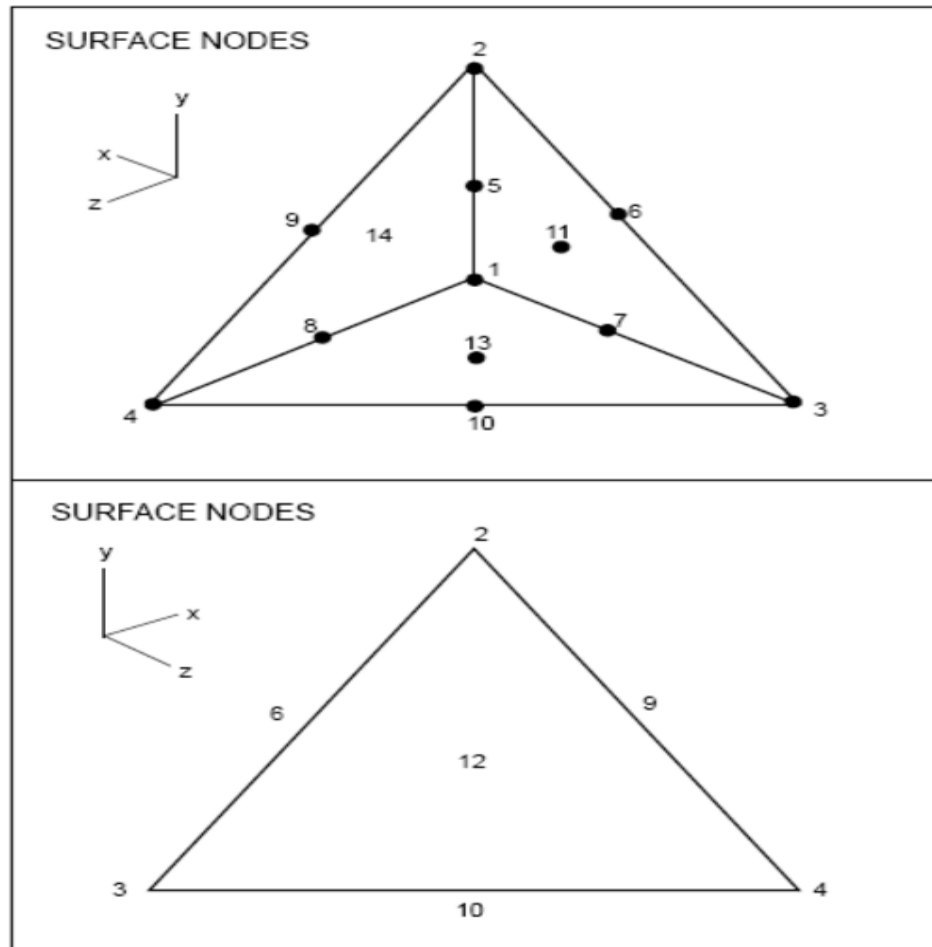


Figure 4.7: Node Ordering for Tetrahedral Tet14 Element.

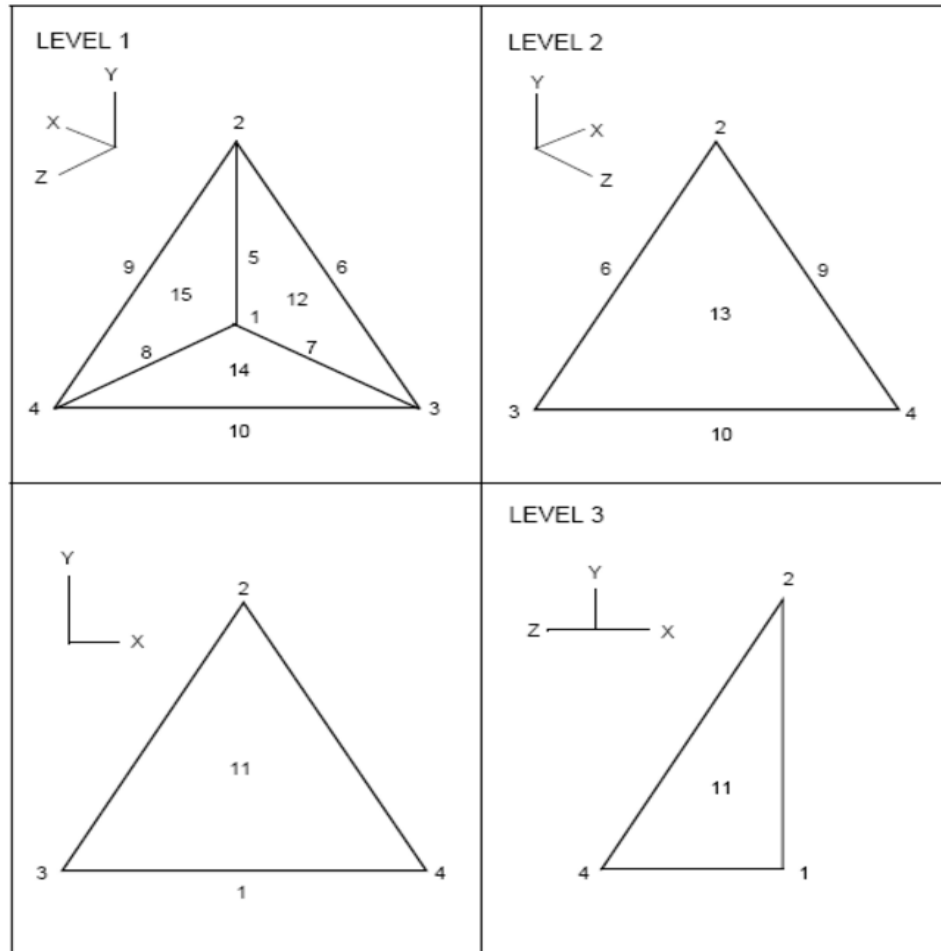


Figure 4.8: Node Ordering for Tetrahedral Tet15 Element.

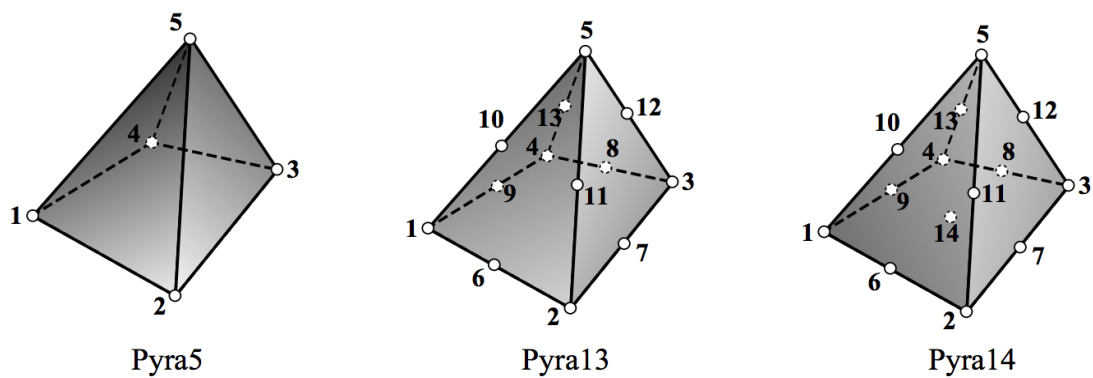


Figure 4.9: Node Ordering for Pyramid Elements (pyramid5, pyramid13, pyramid14).

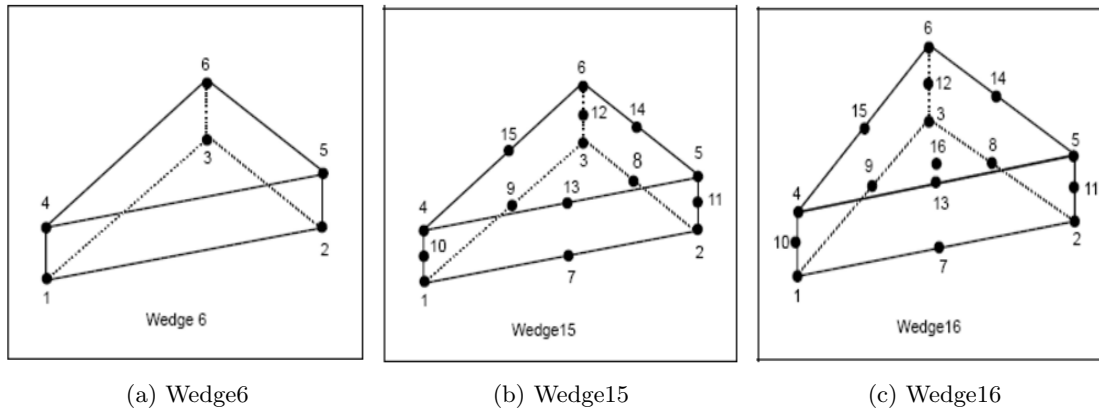


Figure 4.10: Node Ordering for Wedge Elements.

### 4.8.3 Element Attributes

*API Functions:* `ex_put_elem_attr`, `ex_get_elem_attr`

Element attributes are optional floating point numbers that can be assigned to each element. Every element in an element block must have the same number of attributes (as specified in the element block parameters) but the attributes may vary among elements within the block. The length of the attributes array is thus the product of the number of attributes per element and the number of elements in the element block. Table 4.1 lists the standard attributes for the given element types.

Element Type	Attributes
CIRCLE	R
SPHERE	R
TRUSS	A
BEAM	2D: A, I, J 3D: A, $I_1$ , $I_2$ , J, $V_1$ , $V_2$ , $V_3$
TRIANGLE	
QUAD	
SHELL	T
TETRA	
PYRAMID	
WEDGE	
HEX	

Table 4.1: Standard Element Types and Attributes

## 4.9 Node Sets

Node sets provide a means to reference a group of nodes with a single ID. Node sets may be used to specify load or boundary conditions, or to identify nodes for a special output

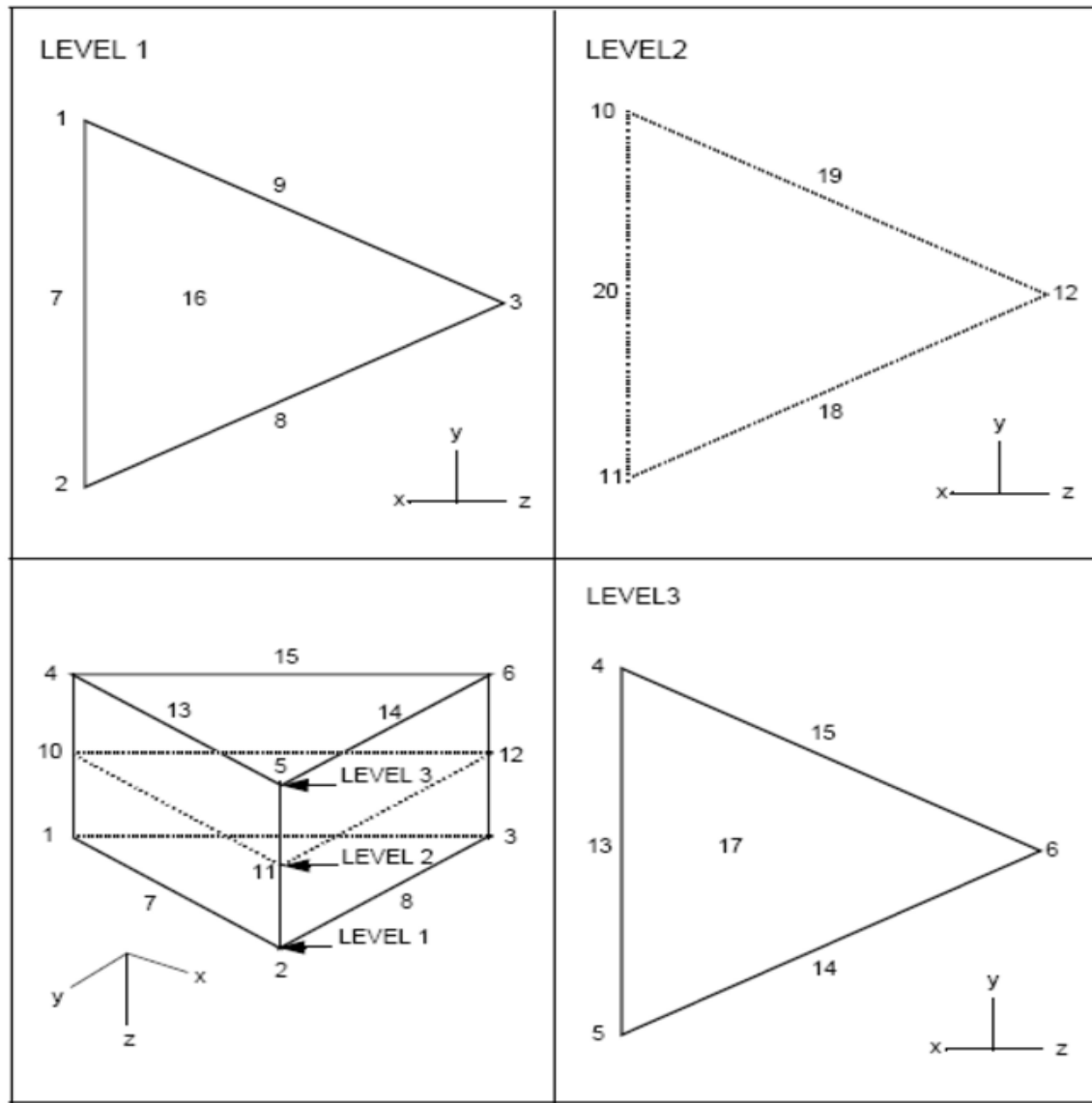


Figure 4.11: Node Ordering for Wedge Elements (Wedge20).

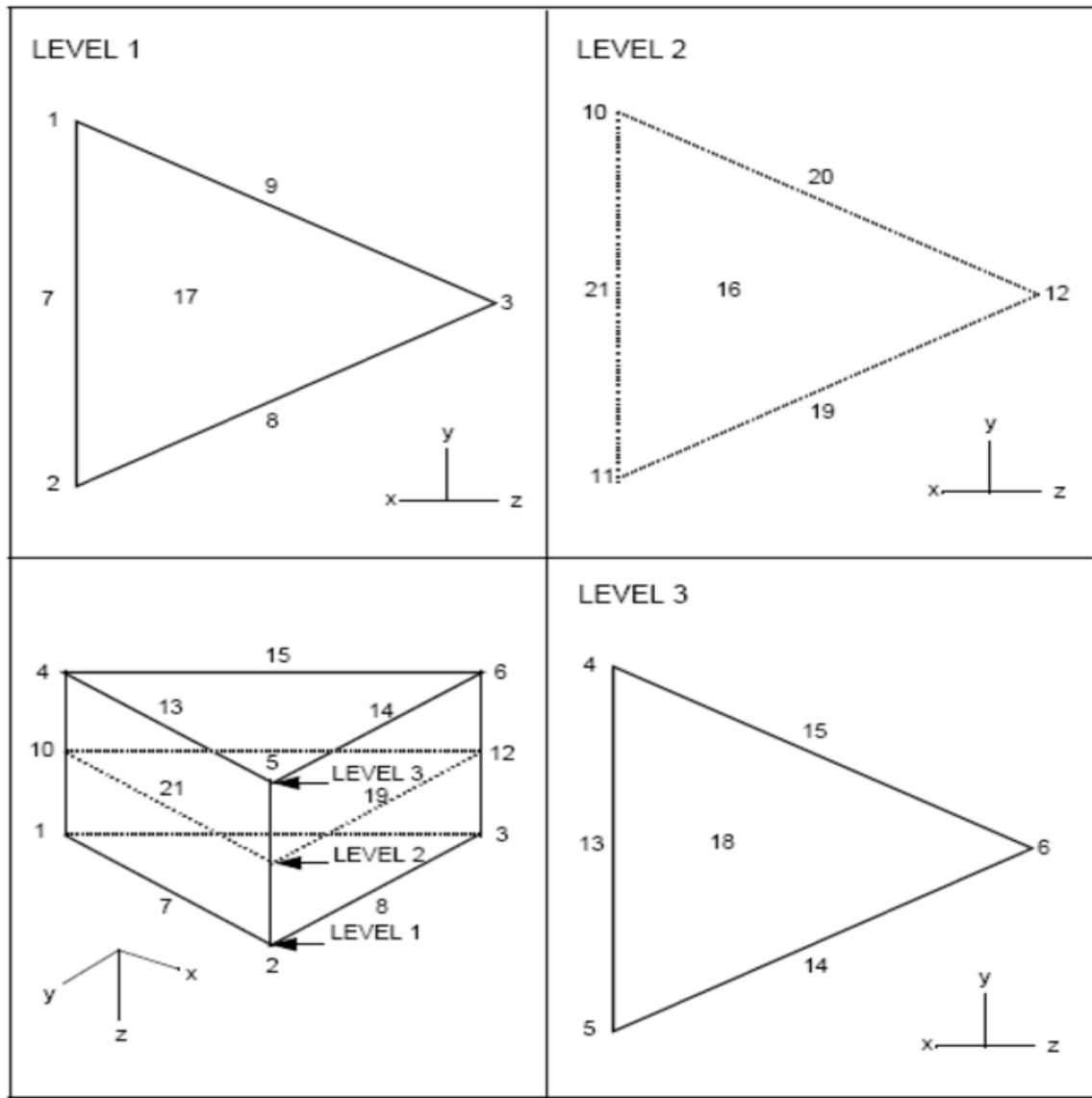


Figure 4.12: Node Ordering for Wedge Elements (Wedge21).



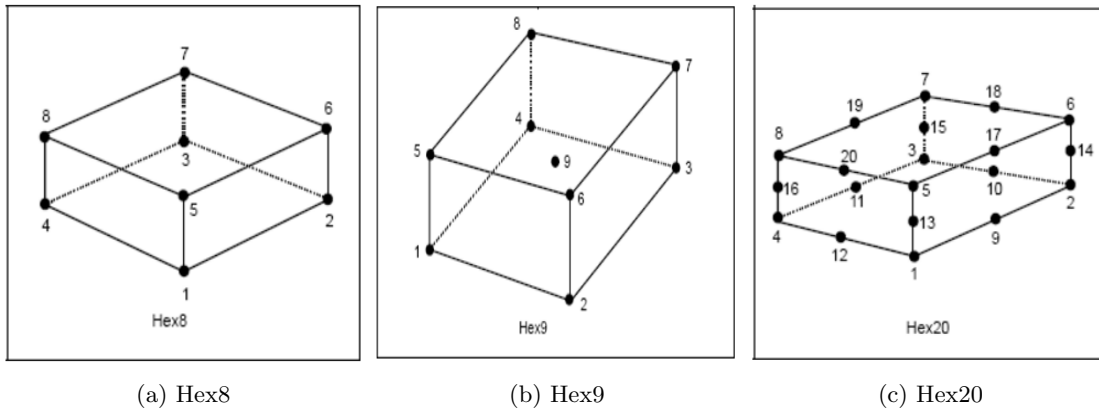


Figure 4.13: Node Ordering for Hexahedral Elements.

request. A particular node may appear in any number of node sets, but may be in a single node set only once. (This restriction is not checked by EXODUS routines.) Node sets may be accessed individually (using node set parameters, node set node list, and node set distribution factors) or in a concatenated format (described in Section 3.10 on page 11). The node sets data are stored identically in the data file regardless of which method (individual or concatenated) was used to output them.

### 4.9.1 Node Set Parameters

*API Functions:* `ex_put_node_set_param`, `ex_get_node_set_param`, `ex_get_node_set_ids`

The following parameters define each node set:

- node set ID – a unique positive integer that identifies the node set.
- Number of nodes – the number of nodes in the node set.
- Number of node set distribution factors – this should be zero if there are no distribution factors for the node set. If there are any distribution factors, this number must equal the number of nodes in the node set since the factors are assigned at each node. Refer to the discussion of distribution factors below.

### 4.9.2 Node Set Node List

*API Functions:* `ex_put_node_set`, `ex_get_node_set`

This is an integer list of all the nodes in the node set. Internal node IDs (see Section 4.6) must be used in this list.

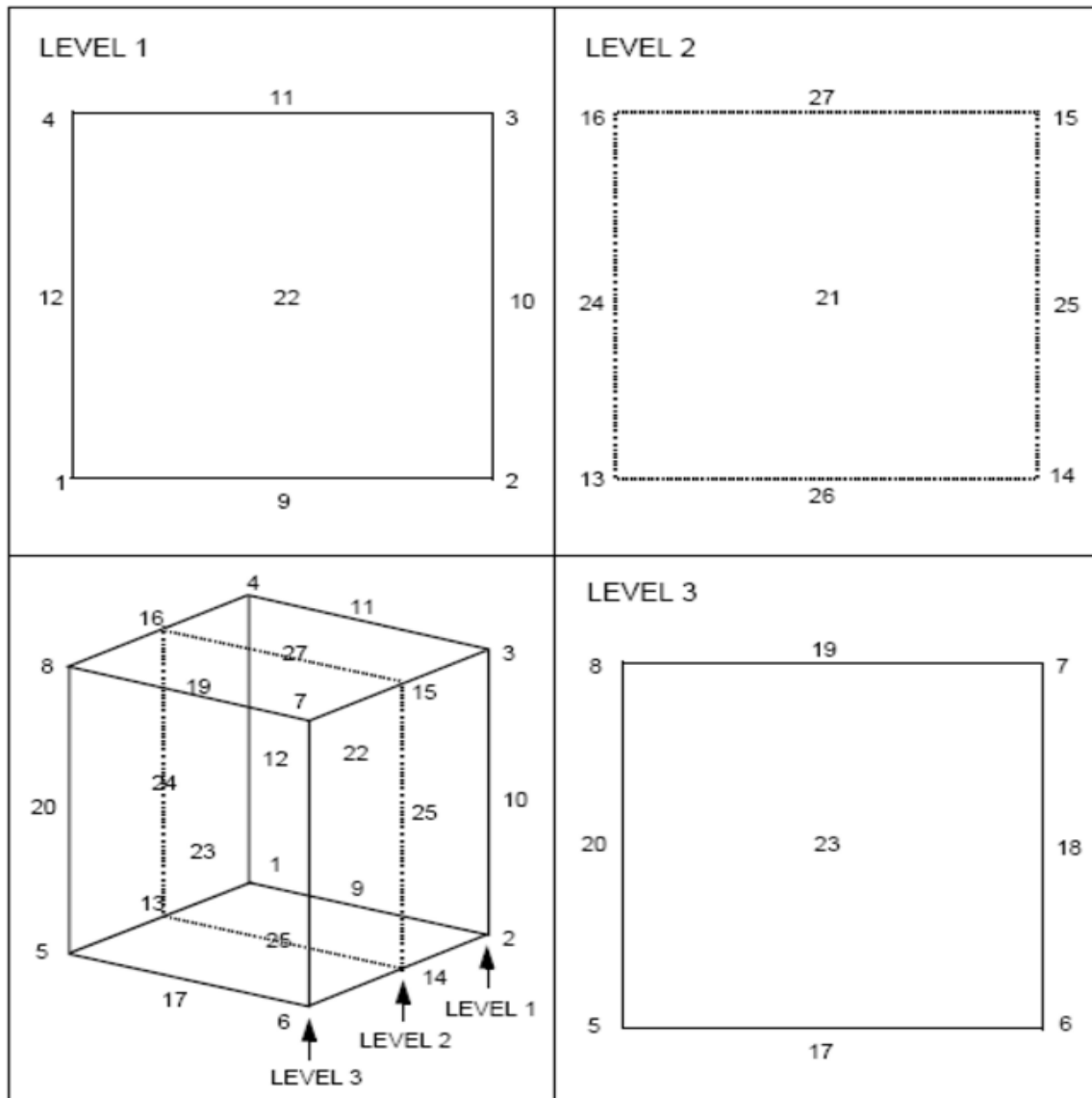


Figure 4.14: Node Ordering for Hexahedral Elements (Hex27).

### 4.9.3 Node Set Distribution Factors

*API Functions:* `ex_put_node_set_dist_fact`, `ex_get_node_set_dist_fact`

This is an optional list of floating point factors associated with the nodes in a node set. These data may be used as multipliers on applied loads. If distribution factors are stored, each entry in this list is associated with the corresponding entry in the node set node list.

## 4.10 Concatenated Node Sets

*API Functions:* `ex_put_concat_node_sets`, `ex_get_concat_node_sets`

Concatenated node sets provide a means of writing/reading all node sets with one function call. This is more efficient because it avoids some I/O overhead, particularly when considering the intricacies of the `NetCDF` library. (Refer to Appendix A for a discussion of efficiency concerns.) This is accomplished with the following lists:

- Node sets IDs – list (of length number of node sets) of unique integer node set ID's. The  $i^{\text{th}}$  entry in this list specifies the ID of the  $i^{\text{th}}$  node set.
- Node sets node counts – list (of length number of node sets) of counts of nodes for each node set. Thus, the  $i^{\text{th}}$  entry in this list specifies the number of nodes in the  $i^{\text{th}}$  node set.
- Node sets distribution factors counts – list (of length number of node sets) of counts of distribution factors for each node set. The  $i^{\text{th}}$  entry in this list specifies the number of distribution factors in the  $i^{\text{th}}$  node set.
- Node sets node pointers – list (of length number of node sets) of indices which are pointers into the node sets node list locating the first node of each node set. The  $i^{\text{th}}$  entry in this list is an index in the node sets node list where the first node of the  $i^{\text{th}}$  node set can be located.
- Node sets distribution factors pointers – list (of length number of node sets) of indices which are pointers into the node sets distribution factors list locating the first factor of each node set. The  $i^{\text{th}}$  entry in this list is an index in the node sets distribution factors list where the first factor of the  $i^{\text{th}}$  node set can be located.
- Node sets node list – concatenated integer list of the nodes in all the node sets. Internal node IDs (see Section 4.6) must be used in this list. The node sets node pointers and node sets node counts are used to find the first node and the number of nodes in a particular node set.
- Node sets distribution factors list – concatenated list of the (floating point) distribution factors in all the node sets. The node sets distribution factors pointers and node sets distribution factors counts are used to find the first factor and the number of factors in a particular node set.

To clarify the use of these lists, refer to the coding examples in Section 4.2.25 and Section 4.2.26.

## 4.11 Side Sets

Side sets provide a second means of applying load and boundary conditions to a model. Unlike node sets, side sets are related to specified sides of elements rather than simply a list of nodes. For example, a pressure load must be associated with an element edge (in 2-d) or face (in 3-d) in order to apply it properly. Each side in a side set is defined by an element number and a local edge (for 2-d elements) or face (for 3-d elements) number. The local number of the edge or face of interest must conform to the conventions as illustrated in Figure 4.15.

In this figure, side set side numbers are enclosed in boxes; only the essential node numbers to describe the element topology are shown. A side set may contain sides of differing types of elements that are contained in different element blocks. For instance, a single side set may contain faces of WEDGE elements, HEX elements, and TETRA elements.

### 4.11.1 Side Set Parameters

*API Functions:* `ex_put_side_set_param`, `ex_get_side_set_param`, `ex_get_side_set_ids`

The following parameters define each side set:

- side set ID – a unique positive integer that identifies the side set.
- Number of sides – the number of sides in the side set.
- Number of side set distribution factors – this should be zero if there are no distribution factors for the side set. If there are any distribution factors, they are assigned at the nodes on the sides of the side set. Refer to the discussion of distribution factors below.

### 4.11.2 Side Set Element List

*API Functions:* `ex_put_side_set`, `ex_get_side_set`

This is an integer list of all the elements in the side set. Internal element IDs (see Section 4.6) must be used in this list.

### 4.11.3 Side Set Side List

*API Functions:* `ex_put_side_set`, `ex_get_side_set`

This is an integer list of all the sides in the side set. This list contains the local edge (for 2-d elements) or face (for 3-d elements) numbers following the conventions specified in Figure 5.

### 4.11.4 Side Set Node List

*API Functions:* `ex_get_side_set_node_list`

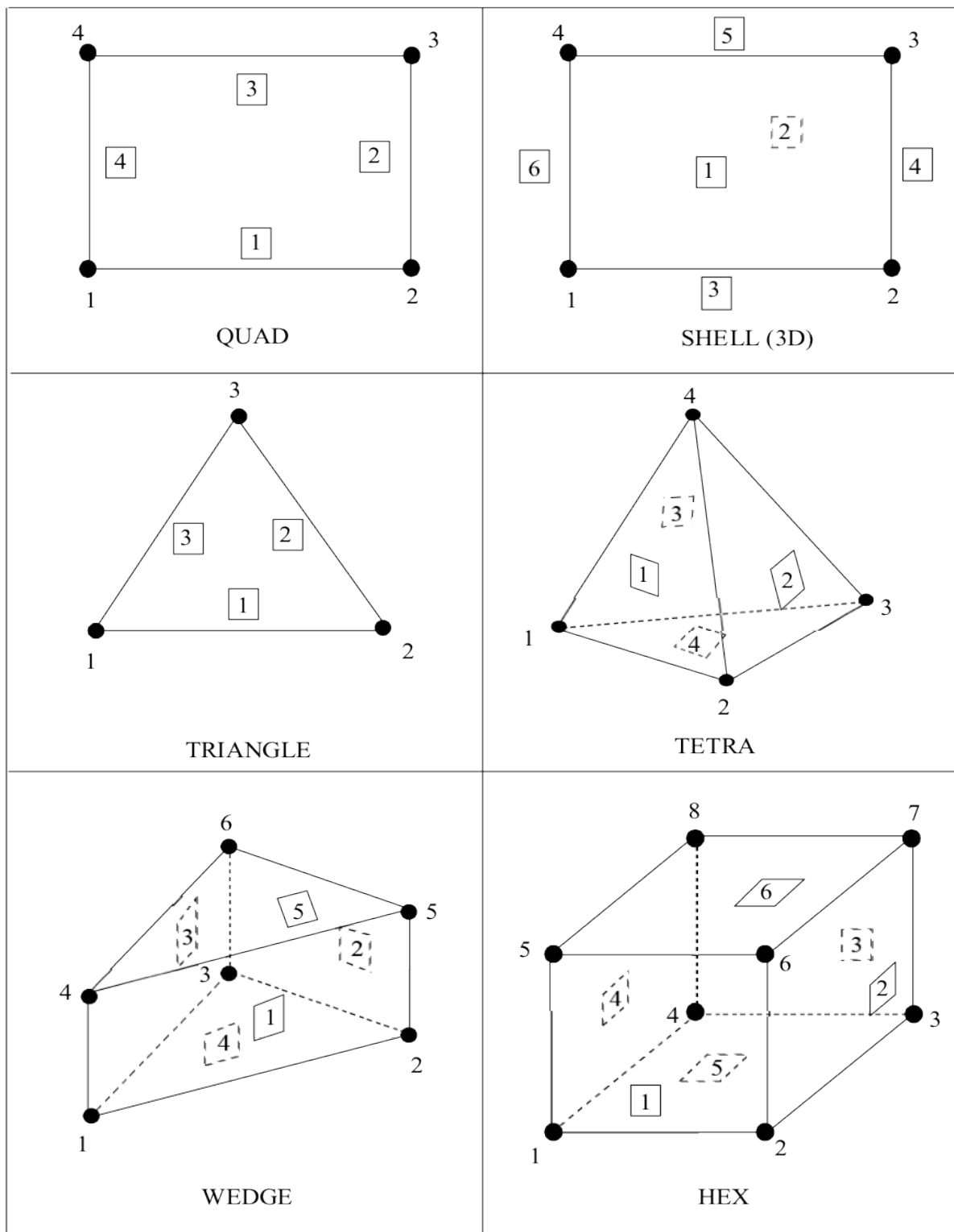


Figure 4.15: Side Set Side Numbering.

It is important to note that the nodes on a side set are not explicitly stored in the data file, but can be extracted from the element numbers in the side set element list, local side numbers in the side set side list, and the element connectivity array. The node IDs that are output are internal node numbers (see Section 4.5). They are extracted according to the following conventions:

1. All nodes for the first side (defined by the first element in the side set element list and the first side in the side set side list) are output before the nodes for the second side. There is no attempt to consolidate nodes; if a node is attached to four different faces, then the same node number will be output four times – once each time the node is encountered when progressing along the side list.
2. The nodes for a single face (or edge) are ordered to assist an application code in determining an “outward” direction. Thus, the node list for a face of a 3-d element proceeds around the face so that the outward normal follows the right-hand rule. The node list for an edge of a 2-d element proceeds such that if the right hand is placed in the plane of the element palm down, thumb extended with the index (and other fingers) pointing from one node to the next in the list, the thumb points to the inside of the element. This node ordering is detailed in Table 4.2 on page 30
3. The nodes required for a first-order element are output first, followed by the nodes of a higher ordered element. Table 4.2 lists the nodes for first-order elements. Refer to the node orderings shown in Figures 4.2 to 4.14 for the additional nodes on higher-order elements. If a face has a mid-face node, it is listed last following all mid-edge nodes. For example, the node ordering for side 1 of the `hex27` element is 1,2,6,5,9,14,17,13,26

#### 4.11.5 Side Set Node Count List

*API Functions:* `ex_get_side_set_node_list`

The length of the side set node count list is the length of the side set element list. For each entry in the side set element list, there is an entry in the side set side list, designating a local side number. The corresponding entry in the side set node count list is the number of nodes which define the particular side. In conjunction with the side set node list, this node count array provides an unambiguous nodal description of the side set.

#### 4.11.6 Side Set Distribution Factors

*API Functions:* `ex_put_side_set_dist_fact` , `ex_get_side_set_dist_fact`

This is an optional list of floating point factors associated with the nodes on a side set. These data may be used for uneven application of load or boundary conditions. Because distribution factors are assigned at the nodes, application codes that utilize these factors must read the side set node list. The distribution factors must be stored/accessed in the same order as the nodes in the side set node list; thus, the ordering conventions described above apply.

Element Type	Side #	Node Order	
QUAD (2D)	1	1, 2,	5
	2	2, 3,	6
	3	3, 4,	7
	4	4, 1,	8
SHELL (Edges)	1	1, 2, 3, 4,	5, 6, 7, 8, 9
	2	1, 4, 3, 2,	8, 7, 6, 5, 9
	3	1, 2,	5
	4	2, 3,	6
	5	3, 4,	7
	6	4, 1,	8
TRIANGLE (2D)	1	1, 2,	4
	2	2, 3,	5
	3	3, 1,	6
TRIANGLE (Shell)	1	1, 2, 3,	4, 5, 6
	2	1, 3, 2,	6, 5, 4
	3	1, 2,	4
	4	2, 3,	5
	5	3, 1,	6
TETRA	1	1, 2, 4,	5, 9, 8
	2	2, 3, 4,	6, 10, 9
	3	1, 4, 3,	8, 10, 7
	4	1, 3, 2,	7, 6, 5
WEDGE	1	1, 2, 5, 4,	7, 11, 13, 10
	2	2, 3, 6, 5,	8, 12, 14, 11
	3	1, 4, 6, 3,	10, 15, 12, 9
	4	1, 3, 2,	9, 8, 7
	5	4, 5, 6,	13, 14, 15
HEX	1	1, 2, 6, 5,	9, 14, 17, 13, 26
	2	2, 3, 7, 6,	10, 15, 18, 14, 25
	3	3, 4, 8, 7,	11, 16, 19, 15, 27
	4	1, 5, 8, 4,	13, 20, 16, 12, 24
	5	1, 4, 3, 2,	12, 11, 10, 9, 22
	6	5, 6, 7, 8,	17, 18, 19, 20, 23
PYRAMID	1	1, 2, 5,	6, 11, 10
	2	2, 3, 5,	7, 12, 11
	3	3, 4, 5,	8, 13, 12
	4	4, 1, 5,	9, 10, 13
	5	1, 4, 3, 2,	9, 8, 7, 6

Table 4.2: Sideset Node Ordering

## 4.12 Concatenated Side Sets

*API Functions:* `ex_put_concat_side_sets`, `ex_get_concat_side_sets`

Concatenated side sets provide a means of writing / reading all side sets with one function call. This is more efficient because it avoids some I/O overhead, particularly when considering the intricacies of the `NetCDF` library. This is accomplished with the following lists:

- Side sets IDs – list (of length number of side sets) of unique positive integer side set ID's. The  $i$ th entry in this list specifies the ID of the  $i$ th side set.
- Side sets side counts – list (of length number of side sets) of counts of sides for each side set. Thus, the  $i$ th entry in this list specifies the number of sides in the  $i$ th node set. This also defines the number of elements in each side set.
- Side sets distribution factors counts – list (of length number of side sets) of counts of distribution factors for each side set. The  $i$ th entry in this list specifies the number of distribution factors in the  $i$ th side set.
- Side sets side pointers – list (of length number of side sets) of indices which are pointers into the side sets element list (and side list) locating the first element (or side) of each side set. The  $i$ th entry in this list is an index in the side sets element list (and side list) where the first element (or side) of the  $i$ th side set can be located.
- Side sets distribution factors pointers – list (of length number of side sets) of indices which are pointers into the side sets distribution factors list locating the first factor of each side set. The  $i$ th entry in this list is an index in the side sets distribution factors list where the first factor of the  $i$ th side set can be located.
- Side sets element list – concatenated integer list of the elements in all the side sets. Internal element IDs (see Section 4.6) must be used in this list. The side sets side pointers and side sets side counts are used to find the first element and the number of elements in a particular side set.
- Side sets side list – concatenated integer list of the sides in all the side sets. The side sets side pointers and side sets side counts are used to find the first side and the number of sides in a particular side set.
- Side sets distribution factors list – concatenated list of the (floating point) distribution factors in all the side sets. The side sets distribution factors pointers and side sets distribution factors counts are used to find the first factor and the number of factors in a particular side set.

### 4.12.1 Object Properties

Certain EXODUS objects (currently element blocks, node sets, and side sets) can be given integer properties, providing the following capabilities:

1. assign a specific integer value to a named property of an object.
2. tag objects as members of a group. For example element blocks 1 and 3 and side sets 1 and 2 could be put in a group named "TOP."



Name	EB 1	EB 2	EB 3	NS 1	SS 1	SS 2
ID	1	0	1	0	1	1
TOP	1	1	0	1	1	0
LEFT	0	0	1	NULL	NULL	NULL
STEEL	0	1	1	NULL	NULL	NULL
COPPER	1	1	0	NULL	NULL	NULL

Table 4.3: Example Property Table.

This functionality is illustrated in Table 4.3 which contains the property values of a sample EXODUS file with three element blocks, one node set, and two side sets. Note that an application code can define properties to be valid for only specified object types. In this example, “STEEL” and “COPPER” are valid for all element blocks but are not defined for node sets and side sets.

Interpretation of the integer values of the properties is left to the application codes, but in general, a nonzero positive value means the object has the named property (or is in the named group); a zero means the object does not have the named property (or is not in the named group). Thus, element block 1 has an ID of 10 (1 is a counter internal to the data base; an application code accesses the element block using the ID), node set 1 has an ID of 100, etc. The group “TOP” includes element block 1, element block 3, and side sets 1 and 2.

#### 4.12.2 Property Values

*API Functions:* `ex_put_prop`, `ex_get_prop`, `ex_put_prop_array`, `ex_get_prop_array`

Valid values for the properties are positive integers and zero. Property values are stored in arrays in the data file but can be written / read individually given an object type (i.e., element block, node set, or side set), object ID, and property name or as an array given an object type and property name. If accessed as an array, the order of the values in the array must correspond to the order in which the element blocks, node sets, or side sets were introduced into the file. For instance, if the parameters for element block with ID 20 were written to a file, and then parameters for element block with ID 10, followed by the parameters for element block with ID 30, the first, second, and third elements in the property array would correspond to element block 20, element block 10, and element block 30, respectively. This order can be determined with a call to `ex_get_elem_blk_ids` which returns an array of element block IDs in the order that the corresponding element blocks were introduced to the data file.

### 4.13 Results Parameters

*API Functions:* `ex_put_variable_param`, `ex_get_variable_param`

The number of each type of results variables (element, nodal, and global) is specified only once, and cannot change through time.

#### 4.13.1 Results Names

*API Functions:* `ex_put_variable_names`, `ex_get_variable_names`

Associated with each results variable is a unique name of length `MAX_STR_LENGTH`.

## 4.14 Results Data

An integer output time step number (beginning with 1) is used as an index into the results variables written to or read from an EXODUS file. It is a counter of the number of “data planes” that have been written to the file. The maximum time step number (i.e., the number of time steps that have been written) is available via a call to the database inquire function ( See Section 5.1.10). For each output time step, the following information is stored.

### 4.14.1 Time Values

*API Functions:* `ex_put_time`, `ex_get_time`, `ex_get_all_times`

A floating point value must be stored for each time step to identify the “data plane.” Typically, this is the analysis time but can be any floating point variable that distinguishes the time steps. For instance, for a modal analysis, the natural frequency for each mode may be stored as a “time value” to discriminate the different sets of eigen vectors. The only restriction on the time values is that they must monotonically increase.

### 4.14.2 Global Results

*API Functions:* `ex_put_glob_vars`, `ex_get_glob_vars`, `ex_get_glob_var_time`

This object contains the floating point global data for the time step. The length of the array is the number of global variables, as specified in the results parameters.

### 4.14.3 Nodal Results

*API Functions:* `ex_put_nodal_var`, `ex_get_nodal_var`, `ex_get_nodal_var_time`

This object contains the floating point nodal data for the time step. The size of the array is the number of nodes, as specified in the global parameters, times the number of nodal variables.

### 4.14.4 Element Results

*API Functions:* `ex_put_elem_var`, `ex_get_elem_var`, `ex_get_elem_var_time`

Element variables are output for a given element block and a given element variable. Thus, at each time step, up to  $m$  element variable objects (where  $m$  is the product of the number of element blocks and the number of element variables) may be stored. However, since not all element variables must be output for all element blocks (see the next section),  $m$  is the *maximum* number of element variable objects. The actual number of objects stored is the number of unique combinations of element variable index and element block ID passed to `ex_put_elem_var` or the number of non-zero

	Elem Block 1	Elem Block 2	Elem Block 3	Elem Block 4
Elem Var 1	1	1	1	1
Elem Var 2	1	0	0	1

Table 4.4: Element Variable Truth Table

entries in the element variable truth table (if it is used). The length of each object is the number of elements in the given element block.

## 4.15 Element Variable Truth Table

*API Functions:* `ex_put_elem_var_tab`, `ex_get_elem_var_tab`

Because some element variables are not applicable (and thus not computed by a simulation code) for all element types, the element variable truth table is an optional mechanism for specifying whether a particular element result is output for the elements in a particular element block. For example, hydrostatic stress may be an output result for the elements in element block 3, but not those in element block 6.

It is helpful to describe the element variable truth table as a two-dimensional array, as shown in Table 4.4, each row of the array is associated with an element variable; each column of the array is associated with an element block. If a datum in the truth table is zero ( $\text{table}(i, j) = 0$ ), then no results are output for the  $i^{\text{th}}$  element variable for the  $j^{\text{th}}$  element block. A nonzero entry indicates that the appropriate result will be output. In this example, element variable 1 will be stored for all element blocks; element variable 2 will be stored for element blocks 1 and 4; and element variable 3 will be stored for element blocks 3 and 4. The table is stored such that the variable index cycles faster than the block index.

## Chapter 5

# Application Programming Interface (API)

EXODUS files can be written and read by application codes written in C, C++, or Fortran via calls to functions in the application programming interface (API). Functions within the API are categorized as data file utilities, model description functions, or results data functions.

In general, the following pattern is followed for writing data objects to a file:

1. create the file with `ex_create`;
2. write out global parameters to the file using `ex_put_init`;
3. write out specific data object parameters; for example, put out element block parameters with `ex_put_elem_block`;
4. write out the data object; for example, put out the connectivity for an element block with `ex_put_elem_conn`;
5. close the file with `ex_close`.

Steps 3 and 4 are repeated within this pattern for each data object (i.e., nodal coordinates, element blocks, node sets, side sets, results variables, etc.). For some data object types, steps 3 and 4 are combined in a single call. For instance, `ex_put_qa` writes out the parameters (number of QA records) as well as the data object itself (the QA records). During the database writing process, there are a few order dependencies (e.g., an element block must be written before element variables for that element block are written) which are documented in the description of each library function.

The invocation of the EXODUS API functions for reading data is order independent, providing random read access. The following steps are typically used for reading data:

1. open the file with `ex_open`;
2. read the global parameters for dimensioning purposes with `ex_get_init`;
3. read specific data object parameters; for example, read node set parameters with `ex_get_node_set_param`;
4. read the data object; for example, read the node set node list with `ex_get_node_set`;

5. close the file with `ex_close`.

Again, steps 3 and 4 are repeated for each object. For some object parameters, step 3 may be accomplished with a call to `ex_inquire` to inquire the size of certain objects.

In developing applications using the EXODUS API, the following points may prove beneficial:

- All functions that write objects to the database begin with `ex_put_`; functions that read objects from the database begin with `ex_get_`.
- Function arguments are classified as readable [in], writable [out], or both [inout]. Readable arguments are not modified by the API routines; writable arguments are modified; read-write arguments may be either depending on the value of the argument.
- All application codes which use the EXODUS API must include the file ‘exodusII.h’ for C. This file defines constants that are used (1) as arguments to the API routines, (2) to set global parameters such as maximum string length and database version, and (3) as error condition or function return values.
- Throughout this section, sample code segments have been included to aid the application developer in using the API routines. These segments are not complete and there has been no attempt to include all calling sequence dependencies within them.
- Because 2-dimensional arrays cannot be statically dimensioned, either dynamic dimensioning or user indexing is required. Most of the sample code segments utilize user indexing within 1-dimensional arrays even though the variables are logically 2-dimensional.
- There are many NetCDF utilities that prove useful. `ncdump`, which converts a binary NetCDF file to a readable ASCII version of the file, is the most notable.
- Because NetCDF buffers I/O, it is important to flush all buffers with `ex_update` when debugging an application that produces an EXODUS file.

## 5.1 Data File Utilities

This section describes data file utility functions for creating / opening a file, initializing a file with global parameters, reading / writing information text, inquiring on parameters stored in the data file, and error reporting.

### 5.1.1 Create EXODUS File

The function `ex_create` creates a new EXODUS file and returns an ID that can subsequently be used to refer to the file.

All floating point values in an EXODUS file are stored as either 4-byte (“float”) or 8-byte (“double”) numbers; no mixing of 4- and 8-byte numbers in a single file is allowed. An application code can compute either 4- or 8-byte values and can designate that the values be stored in the EXODUS file as either 4- or 8-byte numbers; conversion between the 4- and 8-byte values is performed automatically by the API routines. Thus, there are four possible combinations of compute word size and storage (or I/O) word size.

In case of an error, `ex_create` returns a negative number. Possible causes of errors include:

- Passing a file name that includes a directory that does not exist.
- Specifying a file name of a file that exists and also specifying a no clobber option.
- Attempting to create a file in a directory without permission to create files there.
- Passing an invalid file clobber mode.

```
int ex_create (char *path,
              int mode,
              int *comp_ws,
              int *io_ws)
```

**char\* path [in]**

The file name of the new EXODUS file. This can be given as either an absolute path name (from the root of the file system) or a relative path name (from the current directory).

**int mode [in]**

Mode. Use one of the following predefined constants:

**EX\_NOCLOBBER** To create the new file only if the given file name does not refer to a file that already exists.

**EX\_CLOBBER** To create the new file, regardless of whether a file with the same name already exists. If a file with the same name does exist, its contents will be erased.

**EX\_LARGE\_MODEL** To create a model that can store individual datasets larger than 2 gigabytes. This modifies the internal storage used by exodusII and also puts the underlying **NetCDF** file into the “64-bit offset” mode. See Appendix ?? for more details on this mode.<sup>1</sup>

**EX\_NORMAL\_MODEL** Create a standard model.

**EX\_NETCDF4** To create a model using the **HDF5**-based **NetCDF-4** output. (Future capability)<sup>2</sup>

**EX\_NOSHARE** Do not open the underlying **NetCDF** file in “share” mode. See the **NetCDF** documentation for more details.

**EX\_SHARE** Do open the underlying **NetCDF** file in “share” mode. See the **NetCDF** documentation for more details.

**int\* comp\_ws [inout]**

The word size in bytes (0, 4 or 8) of the floating point variables used in the application program. If 0 (zero) is passed, the default `sizeof(float)` will be used and returned in this variable. **WARNING:** all EXODUS functions requiring floats must be passed floats declared with this passed in or returned compute word size (4 or 8).

**int\* io\_ws [in]**

The word size in bytes (4 or 8) of the floating point data as they are to be stored in the EXODUS file.

---

<sup>1</sup>A “large model” file will also be created if the environment variable **EXODUS\_LARGE\_MODEL** is defined in the users environment. A message will be printed to standard output if this environment variable is found.

<sup>2</sup>**NetCDF-4** is currently in beta mode; however, it will be used for ExodusII when available, so this mode is being defined here for future completeness. An **HDF5**-based **NetCDF-4** file will also be created if the environment variable **EXODUS\_NETCDF4** is defined in the users environment. A message will be printed to standard output if this environment variable is found.

The following code segment creates an EXODUS file called *test.exo*:

```

1 #include "exodusII.h"
2 int CPU_word_size, IO_word_size, exoid;
3 CPU_word_size = sizeof(float); /* use float or double */
4 IO_word_size = 8; /* store variables as doubles */
5
6 /* create \exo{} file */
7 exoid = ex_create ("test.exo"      /* filename path */
8                  EX_CLOBBER,      /* create mode */
9                  &CPU_word_size, /* CPU float word size in bytes */
10                 &IO_word_size); /* I/O float word size in bytes */

```

### 5.1.2 Open EXODUS File

The function `ex_open` opens an existing EXODUS file and returns an ID that can subsequently be used to refer to the file, the word size of the floating point values stored in the file, and the version of the EXODUS database (returned as a “float”, regardless of the compute or I/O word size). Multiple files may be “open” simultaneously.

In case of an error, `ex_open` returns a negative number. Possible causes of errors include:

- The specified file does not exist.
- The mode specified is something other than the predefined constant `EX_READ` or `EX_WRITE`.
- Database version is earlier than 2.0.

```

int ex_open (char *path,
            int mode,
            int *comp_ws,
            int *io_ws,
            float *version)

```

**char\* path [in]**

The file name of the EXODUS file. This can be given as either an absolute path name (from the root of the file system) or a relative path name (from the current directory).

**int mode [in]**

Access mode. Use one of the following predefined constants:

**EX\_READ** To open the file just for reading.

**EX\_WRITE** To open the file for writing and reading.

**int\* comp\_ws [inout]**

The word size in bytes (0, 4 or 8) of the floating point variables used in the application program. If 0 (zero) is passed, the default size of floating point values for the machine will be used and returned in this variable. **WARNING:** all EXODUS functions requiring reals must be passed reals declared with this passed in or returned compute word size (4 or 8).

**int\* io\_ws [inout]**

The word size in bytes (0, 4 or 8) of the floating point data as they are stored in the EXODUS file. If the word size does not match the word size of data stored in the file, a fatal error is returned. If this argument is 0, the word size of the floating point data already stored in the file is returned.

**float\* version [out]**

Returned EXODUS database version number. The current version is 4.72

The following opens an EXODUS file named *test.exo* for read only, using default settings for compute and I/O word sizes:

```

1 #include "exodusII.h"
2 int CPU_word_size, IO_word_size, exoid;
3 float version;
4
5 CPU_word_size = sizeof(float); /* float or double */
6 IO_word_size = 0;             /* use what is stored in file */
7
8 /* open \exo{} files */
9 exoid = ex_open ("test.exo",    /* filename path */
10                EX_READ,        /* access mode = READ */
11                &CPU_word_size, /* CPU word size */
12                &IO_word_size,  /* IO word size */
13                &version);      /* ExodusII library version */

```

### 5.1.3 Close EXODUS File

The function `ex_close` updates and then closes an open EXODUS file.

In case of an error, `ex_close` returns a negative number; a warning will return a positive number. Possible causes of errors include:

- data file not properly opened with call to `ex_create` or `ex_open`

**int ex\_close (int exoid)**

**int exoid [in]**

EXODUS file ID returned from a previous call to `ex_create` or `ex_open`.

The following code segment closes an open EXODUS file:

```

1 int error, exoid;
2 error = ex_close (exoid);

```

### 5.1.4 Write Initialization Parameters

The function `ex_put_init` writes the initialization parameters to the EXODUS file. This function must be called once (and only once) before writing any data to the file.

In case of an error, `ex_put_init` returns a negative number; a warning will return a positive number. Possible causes of errors include:



- data file not properly opened with call to `ex_create` or `ex_open`
- data file opened for read only.
- this routine has been called previously.

```
int ex_put_init (int exoid,
                char *title,
                int num_dim,
                int num_nodes,
                int num_elem,
                int num_elem_blk,
                int num_node_sets,
                int num_side_sets)
```

`int exoid [in]`

EXODUS file ID returned from a previous call to `ex_create` or `ex_open`.

`char* titletitle [in]`

Database title. Maximum length is `MAX_LINE_LENGTH`.

`int num_dim [in]`

The dimensionality of the database. This is the number of coordinates per node.

`int num_nodes [in]`

The number of nodal points.

`int num_elem [in]`

The number of elements.

`int num_elem_blk [in]`

The number of element blocks.

`int num_node_sets [in]`

The number of node sets.

`int num_side_sets [in]`

The number of side sets.

The following code segment will initialize an open EXODUS file with the specified parameters:

```
1 int num_dim, num_nods, num_el, num_el_blk, num_ns, num_ss, error, exoid;
2
3 /* initialize file with parameters */
4 num_dim = 3; num_nods = 46; num_el = 5; num_el_blk = 5;
5 num_ns = 2; num_ss = 5;
6
7 error = ex_put_init (exoid, "This is the title", num_dim,
8                      num_nods, num_el, num_el_blk, num_ns, num_ss);
```

### 5.1.5 Read Initialization Parameters

The function `ex_get_init` reads the initialization parameters from an opened EXODUS file.

In case of an error, `ex_get_init` returns a negative number; a warning will return a positive number. Possible causes of errors include:

- data file not properly opened with call to `ex_create` or `ex_open`.

```
int ex_get_init (int exoid,
                char *title,
                int num_dim,
                int num_nodes,
                int num_elem,
                int num_elem_blk,
                int num_node_sets,
                int num_side_sets)
```

`int exoid [in]`

EXODUS file ID returned from a previous call to `ex_create` or `ex_open`.

`char* title [out]`

Returned database title. String length may be up to `MAX_LINE_LENGTH` bytes.

`int* num_dim [out]`

Returned dimensionality of the database. This is the number of coordinates per node.

`int* num_nodes [out]`

Returned number of nodal points.

`int* num_elem [out]`

Returned number of elements.

`int* num_elem_blk [out]`

Returned number of element blocks.

`int* num_node_sets [out]`

Returned number of node sets.

`int* num_side_sets [out]`

Returned number of side sets.

The following code segment will read the initialization parameters from the open EXODUS file:

```
1 #include "exodusII.h"
2 int num_dim, num_nodes, num_elem, num_elem_blk,
3   num_node_sets, num_side_sets, error, exoid;
4
```

```

5 char title[MAX_LINE_LENGTH+1];
6
7 /* read database parameters */
8 error = ex_get_init (exoid, title, &num_dim, &num_nodes,
9                      &num_elem, &num_elem_blk, &num_node_sets, &num_side_sets);

```

### 5.1.6 Write Quality Assurance (QA) Records

The function `ex_put_qa` writes the QA records to the database. Each QA record contains four `MAX_STR_LENGTH`-byte character strings. The character strings are:

- the analysis code name
- the analysis code QA descriptor
- the analysis date
- the analysis time

In case of an error, `ex_put_qa` returns a negative number; a warning will return a positive number. Possible causes of errors include:

- data file not properly opened with call to `ex_create` or `ex_open`
- data file opened for read only.
- QA records already exist in file.

```

int ex_put_qa (int exoid,
               int num_qa_records,
               char *qa_record[][4])

```

`int exoid [in]`

EXODUS file ID returned from a previous call to `ex_create` or `ex_open`.

`int num_qa_records [in]`

The number of QA records.

`char* qa_record [in]`

Array containing the QA records.

The following code segment will write out two QA records:

```

1 #include "exodusII.h"
2 int num_qa_rec, error, exoid;
3 char *qa_record[2][4];
4
5 /* write QA records */
6 num_qa_rec = 2;
7

```

```

8  qa_record[0][0] = "TESTWT1";
9  qa_record[0][1] = "testwt1";
10 qa_record[0][2] = "07/07/93";
11 qa_record[0][3] = "15:41:33";
12 qa_record[1][0] = "FASTQ";
13 qa_record[1][1] = "fastq";
14 qa_record[1][2] = "07/07/93";
15 qa_record[1][3] = "16:41:33";
16
17 error = ex_put_qa (exoid, num_qa_rec, qa_record);

```

### 5.1.7 Read Quality Assurance (QA) Records

The function `ex_get_qa` reads the QA records from the database. Each QA record contains four `MAX_STR_LENGTH`-byte character strings. The character strings are:

- the analysis code name
- the analysis code QA descriptor
- the analysis date
- the analysis time

Memory must be allocated for the QA records before this call is made. The number of QA records can be determined by invoking `ex_inquire`.

In case of an error, `ex_get_qa` returns a negative number; a warning will return a positive number. Possible causes of errors include:

- data file not properly opened with call to `ex_create` or `ex_open`
- a warning value is returned if no QA records were stored.

```

int ex_get_qa (int exoid,
               char *qa_record[][4])

```

`int exoid [in]`  
EXODUS file ID returned from a previous call to `ex_create` or `ex_open`.

`char* qa_record [out]`  
Returned array containing the QA records.

The following will determine the number of QA records and read them from the open EXODUS file:

```

1  #include "exodusII.h"
2  int num_qa_rec, error, exoid
3  char *qa_record[MAX_QA_REC][4];
4
5  /* read QA records */
6  num_qa_rec = ex_inquire_int(exoid, EX_INQ_QA);

```

```

7
8 for (i=0; i<num_qa_rec; i++) {
9     for (j=0; j<4; j++)
10         qa_record[i][j] = (char *) calloc ((MAX_STR_LENGTH+1), sizeof(char));
11 }
12 error = ex_get_qa (exoid, qa_record);

```

### 5.1.8 Write Information Records

The function `ex_put_info` writes information records to the database. The records are MAX`LINE`LENGTH-character strings.

In case of an error, `ex_put_info` returns a negative number; a warning will return a positive number. Possible causes of errors include:

- data file not properly opened with call to `ex_create` or `ex_open`
- data file opened for read only.
- information records already exist in file.

```

int ex_put_info (int exoid,
                 int num_info,
                 char **info)

```

`int exoid [in]`

EXODUS file ID returned from a previous call to `ex_create` or `ex_open`.

`int num_info [in]`

The number of information records.

`char** info [in]`

Array containing the information records.

The following code will write out three information records to an open EXODUS file:

```

1 #include "exodusII.h"
2 int error, exoid, num_info;
3 char *info[3];
4
5 /* write information records */
6 num_info = 3;
7
8 info[0] = "This is the first information record.";
9 info[1] = "This is the second information record.";
10 info[2] = "This is the third information record.";
11
12 error = ex_put_info(exoid, num_info, info);

```

### 5.1.9 Read Information Records

The function `ex_get_info` reads information records from the database. The records are `MAX_LINE_LENGTH`-character strings. Memory must be allocated for the information records before this call is made. The number of records can be determined by invoking `ex_inquire` or `ex_inquire.int`.

In case of an error, `ex_get_info` returns a negative number; a warning will return a positive number. Possible causes of errors include:

- data file not properly opened with call to `ex_create` or `ex_open`
- a warning value is returned if no information records were stored.

```
int ex_get_info (int exoid,
                char **info)
```

```
int exoid [in]
```

EXODUS file ID returned from a previous call to `ex_create` or `ex_open`.

```
char** info [out]
```

Returned array containing the information records.

The following code segment will determine the number of information records and read them from an open EXODUS file:

```
1 #include "exodusII.h"
2 int error, exoid, num_info;
3 char *info[MAXINFO];
4
5 /* read information records */
6 num_info = ex_inquire_int (exoid, EX_INQ_INFO);
7 for (i=0; i < num_info; i++) {
8     info[i] = (char *) calloc ((MAX_LINE_LENGTH+1), sizeof(char));
9 }
10 error = ex_get_info (exoid, info);
```

### 5.1.10 Inquire EXODUS Parameters

The function `ex_inquire` is used to inquire values of certain data entities in an EXODUS file. Memory must be allocated for the returned values before this function is invoked.

In case of an error, `ex_inquire` returns a negative number; a warning will return a positive number. Possible causes of errors include:

- data file not properly opened with call to `ex_create` or `ex_open`.
- requested information not stored in the file.
- invalid request flag.

```
int ex_inquire (int exoid,
               ex_inquiry req_info,
               int *ret_int,
               float *ret_float,
               char *ret_char)
```

`int exoid [in]`

EXODUS file ID returned from a previous call to `ex_create` or `ex_open`.

`ex_inquiry req_info [in]`

A flag which designates what information is requested. It must be one of the following constants (predefined in the file *exodusII.h*):

EX_INQ_API_VERS	The EXODUS API version number is returned in <code>ret_float</code> and an “undotted” version number is returned in <code>ret_int</code> . The API version number reflects the release of the function library (i.e., function names, argument list, etc.). The current API version is 4.72 or 472 <sup>3</sup> .
EX_INQ_DB_VERS	The EXODUS database version number is returned in <code>ret_float</code> and an “undotted” version number is returned in <code>ret_int</code> . The database version number reflects the version of the library that was used to write the file pointed to by <code>exoid</code> . The current database version is 4.72 or 472.
EX_INQ_LIB_VERS	The EXODUS library version number is returned in <code>ret_float</code> and an “undotted” version number is returned in <code>ret_int</code> . The API library version number reflects the version number of the EXODUS library linked with this application. The current library version is 4.72 or 472.
EX_INQ_TITLE	The title stored in the database is returned in <code>ret_char</code> .
EX_INQ_DIM	The dimensionality, or number of coordinates per node (1, 2 or 3), of the database is returned in <code>ret_int</code> .
EX_INQ_NODES	The number of nodes is returned in <code>ret_int</code> .
EX_INQ_ELEM	The number of elements is returned in <code>ret_int</code> .
EX_INQ_ELEM_BLK	The number of element blocks is returned in <code>ret_int</code> .
EX_INQ_NODE_SETS	The number of node sets is returned in <code>ret_int</code> .
EX_INQ_NS_NODE_LEN	The length of the concatenated node sets node list is returned in <code>ret_int</code> .
EX_INQ_NS_DF_LEN	The length of the concatenated node sets distribution list is returned in <code>ret_int</code> .
EX_INQ_SIDE_SETS	The number of side sets is returned in <code>ret_int</code> .
EX_INQ_SS_ELEM_LEN	The length of the concatenated side sets element list is returned in <code>ret_int</code> .
EX_INQ_SS_DF_LEN	The length of the concatenated side sets distribution factor list is returned in <code>ret_int</code> .
EX_INQ_SS_NODE_LEN	The aggregate length of all of the side sets node lists is returned in <code>ret_int</code> .
EX_INQ_EB_PROP	The number of integer properties stored for each element block is returned in <code>ret_int</code> ; this number includes the property named “ID”.
EX_INQ_NS_PROP	The number of integer properties stored for each node set is returned in <code>ret_int</code> ; this number includes the property named “ID”.

---

<sup>3</sup>The API and DB version numbers are synchronized and will always match. Initially, it was thought that maintaining the two versions separately would be a benefit, but that was more confusing than helpful, so the numbers were made the same awhile ago

EX_INQ_SS_PROP	The number of integer properties stored for each side set is returned in <code>ret_int</code> ; this number includes the property named “ID”.
EX_INQ_QA	The number of QA records is returned in <code>ret_int</code> .
EX_INQ_INFO	The number of information records is returned in <code>ret_int</code> .
EX_INQ_TIME	The number of time steps stored in the database is returned in <code>ret_int</code> .
EX_INQ_EDGE_BLK	The number of edge blocks is returned in <code>ret_int</code> .
EX_INQ_EDGE_MAP	The number of edge maps is returned in <code>ret_int</code> .
EX_INQ_EDGE_PROP	The number of properties stored per edge block is returned in <code>ret_int</code> .
EX_INQ_EDGE_SETS	The number of edge sets is returned in <code>ret_int</code> .
EX_INQ_EDGE	The number of edges is returned in <code>ret_int</code> .
EX_INQ_FACE	The number of faces is returned in <code>ret_int</code> .
EX_INQ_EB_PROP	The number of element block properties is returned in <code>ret_int</code> .
EX_INQ_ELEM_MAP	The number of element maps is returned in <code>ret_int</code> .
EX_INQ_ELEM_SETS	The number of element sets is returned in <code>ret_int</code> .
EX_INQ_ELS_DF_LEN	The length of the concatenated element set distribution factor list is returned in <code>ret_int</code> .
EX_INQ_ELS_LEN	The length of the concatenated element set element list is returned in <code>ret_int</code> .
EX_INQ_ELS_PROP	The number of properties stored per elem set is returned in <code>ret_int</code> .
EX_INQ_EM_PROP	The number of element map properties is returned in <code>ret_int</code> .
EX_INQ_ES_DF_LEN	The length of the concatenated edge set distribution factor list is returned in <code>ret_int</code> .
EX_INQ_ES_LEN	The length of the concatenated edge set edge list is returned in <code>ret_int</code> .
EX_INQ_ES_PROP	The number of properties stored per edge set is returned in <code>ret_int</code> .
EX_INQ_FACE_BLK	The number of face blocks is returned in <code>ret_int</code> .
EX_INQ_FACE_MAP	The number of face maps is returned in <code>ret_int</code> .
EX_INQ_FACE_PROP	The number of properties stored per face block is returned in <code>ret_int</code> .
EX_INQ_FACE_SETS	The number of face sets is returned in <code>ret_int</code> .
EX_INQ_FS_DF_LEN	The length of the concatenated face set distribution factor list is returned in <code>ret_int</code> .
EX_INQ_FS_LEN	The length of the concatenated face set face list is returned in <code>ret_int</code> .
EX_INQ_FS_PROP	The number of properties stored per face set is returned in <code>ret_int</code> .
EX_INQ_NM_PROP	The number of node map properties is returned in <code>ret_int</code> .
EX_INQ_NODE_MAP	The number of node maps is returned in <code>ret_int</code> .

`int* ret_int [out]`

Returned integer, if an integer value is requested according to `req_info`); otherwise, supply a dummy argument.

`float* ret_float [out]`

Returned float, if a float value is requested (according to `req_info`); otherwise, supply a dummy argument<sup>4</sup>.

`char* ret_char [out]`

Returned character string, if a character value is requested according to `req_info`); otherwise, supply a dummy argument.

As an example, the following will return the number of element block properties stored in the EXODUS file:

---

<sup>4</sup>NOTE: This argument is always a float even if the database IO and/or CPU word size is a double.



```

1 #include "exodusII.h"
2 int error, exoid, num_props;
3 float fdum;
4 char *cdum;
5
6 /* determine the number of element block properties */
7 error = ex_inquire (exoid, EX_INQ_EB_PROP, &num_props,
8                    &fdum, cdum);

```

### 5.1.11 Inquire EXODUS Integer Parameters

The function `ex_inquire_int` is used to query or inquire values of certain integer data entities in an EXODUS file. It is a short-cut to the `ex_inquire` function defined in the previous section. If there is no error, the queried value will be returned as a positive number. In case of an error, `ex_inquire` returns a negative number.

- data file not properly opened with call to `ex_create` or `ex_open`.
- requested information not stored in the file.
- invalid request flag.

```
int ex_inquire_int (int exoid,
                  ex_inquiry req_info)
```

`int exoid` [in]

EXODUS file ID returned from a previous call to `ex_create` or `ex_open`.

`ex_inquiry req_info` [in]

A flag which designates what information is requested. It must be one of the following constants (predefined in the file *exodusII.h*):

EX_INQ_API_VERS	The “undotted” EXODUS API version number is returned. The API version number reflects the release of the function library (i.e., function names, argument list, etc.). The current “undotted” API version is 472.
EX_INQ_LIB_VERS	The “undotted” EXODUS API library version number is returned. The API library version number reflects the format of the data as it is stored in the NetCDF database. The current API version is 472
EX_INQ_DB_VERS	The “undotted” EXODUS database version number is returned. The database version number reflects the version of the library that was used to write the file pointed to by <code>exoid</code> . The current database version is 472.
EX_INQ_DIM	The dimensionality, or number of coordinates per node (1, 2 or 3), of the database is returned.
EX_INQ_NODES	The number of nodes is returned.
EX_INQ_ELEM	The number of elements is returned.
EX_INQ_ELEM_BLK	The number of element blocks is returned.
EX_INQ_NODE_SETS	The number of node sets is returned.
EX_INQ_NS_NODE_LEN	The length of the concatenated node sets node list is returned.
EX_INQ_NS_DF_LEN	The length of the concatenated node sets distribution list is returned.
EX_INQ_SIDE_SETS	The number of side sets is returned.

EX_INQ_SS_ELEM_LEN	The length of the concatenated side sets element list is returned.
EX_INQ_SS_DF_LEN	The length of the concatenated side sets distribution factor list is returned.
EX_INQ_SS_NODE_LEN	The aggregate length of all of the side sets node lists is returned.
EX_INQ_EB_PROP	The number of integer properties stored for each element block is returned; this number includes the property named "ID".
EX_INQ_NS_PROP	The number of integer properties stored for each node set is returned; this number includes the property named "ID".
EX_INQ_SS_PROP	The number of integer properties stored for each side set is returned; this number includes the property named "ID".
EX_INQ_QA	The number of QA records is returned.
EX_INQ_INFO	The number of information records is returned.
EX_INQ_TIME	The number of time steps stored in the database is returned.
EX_INQ_EDGE_BLK	The number of edge blocks is returned.
EX_INQ_EDGE_MAP	The number of edge maps is returned.
EX_INQ_EDGE_PROP	The number of properties stored per edge block is returned.
EX_INQ_EDGE_SETS	The number of edge sets is returned.
EX_INQ_EDGE	The number of edges is returned.
EX_INQ_FACE	The number of faces is returned.
EX_INQ_EB_PROP	The number of element block properties is returned.
EX_INQ_ELEM_MAP	The number of element maps is returned.
EX_INQ_ELEM_SETS	The number of element sets is returned.
EX_INQ_ELS_DF_LEN	The length of the concatenated element set distribution factor list is returned.
EX_INQ_ELS_LEN	The length of the concatenated element set element list is returned.
EX_INQ_ELS_PROP	The number of properties stored per elem set is returned.
EX_INQ_EM_PROP	The number of element map properties is returned.
EX_INQ_ES_DF_LEN	The length of the concatenated edge set distribution factor list is returned.
EX_INQ_ES_LEN	The length of the concatenated edge set edge list is returned.
EX_INQ_ES_PROP	The number of properties stored per edge set is returned.
EX_INQ_FACE_BLK	The number of face blocks is returned.
EX_INQ_FACE_MAP	The number of face maps is returned.
EX_INQ_FACE_PROP	The number of properties stored per face block is returned.
EX_INQ_FACE_SETS	The number of face sets is returned.
EX_INQ_FS_DF_LEN	The length of the concatenated face set distribution factor list is returned.
EX_INQ_FS_LEN	The length of the concatenated face set face list is returned.
EX_INQ_FS_PROP	The number of properties stored per face set is returned.
EX_INQ_NM_PROP	The number of node map properties is returned.
EX_INQ_NODE_MAP	The number of node maps is returned.

As an example, the following will return the number of nodes, elements, and element blocks stored in the EXODUS file:

```

1 #include "exodusII.h"
2 int exoid;
3 int num_nodes = ex_inquire_int(exoid, EX_INQ_NODES);
4 int num_elems = ex_inquire_int(exoid, EX_INQ_ELEM);
5 int num_block = ex_inquire_int(exoid, EX_INQ_ELEM_BLK);

```

### 5.1.12 Error Reporting

The function `ex_err` logs an error to `stderr`. It is intended to provide explanatory messages for error codes returned from other EXODUS routines. This function

The passed in error codes and corresponding messages are listed in Appendix C. The programmer may supplement the error message printed for standard errors by providing an error message. If the error code is provided with no error message, the predefined message will be used. The error code `EX_MSG` is available to log application specific messages.

```
void ex_err (char *module_name,
             char *message,
             int err_num)
```

`char* module_name [in]`

This is a string containing the name of the calling function.

`char* message [in]`

This is a string containing a message explaining the error or problem. If `EX_VERBOSE` (see `ex_opts`) is true, this message will be printed to `stderr`. Otherwise, nothing will be printed.

`int err_num [in]`

This is an integer code identifying the error. EXODUS C functions place an error code value in `exerrval`, an external int. Negative values are considered fatal errors while positive values are warnings. There is a set of predefined values defined in *exodusII.h*. The predefined constant `EX_PRTLASTMSG` will cause the last error message to be output, regardless of the setting of the error reporting level (see `ex_opts`).

The following is an example of the use of this function:

```
1 #include "exodusII.h"
2 int exoid, CPU_word_size, IO_word_size, errval;
3 float version;
4 char errmsg[MAX_ERR_LENGTH];
5
6 CPU_word_size = sizeof(float);
7 IO_word_size = 0;
8
9 /* open \exo{} file */
10 if (exoid = ex_open ("test.exo", EX_READ, &CPU_word_size,
11                     &IO_word_size, &version)) {
12     errval = 999;
13     sprintf(errmsg, "Error: cannot open file test.exo");
14     ex_err("prog_name", errmsg, errval);
15 }
```

### 5.1.13 Set Error Reporting Level

The function `ex_opts` is used to set message reporting options.

In case of an error, `ex_opts` returns a negative number; a warning will return a positive number.

```
int ex_opts (ex_options option_val)
```

```
int option_val [in]
```

Integer option value. Current options are:

**EX\_ABORT** Causes fatal errors to force program exit. (Default is false.)

**EX\_DEBUG** Causes certain messages to print for debug use. (Default is false.)

**EX\_VERBOSE** Causes all error messages to print when true, otherwise no error messages will print. (Default is false.)

NOTE: Values may be OR'ed together to provide any combination of these capabilities.

For example, the following will cause all messages to print and will cause the program to exit upon receipt of fatal error:

```
1 #include "exodusII.h"
2 ex_opts (EX_ABORT | EX_VERBOSE);
```

## 5.2 Model Description

The routines in this section read and write information which describe an EXODUS finite element model. This includes nodal coordinates, element order map, element connectivity arrays, element attributes, node sets, side sets, and object properties.

### 5.2.1 Write Nodal Coordinates

The function **ex\_put\_coord** writes the nodal coordinates of the nodes in the model. The function **ex\_put\_init** must be invoked before this call is made.

Because the coordinates are floating point values, the application code must declare the arrays passed to be the appropriate type ("float" or "double") to match the compute word size passed in **ex\_create** or **ex\_open**.

In case of an error, **ex\_put\_coord** returns a negative number; a warning will return a positive number. Possible causes of errors include:

- data file not properly opened with call to **ex\_create** or **ex\_open**
- data file opened for read only.
- data file not initialized properly with call to **ex\_put\_init**.

```
int ex_put_coord (int exoid,
                 void *x_coor,
                 void *y_coor,
                 void *z_coor)
```

```
int exoid [in]
```

EXODUS file ID returned from a previous call to **ex\_create** or **ex\_open**.

**void\* x\_coor [in]**

The X-coordinates of the nodes. If this is NULL, the X-coordinates will not be written.

**void\* y\_coor [in]**

The Y-coordinates of the nodes. These are stored only if `num_dim > 1`; otherwise, pass in dummy address. If this is NULL, the Y-coordinates will not be written.

**void\* z\_coor [in]**

The Z-coordinates of the nodes. These are stored only if `num_dim > 2`; otherwise, pass in dummy address. If this is NULL, the Z-coordinates will not be written.

The following will write the nodal coordinates to an open EXODUS file:

```

1  int error, exoid;
2
3  /* if file opened with compute word size of sizeof(float) */
4  float x[8], y[8], z[8];
5
6  /* write nodal coordinates values to database */
7  x[0] = 0.0; y[0] = 0.0; z[0] = 0.0;
8  x[1] = 0.0; y[1] = 0.0; z[1] = 1.0;
9  x[2] = 1.0; y[2] = 0.0; z[2] = 1.0;
10 x[3] = 1.0; y[3] = 0.0; z[3] = 0.0;
11 x[4] = 0.0; y[4] = 1.0; z[4] = 0.0;
12 x[5] = 0.0; y[5] = 1.0; z[5] = 1.0;
13 x[6] = 1.0; y[6] = 1.0; z[6] = 1.0;
14 x[7] = 1.0; y[7] = 1.0; z[7] = 0.0;
15
16 error = ex_put_coord(exoid, x, y, z);
17
18 /* Do the same as the previous call in three separate calls */
19 error = ex_put_coord(exoid, x, NULL, NULL);
20 error = ex_put_coord(exoid, NULL, y, NULL);
21 error = ex_put_coord(exoid, NULL, NULL, z);

```

## 5.2.2 Read Nodal Coordinates

The function `ex_get_coord` reads the nodal coordinates of the nodes. Memory must be allocated for the coordinate arrays (`x_coor`, `y_coor`, and `z_coor`) before this call is made. The length of each of these arrays is the number of nodes in the mesh.

Because the coordinates are floating point values, the application code must declare the arrays passed to be the appropriate type (“float” or “double”) to match the compute word size passed in `ex_create` or `ex_open`.

In case of an error, `ex_get_coord` returns a negative number; a warning will return a positive number. Possible causes of errors include:

- data file not properly opened with call to `ex_create` or `ex_open`
- a warning value is returned if nodal coordinates were not stored.

```
int ex_get_coord (int exoid,
                 void *x_coor,
                 void *y_coor,
                 void *z_coor)
```

```
int exoid [in]
```

EXODUS file ID returned from a previous call to `ex_create` or `ex_open`.

```
void* x_coor [out]
```

Returned X coordinates of the nodes. If this is `NULL`, the X-coordinates will not be read.

```
void* y_coor [out]
```

Returned Y coordinates of the nodes. These are returned only if `num_dim > 1`; otherwise, pass in a dummy address. If this is `NULL`, the Y-coordinates will not be read.

```
void* z_coor [out]
```

Returned Z coordinates of the nodes. These are returned only if `num_dim > 2`; otherwise, pass in a dummy address. If this is `NULL`, the Z-coordinates will not be read.

The following code segment will read the nodal coordinates from an open EXODUS file:

```
1 int error, exoid;
2
3 float *x, *y, *z;
4
5 /* read nodal coordinates values from database */
6 x = (float *)calloc(num_nodes, sizeof(float));
7 y = (float *)calloc(num_nodes, sizeof(float));
8 if (num_dim >= 3)
9     z = (float *)calloc(num_nodes, sizeof(float));
10 else
11     z = 0;
12
13 error = ex_get_coord(exoid, x, y, z);
14
15 /* Do the same as the previous call in three separate calls */
16 error = ex_get_coord(exoid, x, NULL, NULL);
17 error = ex_get_coord(exoid, NULL, y, NULL);
18 if (num_dim >= 3)
19     error = ex_get_coord(exoid, NULL, NULL, z);
```

### 5.2.3 Write Coordinate Names

- data file not properly opened with call to `ex_create` or `ex_open`
- data file opened for read only.
- data file not initialized properly with call to `ex_put_init`.

```
int ex_put_coord_names (int exoid,
                       char **coord_names)
```

**int exoid [in]**

EXODUS file ID returned from a previous call to `ex.create` or `ex.open`.

**char\*\* coord\_names [in]**

Array containing `num_dim` names of length `MAX_STR_LENGTH` of the nodal coordinate arrays.

The following coding will write the coordinate names to an open EXODUS file:

```

1 int error, exoid;
2
3 char *coord_names[3];
4 coord_names[0] = "xcoor";
5 coord_names[1] = "ycoor";
6 coord_names[2] = "zcoor";
7
8 error = ex_put_coord_names (exoid, coord_names);

```

### 5.2.4 Read Coordinate Names

The function `ex.get.coord_names` reads the names (`MAX_STR_LENGTH`-characters in length) of the coordinate arrays from the database. Memory must be allocated for the character strings before this function is invoked.

In case of an error, `ex.get.coord_names` returns a negative number; a warning will return a positive number. Possible causes of errors include:

- data file not properly opened with call to `ex.create` or `ex.open`
- a warning value is returned if coordinate names were not stored.

**int ex\_get\_coord\_names (int exoid,  
char \*\*coord\_names)**

**int exoid [in]**

EXODUS file ID returned from a previous call to `ex.create` or `ex.open`.

**char\*\* coord\_names [out]**

Returned pointer to a vector containing `num_dim` names of the nodal coordinate arrays.

The following code segment will read the coordinate names from an open EXODUS file:

```

1 int error, exoid;
2 char *coord_names[3];
3
4 for (i=0; i < num_dim; i++) {
5     coord_names[i] = (char *)calloc((MAX_STR_LENGTH+1), sizeof(char));
6 }
7
8 error = ex_get_coord_names (exoid, coord_names);

```

### 5.2.5 Write Node Number Map

The function `ex_put_node_num_map` writes out the optional node number map to the database. See Section 4.5 for a description of the node number map. The function `ex_put_init` must be invoked before this call is made.

In case of an error, `ex_put_node_num_map` returns a negative number; a warning will return a positive number. Possible causes of errors include:

- data file not properly opened with call to `ex_create` or `ex_open`
- data file opened for read only.
- data file not initialized properly with call to `ex_put_init`.
- a node number map already exists in the file.

```
int ex_put_node_num_map (int exoid,
                        int *node_map)
```

`int exoid [in]`  
EXODUS file ID returned from a previous call to `ex_create` or `ex_open`.

`int* node_map [in]`  
The node number map.

The following code generates a default node number map and outputs it to an open EXODUS file. This is a trivial case and included just for illustration. Since this map is optional, it should be written out only if it contains something other than the default map.

```
1 int error, exoid;
2 int *node_map = (int *)calloc(num_nodes, sizeof(int));
3
4 for (i=1; i <= num_nodes; i++)
5     node_map[i-1] = i;
6
7 error = ex_put_node_num_map(exoid, node_map);
```

### 5.2.6 Read Node Number Map

The function `ex_get_node_num_map` reads the optional node number map from the database. See Section 4.5 for a description of the node number map. If a node number map is not stored in the data file, a default array (1,2,3, .. `num_nodes`) is returned. Memory must be allocated for the node number map array (`num_nodes` in length) before this call is made.

In case of an error, `ex_get_node_num_map` returns a negative number; a warning will return a positive number. Possible causes of errors include:

- data file not properly opened with call to `ex_create` or `ex_open`
- if a node number map is not stored, a default map and a warning value are returned.



```
int ex_get_node_num_map (int exoid,
                        int *node_map)
```

```
int exoid [in]
```

EXODUS file ID returned from a previous call to `ex_create` or `ex_open`.

```
int* node_map [out]
```

Returned node number map.

The following code will read a node number map from an open EXODUS file:

```
1 int *node_map, error, exoid;
2
3 /* read node number map */
4 node_map = (int *)calloc(num_nodes, sizeof(int));
5 error = ex_get_node_num_map(exoid, node_map);
```

### 5.2.7 Write Element Number Map

The function `ex_put_elem_num_map` writes out the optional element number map to the database. See Section 4.6 for a description of the element number map. The function `ex_put_init` must be invoked before this call is made.

In case of an error, `ex_put_elem_num_map` returns a negative number; a warning will return a positive number. Possible causes of errors include:

- data file not properly opened with call to `ex_create` or `ex_open`
- data file opened for read only.
- data file not initialized properly with call to `ex_put_init`.
- an element number map already exists in the file.

```
int ex_put_elem_num_map (int exoid,
                        int *elem_map)
```

```
int exoid [in]
```

EXODUS file ID returned from a previous call to `ex_create` or `ex_open`.

```
int* elem_map [in]
```

The element number map.

The following code generates a default element number map and outputs it to an open EXODUS file. This is a trivial case and included just for illustration. Since this map is optional, it should be written out only if it contains something other than the default map.

```
1 int error, exoid;
2 int *elem_map = (int *)calloc(num_elem, sizeof(int));
3
```

```

4 for (i=1; i <= num_elem; i++)
5     elem_map[i-1] = i;
6
7 error = ex_put_elem_num_map(exoid, elem_map);

```

### 5.2.8 Read Element Number Map

The function `ex_get_elem_num_map` reads the optional element number map from the database. See Section 4.6 for a description of the element number map. If an element number map is not stored in the data file, a default array (1,2,3,... `num_elem`) is returned. Memory must be allocated for the element number map array (`num_elem` in length) before this call is made.

In case of an error, `ex_get_elem_num_map` returns a negative number; a warning will return a positive number. Possible causes of errors include:

- data file not properly opened with call to `ex_create` or `ex_open`
- if an element number map is not stored, a default map and a warning value are returned.

```

int ex_get_elem_num_map (int exoid,
                        int *elem_map)

```

`int exoid [in]`

EXODUS file ID returned from a previous call to `ex_create` or `ex_open`.

`int* elem_map [out]`

Returned element number map.

The following code will read an element number map from an open EXODUS file:

```

1 int *elem_map, error, exoid;
2
3 /* read element number map */
4 elem_map = (int *) calloc(num_elem, sizeof(int));
5 error = ex_get_elem_num_map (exoid, elem_map);

```

### 5.2.9 Write Element Order Map

The function `ex_put_map` writes out the optional element order map to the database. See Section 4.7 for a description of the element order map. The function `ex_put_init` must be invoked before this call is made.

In case of an error, `ex_put_map` returns a negative number; a warning will return a positive number. Possible causes of errors include:

- data file not properly opened with call to `ex_create` or `ex_open`
- data file opened for read only.
- data file not initialized properly with call to `ex_put_init`.

- an element map already exists in the file.

```
int ex_put_map (int exoid,
               int *elem_map)
```

int exoid [in]

EXODUS file ID returned from a previous call to `ex_create` or `ex_open`.

int\* elem\_map [in]

The element order map.

The following code generates a default element order map and outputs it to an open EXODUS file. This is a trivial case and included just for illustration. Since this map is optional, it should be written out only if it contains something other than the default map.

```
1 int error, exoid;
2 int *elem_map = (int *)calloc(num_elem, sizeof(int));
3 for (i=0; i < num_elem; i++) {
4     elem_map[i] = i+1;
5 }
6 error = ex_put_map(exoid, elem_map);
```

### 5.2.10 Read Element Order Map

The function `ex_get_map` reads the element order map from the database. See Section 4.7 for a description of the element order map. If an element order map is not stored in the data file, a default array (1,2,3,... `num_elem`) is returned. Memory must be allocated for the element map array (`num_elem` in length) before this call is made.

In case of an error, `ex_get_map` returns a negative number; a warning will return a positive number. Possible causes of errors include:

- data file not properly opened with call to `ex_create` or `ex_open`
- if an element order map is not stored, a default map and a warning value are returned.

```
int ex_get_map (int exoid,
               int *elem_map)
```

int exoid [in]

EXODUS file ID returned from a previous call to `ex_create` or `ex_open`.

int\* elem\_map [out]

Returned element order map.

The following code will read an element order map from an open EXODUS file:

```
1 int *elem_map, error, exoid;
2
```

```

3  /* read element order map */
4  elem_map = (int *)calloc(num_elem, sizeof(int));
5
6  error = ex_get_map(exoid, elem_map);

```

### 5.2.11 Write Element Block Parameters

The function `ex_put_elem_block` writes the parameters used to describe an element block.

In case of an error, `ex_put_elem_block` returns a negative number; a warning will return a positive number. Possible causes of errors include:

- data file not properly opened with call to `ex_create` or `ex_open`
- data file opened for read only.
- data file not initialized properly with call to `ex_put_init`.
- an element block with the same ID has already been specified.
- the number of element blocks specified in the call to `ex_put_init` has been exceeded.

```

int ex_put_elem_block (int exoid,
                      int elem_blk_id,
                      char *elem_type,
                      int num_elem_this_blk,
                      int num_nodes_per_elem,
                      int num_attr)

```

`int exoid [in]`

EXODUS file ID returned from a previous call to `ex_create` or `ex_open`.

`int elem_blk_id [in]`

The element block ID.

`char* elem_type [in]`

The type of elements in the element block. The maximum length of this string is `MAX_STR_LENGTH`.

`int num_elem_this_blk [in]`

The number of elements in the element block.

`int num_nodes_per_elem [in]`

The number of nodes per element in the element block.

`int num_attr [in]`

The number of attributes per element in the element block.

For example, the following code segment will initialize an element block with an ID of 10, write out the connectivity array, and write out the element attributes array:

```

1  int id, error, exoid, num_elem_in_blk, num_nodes_per_elem,
2      *connect, num_attr;
3
4  float *attrib;
5
6  /* write element block parameters */
7  id = 10;
8  num_elem_in_blk = 2;
9  num_nodes_per_elem = 4; /* elements are 4-node shells */
10 num_attr = 1;           /* one attribute per element */
11
12 error = ex_put_elem_block(exoid, id, "SHEL", num_elem_in_blk,
13                           num_nodes_per_elem, num_attr);
14
15 /* write element connectivity */
16 connect = (int *)calloc(num_elem_in_blk*num_nodes_per_elem, sizeof(int));
17
18 /* fill connect with node numbers; nodes for first element*/
19 connect[0] = 1; connect[1] = 2; connect[2] = 3; connect[3] = 4;
20
21 /* nodes for second element */
22 connect[4] = 5; connect[5] = 6; connect[6] = 7; connect[7] = 8;
23
24 error = ex_put_elem_conn (exoid, id, connect);
25
26 /* write element block attributes */
27 attrib = (float *) calloc (num_attr*num_elem_in_blk, sizeof(float));
28
29 for (i=0, cnt=0; i < num_elem_in_blk; i++) {
30     for (j=0; j < num_attr; j++, cnt++) {
31         attrib[cnt] = 1.0;
32     }
33 }
34
35 error = ex_put_elem_attr (exoid, id, attrib);

```

### 5.2.12 Read Element Block Parameters

The function `ex_get_elem_block` reads the parameters used to describe an element block. IDs of all element blocks stored can be determined by calling `ex_get_elem_blk_ids`.

In case of an error, `ex_get_elem_block` returns a negative number; a warning will return a positive number. Possible causes of errors include:

- data file not properly opened with call to `ex_create` or `ex_open`
- element block with specified ID is not stored in the data file.

```
int ex_get_elem_block (int exoid,
                      int elem_blk_id,
                      char *elem_type,
                      int *num_elem_this_blk,
                      int *num_nodes_per_elem,
                      int *num_attr)
```

**int exoid [in]**

EXODUS file ID returned from a previous call to `ex_create` or `ex_open`.

**int elem\_blk\_id [in]**

The element block ID.

**char\* elem\_type [out]**

Returned element type of elements in the element block. The maximum length of this string is `MAX_STR_LENGTH`.

**int\* num\_elem\_this\_blk [out]**

Returned number of elements in the element block.

**int\* num\_nodes\_per\_elem [out]**

Returned number of nodes per element in the element block.

**int\* num\_attr [out]**

Returned number of attributes per element in the element block.

As an example, the following code segment will read the parameters for the element block with an ID of 10 and read the connectivity and element attributes arrays from an open EXODUS file:

```
1 #include "exodusII.h"
2 int id, error, exoid, num_el_in_blk, num_nod_per_el, num_attr,
3     *connect;
4 float *attrib;
5 char elem_type[MAX_STR_LENGTH+1];
6
7 /* read element block parameters */
8 id = 10;
9
10 error = ex_get_elem_block(exoid, id, elem_type, &num_el_in_blk,
11                          &num_nod_per_el, &num_attr);
12
13 /* read element connectivity */
14 connect = (int *) calloc(num_nod_per_el*num_el_in_blk,
15                          sizeof(int));
16
17 error = ex_get_elem_conn(exoid, id, connect);
18
19 /* read element block attributes */
20 attrib = (float *) calloc (num_attr * num_el_in_blk, sizeof(float));
21 error = ex_get_elem_attr (exoid, id, attrib);
```

### 5.2.13 Read Element Blocks IDs

The function `ex_get_elem_blk_ids` reads the IDs of all of the element blocks. Memory must be allocated for the returned array of (num\_elem\_blk) IDs before this function is invoked. The required `size(num_elem_blk)` can be determined via a call to `ex_inquire` or `ex_inquire_int`.

In case of an error, `ex_get_elem_blk_ids` returns a negative number; a warning will return a positive number. Possible causes of errors include:

- data file not properly opened with call to `ex_create` or `ex_open`

```
int ex_get_elem_blk_ids (int exoid,
                        int *elem_blk_ids)
```

`int exoid [in]`

EXODUS file ID returned from a previous call to `ex_create` or `ex_open`.

`int* elem_blk_ids [out]`

Returned array of the element blocks IDs. The order of the IDs in this array reflects the sequence that the element blocks were introduced into the file.

The following code segment reads all the element block IDs:

```
1 int error, exoid, *idelbs, num_elem_blk;
2 idelbs = (int *) calloc(num_elem_blk, sizeof(int));
3
4 error = ex_get_elem_blk_ids (exoid, idelbs);
```

### 5.2.14 Write Element Block Connectivity

The function `ex_put_elem_conn` writes the connectivity array for an element block. The function `ex_put_elem_block` must be invoked before this call is made.

In case of an error, `ex_put_elem_conn` returns a negative number; a warning will return a positive number. Possible causes of errors include:

- data file opened for read only.
- data file not initialized properly with call to `ex_put_init`.
- `ex_put_elem_block` was not called previously.

```
int ex_put_elem_conn (int exoid,
                     int elem_blk_id,
                     int *connect)
```

`int exoid [in]`

EXODUS file ID returned from a previous call to `ex_create` or `ex_open`.

```
int elem_blk_id [in]
```

The element block ID.

```
int connect[num_elem_this_blk,num_nodes_per_elem] [in]
```

The connectivity array; a list of nodes (internal node IDs; See Section 4.5) that define each element in the element block. The node index cycles faster than the element index.

Refer to the code example in Section ?? for an example of writing the connectivity array for an element block.

### 5.2.15 Read Element Block Connectivity

The function `ex_get_elem_conn` reads the connectivity array for an element block. Memory must be allocated for the connectivity array( $\text{num\_elem\_this\_blk} \times \text{num\_nodes\_per\_elem}$  in length) before this routine is called.

In case of an error, `ex_get_elem_conn` returns a negative number; a warning will return a positive number. Possible causes of errors include:

- an element block with the specified ID is not stored in the file.

```
int ex_get_elem_conn (int exoid,
                     int elem_blk_id,
                     int *connect)
```

```
int exoid [in]
```

EXODUS file ID returned from a previous call to `ex_create` or `ex_open`.

```
int elem_blk_id [in]
```

The element block ID.

```
int connect[num_elem_this_blk,num_nodes_per_elem] [out]
```

Returned connectivity array; a list of nodes (internal node IDs; See Section 4.5) that define each element. The node index cycles faster than the element index.

Refer to the code example in Section 5.2.12 for an example of reading the connectivity for an element block.

### 5.2.16 Write Element Block Attributes

The function `ex_put_elem_attr` writes the attributes for an element block. Each element in the element block must have the same number of attributes, so there are( $\text{num\_attr} \times \text{num\_elem\_this\_blk}$ ) attributes for each element block. The function `ex_put_elem_block` must be invoked before this call is made.

Because the attributes are floating point values, the application code must declare the array passed to be the appropriate type (“float” or “double”) to match the compute word size passed in `ex_create` or `ex_open`.



In case of an error, `ex_put_elem_attr` returns a negative number; a warning will return a positive number. Possible causes of errors include:

- data file not properly opened with call to `ex_create` or `ex_open`
- data file opened for read only.
- data file not initialized properly with call to `ex_put_init`.
- `ex_put_elem_block` was not called previously for specified element block ID.
- `ex_put_elem_block` was called with 0 attributes specified.

```
int ex_put_elem_attr (int exoid,
                    int elem_blk_id,
                    void *attrib)
```

`int exoid [in]`

EXODUS file ID returned from a previous call to `ex_create` or `ex_open`.

`int elem_blk_id [in]`

The element block ID.

`void attrib[ num_elem_this_blk,num_attr] [in]`

The list of attributes for the element block. The `num_attr` index cycles faster.

Refer to the code example in Section 5.2.11 for an example of writing the attributes array for an element block.

### 5.2.17 Read Element Block Attributes

The function `ex_get_elem_attr` reads the attributes for an element block. Memory must be allocated for  $(\text{num\_attr} \times \text{num\_elem\_this\_blk})$  attributes before this routine is called.

Because the attributes are floating point values, the application code must declare the array passed to be the appropriate type (“float” or “double”) to match the compute word size passed in `ex_create` or `ex_open`.

In case of an error, `ex_get_elem_attr` returns a negative number; a warning will return a positive number. Possible causes of errors include:

- data file not properly opened with call to `ex_create` or `ex_open`
- invalid element block ID.
- a warning value is returned if no attributes are stored in the file.

```
int ex_get_elem_attr (int exoid,
                    int elem_blk_id,
                    void *attrib)
```

`int exoid [in]`

EXODUS file ID returned from a previous call to `ex.create` or `ex.open`.

`int elem_blk_id [in]`

The element block ID.

`void attrib[ num_elem_this_blk,num_attr] [out]`

Returned list of( $\text{num\_attr} \times \text{num\_elem\_this\_blk}$ ) attributes for the element block, with the `num_attr` index cycling faster.

Refer to the code example in Section 5.2.12 for an example of reading the element attributes for an element block.

### 5.2.18 Write Node Set Parameters

The function `ex.put_node_set_param` writes the node set ID, the number of nodes which describe a single node set, and the number of node set distribution factors for the node set.

In case of an error, `ex.put_node_set_param` returns a negative number; a warning will return a positive number. Possible causes of errors include:

- data file not properly opened with call to `ex.create` or `ex.open`
- data file opened for read only.
- data file not initialized properly with call to `ex.put_init`.
- the number of node sets specified in the call to `ex.put_init` was zero or has been exceeded.
- a node set with the same ID has already been stored.
- the specified number of distribution factors is not zero and is not equal to the number of nodes.

```
int ex_put_node_set_param (int exoid,
                          int node_set_id,
                          int num_nodes_in_set,
                          int num_dist_in_set)
```

`int exoid [in]`

EXODUS file ID returned from a previous call to `ex.create` or `ex.open`.

`int node_set_id [in]`

The node set ID.

`int num_nodes_in_set [in]`

The number of nodes in the node set.

`int num_dist_in_set [in]`

The number of distribution factors in the node set. This should be either 0 (zero) for no factors, or should equal `num_nodes_in_set`.

The following code segment will write out a node set to an open EXODUS file:

```

1  int id, num_nodes_in_set, num_dist_in_set, error, exoid,
2      *node_list;
3
4  float *dist_fact;
5
6  /* write node set parameters */
7  id = 20; num_nodes_in_set = 5; num_dist_in_set = 5;
8  error = ex_put_node_set_param(exoid, id, num_nodes_in_set,
9                                num_dist_in_set);
10
11 /* write node set node list */
12 node_list = (int *) calloc (num_nodes_in_set, sizeof(int));
13 node_list[0] = 100; node_list[1] = 101; node_list[2] = 102;
14 node_list[3] = 103; node_list[4] = 104;
15
16 error = ex_put_node_set(exoid, id, node_list);
17
18 /* write node set distribution factors */
19 dist_fact = (float *) calloc (num_dist_in_set, sizeof(float));
20 dist_fact[0] = 1.0; dist_fact[1] = 2.0; dist_fact[2] = 3.0;
21 dist_fact[3] = 4.0; dist_fact[4] = 5.0;
22
23 error = ex_put_node_set_dist_fact(exoid, id, dist_fact);

```

### 5.2.19 Read Node Set Parameters

The function `ex_get_node_set_param` reads the number of nodes which describe a single node set and the number of distribution factors for the node set.

In case of an error, `ex_get_node_set_param` returns a negative number; a warning will return a positive number. Possible causes of errors include:

- data file not properly opened with call to `ex_create` or `ex_open`
- a warning value is returned if no node sets are stored in the file.
- incorrect node set ID.

```

int ex_get_node_set_param (int exoid,
                           int node_set_id,
                           int *num_nodes_in_set,
                           int *num_dist_in_set)

```

`int exoid [in]`  
EXODUS file ID returned from a previous call to `ex_create` or `ex_open`.

`int node_set_id [in]`  
The node set ID.

`int* num_nodes_in_set [out]`  
Returned number of nodes in the node set.

**int\* num\_dist\_in\_set [out]**

Returned number of distribution factors in the node set.

The following code segment will read a node set from an open EXODUS file:

```

1  int error, exoid, id, num_nodes_in_set, num_df_in_set, *node_list;
2
3  float *dist_fact;
4
5  /* read node set parameters */
6  id = 100;
7
8  error = ex_get_node_set_param(exoid, id, &num_nodes_in_set,
9                                &num_df_in_set);
10
11 /* read node set node list */
12 node_list = (int *) calloc(num_nodes_in_set, sizeof(int));
13 error = ex_get_node_set(exoid, id, node_list);
14
15 /* read node set distribution factors */
16 if (num_df_in_set > 0) {
17     dist_fact = (float *) calloc(num_nodes_in_set, sizeof(float));
18     error = ex_get_node_set_dist_fact(exoid, id, dist_fact);
19 }

```

### 5.2.20 Write Node Set

The function `ex_put_node_set` writes the node list for a single node set. The function `ex_put_node_set_param` must be called before this routine is invoked.

In case of an error, `ex_put_node_set` returns a negative number; a warning will return a positive number. Possible causes of errors include:

- data file not properly opened with call to `ex_create` or `ex_open`
- data file opened for read only.
- data file not initialized properly with call to `ex_put_init`.
- `ex_put_node_set_param` not called previously.

**int ex\_put\_node\_set (int exoid,  
                      int node\_set\_id,  
                      int \*node\_set\_node\_list)**

**int exoid [in]**

EXODUS file ID returned from a previous call to `ex_create` or `ex_open`.

**int node\_set\_id [in]**

The node set ID.

```
int* node_set_node_list [in]
```

Array containing the node list for the node set. Internal node IDs are used in this list (See Section 4.5).

Refer to the description of `ex_put_node_set_param` for a sample code segment to write out a node set.

### 5.2.21 Write Node Set Distribution Factors

The function `ex_put_node_set_dist_fact` writes node set distribution factors for a single node set. The function `ex_put_node_set_param` must be called before this routine is invoked.

Because the distribution factors are floating point values, the application code must declare the array passed to be the appropriate type (“float” or “double”) to match the compute word size passed in `ex_create` or `ex_open`.

In case of an error, `ex_put_node_set_dist_fact` returns a negative number; a warning will return a positive number. Possible causes of errors include:

- data file not properly opened with call to `ex_create` or `ex_open`
- data file opened for read only.
- data file not initialized properly with call to `ex_put_init`.
- `ex_put_node_set_param` not called previously.
- a call to `ex_put_node_set_param` specified zero distribution factors.

```
int ex_put_node_set_dist_fact (int exoid,
                             int node_set_id,
                             void 8node_set_dist_fact)
```

```
int exoid [in]
```

EXODUS file ID returned from a previous call to `ex_create` or `ex_open`.

```
int node_set_id [in]
```

The node set ID.

```
void* node_set_dist_fact [in]
```

Array containing the distribution factors in the node set.

Refer to the description of `ex_put_node_set_param` for a sample code segment to write out the distribution factors for a node set.

### 5.2.22 Read Node Set Distribution Factors

The function `ex_get_node_set_dist_fact` returns the node set distribution factors for a single node set. Memory must be allocated for the list of distribution factors(`num_dist_in_set` in length) before this function is invoked.

Because the distribution factors are floating point values, the application code must declare the array passed to be the appropriate type (“float” or “double”) to match the compute word size passed in `ex.create` or `ex.open`.

In case of an error, `ex.get_node_set_dist_fact` returns a negative number; a warning will return a positive number. Possible causes of errors include:

- a warning value is returned if no distribution factors were stored.

```
int ex_get_node_set_dist_fact (int exoid,
                              int node_set_id,
                              void 8node_set_dist_fact)
```

`int exoid [in]`  
EXODUS file ID returned from a previous call to `ex.create` or `ex.open`.

`int node_set_id [in]`  
The node set ID.

`void* node_set_dist_fact [out]`  
Returned array containing the distribution factors in the node set.

Refer to the description of `ex.get_node_set_param` for a sample code segment to read a node set’s distribution factors.

### 5.2.23 Read Node Sets IDs

The function `ex.get_node_set_ids` reads the IDs of all of the node sets. Memory must be allocated for the returned array of (num`node`sets) IDs before this function is invoked.

In case of an error, `ex.get_node_set_ids` returns a negative number; a warning will return a positive number. Possible causes of errors include:

- data file not properly opened with call to `ex.create` or `ex.open`
- a warning value is returned if no node sets are stored in the file.

```
int ex_get_node_set_ids (int exoid,
                        int *node_set_ids)
```

`int exoid [in]`  
EXODUS file ID returned from a previous call to `ex.create` or `ex.open`.

`int* node_set_ids [out]`  
Returned array of the node sets IDs. The order of the IDs in this array reflects the sequence the node sets were introduced into the file.

As an example, the following code will read all of the node set IDs from an open data file:

```

1 int *ids, num_node_sets, error, exoid;
2
3 /* read node sets IDs */
4 ids = (int *) calloc(num_node_sets, sizeof(int));
5
6 error = ex_get_node_set_ids (exoid, ids);

```

### 5.2.24 Write Concatenated Node Sets

The function `ex_put_concat_node_sets` writes the node set ID's, node sets node count array, node sets distribution factor count array, node sets node list pointers array, node sets distribution factor pointer, node set node list, and node set distribution factors for all of the node sets. “Concatenated node sets” refers to the arrays required to define all of the node sets (ID array, counts arrays, pointers arrays, node list array, and distribution factors array) as described in Section 3.10 on page 11. Writing concatenated node sets is more efficient than writing individual node sets. See Appendix A for a discussion of efficiency issues.

Because the distribution factors are floating point values, the application code must declare the array passed to be the appropriate type (“float” or “double”) to match the compute word size passed in `ex_create` or `ex_open`.

In case of an error, `ex_put_concat_node_sets` returns a negative number; a warning will return a positive number. Possible causes of errors include:

- data file not properly opened with call to `ex_create` or `ex_open`
- data file opened for read only.
- data file not initialized properly with call to `ex_put_init`.
- the number of node sets specified in a call to `ex_put_init` was zero or has been exceeded.
- a node set with the same ID has already been stored.
- the number of distribution factors specified for one of the node sets is not zero and is not equal to the number of nodes in the same node set.

```

int ex_put_concat_node_sets (int exoid,
                             int *node_set_ids,
                             int *num_nodes_per_set,
                             int *num_dist_per_set,
                             int *node_sets_node_index,
                             int *node_sets_dist_index,
                             int *node_sets_node_list,
                             void *node_sets_dist_fact)

```

`int exoid [in]`  
 EXODUS file ID returned from a previous call to `ex_create` or `ex_open`.

`int* node_set_ids [in]`  
 Array containing the node set ID for each set.





### 5.2.25 Read Concatenated Node Sets

The function `ex_get_concat_node_sets` reads the node set ID's, node set node count array, node set distribution factors count array, node set node pointers array, node set distribution factors pointer array, node set node list, and node set distribution factors for all of the node sets. "Concatenated node sets" refers to the arrays required to define all of the node sets (ID array, counts arrays, pointers arrays, node list array, and distribution factors array) as described in Section 3.10 on page 11.

Because the distribution factors are floating point values, the application code must declare the array passed to be the appropriate type ("float" or "double") to match the compute word size passed in `ex_create` or `ex_open`.

The length of each of the returned arrays can be determined by invoking `ex_inquire` or `ex_inquire_int`.

In case of an error, `ex_get_concat_node_sets` returns a negative number; a warning will return a positive number. Possible causes of errors include:

- data file not properly opened with call to `ex_create` or `ex_open`
- a warning value is returned if no node sets are stored in the file.

```
int ex_get_concat_node_sets (int exoid,
                           int *node_set_ids,
                           int *num_nodes_per_set,
                           int *num_dist_per_set,
                           int *node_sets_node_index,
                           int *node_sets_dist_index,
                           int *node_sets_node_list,
                           void *node_sets_dist_fact)
```

`int exoid [in]`

EXODUS file ID returned from a previous call to `ex_create` or `ex_open`.

`int* node_set_ids [out]`

Returned array containing the node set ID for each set.

`int* num_nodes_per_set [out]`

Returned array containing the number of nodes for each set.

`int* num_dist_per_set [out]`

Returned array containing the number of distribution factors for each set.

`int* node_sets_node_index [out]`

Returned array containing the indices into the `node_set_node_list` which are the locations of the first node for each set. These indices are 0-based.

`int* node_sets_dist_index [out]`

Returned array containing the indices into the `node_set_dist_fact` which are the locations of the first distribution factor for each set. These indices are 0-based.

**int\* node\_sets\_node\_list [out]**

Returned array containing the nodes for all sets. Internal node IDs are used in this list (see Section 4.5).

**void\* node\_sets\_dist\_fact [out]**

Returned array containing the distribution factors for all sets.

As an example, the following code segment will read concatenated node sets:

```

1  #include "exodusII.h"
2
3  int error, exoid, num_node_sets, list_len, *ids,
4      *num_nodes_per_set, *num_df_per_set, *node_ind,
5      *df_ind, *node_list;
6
7  float *dist_fact
8
9  /* read concatenated node sets */
10 num_node_sets = ex_inquire_int(exoid, EX_INQ_NODE_SETS);
11
12 ids           = (int *) calloc(num_node_sets, sizeof(int));
13 num_nodes_per_set = (int *) calloc(num_node_sets, sizeof(int));
14 num_df_per_set  = (int *) calloc(num_node_sets, sizeof(int));
15 node_ind       = (int *) calloc(num_node_sets, sizeof(int));
16 df_ind         = (int *) calloc(num_node_sets, sizeof(int));
17
18 list_len = ex_inquire_int(exoid, EX_INQ_NS_NODE_LEN);
19 node_list = (int *) calloc(list_len, sizeof(int));
20
21 list_len = ex_inquire_int(exoid, EX_INQ_NS_DF_LEN);
22 dist_fact = (float *) calloc(list_len, sizeof(float));
23
24 error = ex_get_concat_node_sets (exoid, ids, num_nodes_per_set,
25                                 num_df_per_set, node_ind, df_ind,
26                                 node_list, dist_fact);

```

### 5.2.26 Write Side Set Parameters

The function `ex_put_side_set_param` writes the side set set ID and the number of sides (faces on 3D element types; edges on 2D element types) which describe a single side set, and the number of side set distribution factors on the side set. Because each side of a side set is completely defined by an element and a local side number, the number of sides is equal to the number of elements in a side set.

In case of an error, `ex_put_side_set_param` returns a negative number; a warning will return a positive number. Possible causes of errors include:

- data file not properly opened with call to `ex_create` or `ex_open`
- data file opened for read only.
- data file not initialized properly with call to `ex_put_init`.
- the number of side sets specified in the call to `ex_put_init` was zero or has been exceeded.

- a side set with the same ID has already been stored.

```
int ex_put_side_set_param (int exoid,
                          int side_set_id,
                          int num_side_in_set,
                          int num_dist_fact_in_set)
```

**int exoid [in]**  
EXODUS file ID returned from a previous call to `ex_create` or `ex_open`.

**int side\_set\_id [in]**  
The side set ID.

**int num\_side\_in\_set [in]**  
The number of sides (faces or edges) in the side set.

**int num\_dist\_fact\_in\_set [in]**  
The number of distribution factors on the side set.

The following code segment will write a side set to an open EXODUS file:

```
1 int error, exoid, id, num_sides, num_df,
2   elem_list[2], side_list[2];
3
4 float dist_fact[4];
5
6 /* write side set parameters */
7 id = 30;
8
9 num_sides = 2;
10 num_df    = 4;
11
12 error = ex_put_side_set_param (exoid, id, num_sides, num_df);
13
14 /* write side set element and side lists */
15 elem_list[0] = 1; elem_list[1] = 2;
16 side_list[0] = 1; side_list[1] = 1;
17
18 error = ex_put_side_set (exoid, id, elem_list, side_list);
19
20 /* write side set distribution factors */
21 dist_fact[0] = 30.0; dist_fact[1] = 30.1;
22 dist_fact[2] = 30.2; dist_fact[3] = 30.3;
23
24 error = ex_put_side_set_dist_fact (exoid, id, dist_fact);
```

### 5.2.27 Read Side Set Parameters

The function `ex_get_side_set_param` reads the number of sides (faces on 3D element types; edges on 2D element types) which describe a single side set, and the number of side set distribution factors on the side set.

In case of an error, `ex_get_side_set_param` returns a negative number; a warning will return a positive number. Possible causes of errors include:

- data file not properly opened with call to `ex_create` or `ex_open`
- a warning value is returned if no side sets are stored in the file.
- incorrect side set ID.

```
int ex_get_side_set_param (int exoid,
                          int side_set_id,
                          int *num_side_in_set,
                          int *num_dist_fact_in_set)
```

`int exoid [in]`  
EXODUS file ID returned from a previous call to `ex_create` or `ex_open`.

`int side_set_id [in]`  
The side set ID.

`int* num_side_in_set [out]`  
Returned number of sides (faces or edges) in the side set.

`int* num_dist_fact_in_set [out]`  
Returned number of distribution factors on the side set.

The following coding will read all of the side sets from an open EXODUS file:

```
1 int num_side_sets, error, exoid, num_sides_in_set, num_df_in_set,
2   num_elem_in_set, *ids, *elem_list, *side_list, *ctr_list,
3   *node_list;
4
5 float *dist_fact;
6
7 num_side_sets = ex_inquire_int(exoid, EX_INQ_SIDE_SETS);
8 ids = (int *) calloc(num_side_sets, sizeof(int));
9 error = ex_get_side_set_ids (exoid, ids);
10
11 for (i=0; i < num_side_sets; i++) {
12     error = ex_get_side_set_param (exoid, ids[i], tab &num_sides_in_set,
13                                   tab &num_df_in_set);
14
15     num_elem_in_set = num_sides_in_set;
16     elem_list = (int *) calloc(num_elem_in_set, sizeof(int));
17     side_list = (int *) calloc(num_sides_in_set, sizeof(int));
18     error = ex_get_side_set (exoid, ids[i], elem_list, side_list);
19
20     if (num_df_in_set > 0) {
21         /* get side set node list to correlate to dist factors */
22         ctr_list = (int *) calloc(num_elem_in_set, sizeof(int));
23         node_list = (int *) calloc(num_df_in_set, sizeof(int));
24         dist_fact = (float *) calloc(num_df_in_set, sizeof(float));
```

```

25     error = ex_get_side_set_node_list (exoid, ids[i], ctr_list,
26                                     node_list);
27
28     error = ex_get_side_set_dist_fact (exoid, ids[i], tab dist_fact);
29 }
30 }

```

### 5.2.28 Write Side Set

The function `ex_put_side_set` writes the side set element list and side set side (face on 3D element types; edge on 2D element types) list for a single side set. The routine `ex_put_side_set_param` must be called before this function is invoked.

In case of an error, `ex_put_side_set` returns a negative number; a warning will return a positive number. Possible causes of errors include:

- data file not properly opened with call to `ex_create` or `ex_open`
- data file opened for read only.
- data file not initialized properly with call to `ex_put_init`.
- `ex_put_side_set_param` not called previously.

```

int ex_put_side_set (int exoid,
                    int side_set_id,
                    int *side_set_elem_list,
                    int *side_set_side_list)

```

`int exoid [in]`

EXODUS file ID returned from a previous call to `ex_create` or `ex_open`.

`int side_set_id [in]`

The side set ID.

`int* side_set_elem_list [in]`

Array containing the elements in the side set. Internal element IDs are used in this list (see Section 4.5).

`int* side_set_side_list [in]`

Array containing the sides (faces or edges) in the side set.

For an example of a code segment to write a side set, refer to the description for `ex_put_side_set_param`.

### 5.2.29 Read Side Set

The function `ex_get_side_set` reads the side set element list and side set side (face for 3D element types; edge for 2D element types) list for a single side set. Memory must be allocated for the element list and side list (both are num'side'in'set in length) before this function is invoked.

In case of an error, `ex_get_side_set` returns a negative number; a warning will return a positive number. Possible causes of errors include:

- data file not properly opened with call to `ex_create` or `ex_open`
- a warning value is returned if no side sets are stored in the file.
- incorrect side set ID.

```
int ex_get_side_set (int exoid,
                    int side_set_id,
                    int *side_set_elem_list,
                    int *side_set_side_list)
```

`int exoid [in]`

EXODUS file ID returned from a previous call to `ex_create` or `ex_open`.

`int side_set_id [in]`

The side set ID.

`int* side_set_elem_list [out]`

Returned array containing the elements in the side set. Internal element IDs are used in this list (see Section 4.5).

`int* side_set_side_list [out]`

Returned array containing the sides (faces or edges) in the side set.

For an example of code to read a side set from an EXODUS II file, refer to the description for `ex_get_side_set_param`.

### 5.2.30 Write Side Set Distribution Factors

The function `ex_put_side_set_dist_fact` writes side set distribution factors for a single side set. The routine `ex_put_side_set_param` must be called before this function is invoked.

Because the distribution factors are floating point values, the application code must declare the array passed to be the appropriate type (“float” or “double”) to match the compute word size passed in `ex_create` or `ex_open`.

In case of an error, `ex_put_side_set_dist_fact` returns a negative number; a warning will return a positive number. Possible causes of errors include:

- data file not properly opened with call to `ex_create` or `ex_open`
- data file opened for read only.
- data file not initialized properly with call to `ex_put_init`.
- `ex_put_side_set_param` not called previously.
- a call to `ex_put_side_set_param` specified zero distribution factors.

```
int ex_put_side_set_dist_fact (int exoid,
                              int side_set_id,
                              void *side_set_dist_fact)
```

```
int exoid [in]
```

EXODUS file ID returned from a previous call to `ex_create` or `ex_open`.

```
int side_set_id [in]
```

The side set ID.

```
void* side_set_dist_fact [in]
```

Array containing the distribution factors in the side set.

For an example of a code segment to write side set distribution factors, refer to the description for `ex_put_side_set_param`.

### 5.2.31 Read Side Set Distribution Factors

The function `ex_get_side_set_dist_fact` returns the side set distribution factors for a single side set. Memory must be allocated for the list of distribution factors (`num_dist_fact_in_set` in length) before this function is invoked.

Because the distribution factors are floating point values, the application code must declare the array passed to be the appropriate type (“float” or “double”) to match the compute word size passed in `ex_create` or `ex_open`.

In case of an error, `ex_get_side_set_dist_fact` returns a negative number; a warning will return a positive number. Possible causes of errors include:

- a warning value is returned if no distribution factors were stored.

```
int ex_get_side_set_dist_fact (int exoid,
                              int side_set_id,
                              void *side_set_dist_fact)
```

```
int exoid [in]
```

EXODUS file ID returned from a previous call to `ex_create` or `ex_open`.

```
int side_set_id [in]
```

The side set ID.

```
void* side_set_dist_fact [out]
```

Returned array containing the distribution factors in the side set.

For an example of code to read side set distribution factors from an EXODUS file, refer to the description for `ex_get_side_set_param`.

### 5.2.32 Read Side Sets IDs

The function `ex_get_side_set_ids` reads the IDs of all of the side sets. Memory must be allocated for the returned array of (num'side'sets) IDs before this function is invoked.

In case of an error, `ex_get_side_set_ids` returns a negative number; a warning will return a positive number. Possible causes of errors include:

- data file not properly opened with call to `ex_create` or `ex_open`
- a warning value is returned if no side sets are stored in the file.

```
int ex_get_side_set_ids (int exoid,
                        int *side_set_ids)
```

`int exoid [in]`

EXODUS file ID returned from a previous call to `ex_create` or `ex_open`.

`int* side_set_ids [out]`

Returned array of the side set IDs. The order of the IDs in this array reflects the sequence the side sets were introduced into the file.

For an example of code to read side set IDs from an EXODUS II file, refer to the description for `ex_get_side_set_param`.

### 5.2.33 Read Side Set Node List

The function `ex_get_side_set_node_list` returns a node count array and a list of nodes on a single side set. With the 2.0 and later versions of the database, this node list isn't stored directly but can be derived from the element number in the side set element list, local side number in the side set side list, and the element connectivity array. The application program must allocate memory for the node count array and node list.

There is a one-to-one mapping (i.e., same order – as shown in Table 2, “Side Set Node Ordering,” on page 16 – and same number) between the nodes in the side set node list and the side set distribution factors. Thus, if distribution factors are stored for the side set of interest, the required size for the node list is the number of distribution factors returned by `ex_get_side_set_param`. If distribution factors are not stored for the side set, the application program must allocate a maximum size anticipated for the node list. This would be the product of the number of elements in the side set and the maximum number of nodes per side for all types of elements in the model, since side sets can span across different element types.

The length of the node count array is the length of the side set element list. For each entry in the side set element list, there is an entry in the side set side list, designating a local side number. The corresponding entry in the node count array is the number of nodes which define the particular side. In conjunction with the side set node list, this node count array gives an unambiguous nodal description of the side set.

In case of an error, `ex_get_side_set_node_list` returns a negative number; a warning will return a positive number. Possible causes of errors include:



- data file not properly opened with call to `ex_create` or `ex_open`
- a warning value is returned if no side sets are stored in the file.
- incorrect side set ID.

```
int ex_get_side_set_node_list (int exoid,
                             int side_set_id,
                             int *side_set_node_cnt_list,
                             int *side_set_node_list)
```

`int exoid [in]`

EXODUS file ID returned from a previous call to `ex_create` or `ex_open`.

`int side_set_id [in]`

The side set ID.

`int* side_set_node_cnt_list [out]`

Returned array containing the number of nodes for each side (face in 3D, edge in 2D) in the side set.

`int* side_set_node_list [out]`

Returned array containing a list of nodes on the side set. Internal node IDs are used in this list (see Section 3.5 4.5).

For an example of code to read a side set node list from an EXODUS file, refer to the description for `ex_get_side_set_param`.

### 5.2.34 Write Concatenated Side Sets

The function `ex_put_concat_side_sets` writes the side set IDs, side set element count array, side set distribution factor count array, side set element pointers array, side set distribution factors pointers array, side set element list, side set side list, and side set distribution factors. “Concatenated side sets” refers to the arrays needed to define all of the side sets (ID array, side counts array, node counts array, element pointer array, node pointer array, element list, node list, and distribution factors array) as described in Section 3.12 on page 15. Writing concatenated side sets is more efficient than writing individual side sets. See Appendix A for a discussion of efficiency issues.

Because the distribution factors are floating point values, the application code must declare the array passed to be the appropriate type (“float” or “double”) to match the compute word size passed in `ex_create` or `ex_open`.

In case of an error, `ex_put_concat_side_sets` returns a negative number; a warning will return a positive number. Possible causes of errors include:

- data file not properly opened with call to `ex_create` or `ex_open`
- data file opened for read only.
- data file not initialized properly with call to `ex_put_init`.

- the number of side sets specified in a call to `ex_put_init` was zero or has been exceeded.
- a side set with the same ID has already been stored.

```
int ex_put_concat_side_sets (int exoid,
                           int *side_sets_ids,
                           int *num_side_per_set,
                           int *num_dist_per_set,
                           int *side_sets_elem_index,
                           int *side_sets_dist_index,
                           int *side_sets_elem_list,
                           int *side_sets_side_list,
                           void *side_sets_dist_fact)
```

`int exoid [in]`  
EXODUS file ID returned from a previous call to `ex_create` or `ex_open`.

`int* side_sets_ids [in]`  
Array containing the side set ID for each set.

`int* num_side_per_set [in]`  
Array containing the number of sides for each set.

`int* num_dist_per_set [in]`  
Array containing the number of distribution factors for each set.

`int* side_sets_elem_index [in]`  
Array containing the indices into the `side_sets_elem_list` which are the locations of the first element for each set. These indices are 0-based.

`int* side_sets_dist_index [in]`  
Array containing the indices into the `side_sets_dist_fact` which are the locations of the first distribution factor for each set. These indices are 0-based.

`int* side_sets_elem_list [in]`  
Array containing the elements for all side sets. Internal element IDs are used in this list (see Section 4.6).

`int* side_sets_side_list [in]`  
Array containing the sides for all side sets.

`void* side_sets_dist_fact [in]`  
Array containing the distribution factors for all side sets.

The following coding will write out two side sets in a concatenated format:

```
1 int error, exoid, ids[2], num_side_per_set[2], elem_ind[2],
2   num_df_per_set[2], df_ind[2], elem_list[4], side_list[4];
```

```

3
4 float dist_fact[8];
5
6 /* write concatenated side sets */
7 ids[0] = 30;
8 ids[1] = 31;
9
10 num_side_per_set[0] = 2;
11 num_side_per_set[1] = 2;
12
13 elem_ind[0] = 0;
14 elem_ind[1] = 2;
15
16 num_df_per_set[0] = 4;
17 num_df_per_set[1] = 4;
18
19 df_ind[0] = 0;
20 df_ind[1] = 4;
21
22 /* side set #1 */
23 elem_list[0] = 2; elem_list[1] = 2;
24 side_list[0] = 2; side_list[1] = 1;
25
26 dist_fact[0] = 30.0; dist_fact[1] = 30.1;
27 dist_fact[2] = 30.2; dist_fact[3] = 30.3;
28
29 /* side set #2 */
30 elem_list[2] = 1; elem_list[3] = 2;
31 side_list[2] = 4; side_list[3] = 3;
32
33 dist_fact[4] = 31.0; dist_fact[5] = 31.1;
34 dist_fact[6] = 31.2; dist_fact[7] = 31.3;
35
36 error = ex_put_concat_side_sets (exoid, ids, num_side_per_set,
37                                 num_df_per_set, elem_ind, df_ind,
38                                 elem_list, side_list, dist_fact);

```

### 5.2.35 Read Concatenated Side Sets

The function `ex_get_concat_side_sets` reads the side set IDs, side set element count array, side set distribution factors count array, side set element pointers array, side set distribution factors pointers array, side set element list, side set side list, and side set distribution factors. “Concatenated side sets” refers to the arrays needed to define all of the side sets (ID array, side counts array, node counts array, element pointer array, node pointer array, element list, node list, and distribution factors array) as described in Section 3.12 on page 15.

Because the distribution factors are floating point values, the application code must declare the array passed to be the appropriate type (“float” or “double”) to match the compute word size passed in `ex_create` or `ex_open`.

The length of each of the returned arrays can be determined by invoking `ex_inquire` or `ex_inquire_int`.

In case of an error, `ex_get_concat_side_sets` returns a negative number; a warning will return a positive number. Possible causes of errors include:

- data file not properly opened with call to `ex_create` or `ex_open`
- a warning value is returned if no side sets are stored in the file.

```
int ex_get_concat_side_sets (int exoid,
                            int *side_set_ids,
                            int *num_side_per_set,
                            int *num_dist_per_set,
                            int *side_sets_elem_index,
                            int *side_sets_dist_index,
                            int *side_sets_elem_list,
                            int *side_sets_side_list,
                            void *side_sets_dist_fact)
```

`int exoid [in]`

EXODUS file ID returned from a previous call to `ex_create` or `ex_open`.

`int* side_set_ids [out]`

Returned array containing the side set ID for each set.

`int* num_side_per_set [out]`

Returned array containing the number of sides for each set.

`int* num_dist_per_set [out]`

Returned array containing the number of distribution factors for each set.

`int* side_sets_elem_index [out]`

Returned array containing the indices into the `side_sets_elem_list` which are the locations of the first element for each set. These indices are 0-based.

`int* side_sets_dist_index [out]`

Returned array containing the indices into the `side_sets_dist_fact` array which are the locations of the first distribution factor for each set. These indices are 0-based.

`int* side_sets_elem_list [out]`

Returned array containing the elements for all side sets. Internal element IDs are used in this list (see Section 4.6).

`int* side_sets_side_list [out]`

Returned array containing the sides for all side sets.

`void* side_sets_dist_fact [out]`

Returned array containing the distribution factors for all side sets.

The following code segment will return in concatenated format all the side sets stored in an EXODUS file:

```
1 #include "exodusII.h"
```

```

2
3 int error, exoid, num_ss, elem_list_len, df_list_len,
4     *ids, *side_list, *num_side_per_set, *num_df_per_set,
5     *elem_ind, *df_ind, *elem_list;
6
7 float *dist_fact;
8
9 num_ss = ex_inquire_int(exoid, EX_INQ_SIDE_SETS);
10
11 if (num_ss > 0) {
12     elem_list_len = ex_inquire_int(exoid, EX_INQ_SS_ELEM_LEN);
13     df_list_len   = ex_inquire_int(exoid, EX_INQ_SS_DF_LEN);
14
15     /* read concatenated side sets */
16     ids = (int *) calloc(num_ss, sizeof(int));
17     num_side_per_set = (int *) calloc(num_ss, sizeof(int));
18     num_df_per_set   = (int *) calloc(num_ss, sizeof(int));
19     elem_ind         = (int *) calloc(num_ss, sizeof(int));
20     df_ind           = (int *) calloc(num_ss, sizeof(int));
21     elem_list        = (int *) calloc(elem_list_len, sizeof(int));
22     side_list        = (int *) calloc(elem_list_len, sizeof(int));
23     dist_fact        = (float *) calloc(df_list_len, sizeof(float));
24
25     error = ex_get_concat_side_sets (exoid, ids, num_side_per_set,
26                                     num_df_per_set, elem_ind, df_ind,
27                                     elem_list, side_list, dist_fact);
28 }

```

### 5.2.36 Convert Side Set Nodes to Sides

The function `ex_cvt_nodes_to_sides` is used to convert a side set node list to a side set side list. This routine is provided for application programs that utilize side sets defined by nodes (as was done previous to release 2.0) rather than local faces or edges. The application program must allocate memory for the returned array of sides. The length of this array is the same as the length of the concatenated side sets element list, which can be determined with a call to `ex_inquire` or `ex_inquire_int`.

In case of an error, `ex_cvt_nodes_to_sides` returns a negative number; a warning will return a positive number. Possible causes of errors include:

- a warning value is returned if no side sets are stored in the file.
- because the faces of a wedge require a different number of nodes to describe them (quadrilateral vs. triangular faces), the function will abort with a fatal return code if a wedge is encountered in the side set element list.

```
int ex_cvt_nodes_to_sides (int exoid,
                          int *num_side_per_set,
                          int *num_nodes_per_set,
                          int *side_sets_elem_index,
                          int *side_sets_node_index,
                          int *side_sets_elem_list,
                          int *side_sets_node_list,
                          int *side_sets_side_list)
```

**int exoid [in]**

EXODUS file ID returned from a previous call to `ex_create` or `ex_open`.

**int\* num\_side\_per\_set [in]**

Array containing the number of sides for each set. The number of sides is equal to the number of elements for each set.

**int\* num\_nodes\_per\_set [in]**

Array containing the number of nodes for each set.

**int\* side\_sets\_elem\_index [in]**

Array containing indices into the `side_sets_elem_list` which are the locations of the first element for each set. These indices are 0-based.

**int\* side\_sets\_node\_index [in]**

Array containing indices into the `side_sets_node_list` which are the locations of the first node for each set. These indices are 0-based.

**int\* side\_sets\_elem\_list [in]**

Array containing the elements for all side sets. Internal element IDs are used in this list (see Section 4.6).

**int\* side\_sets\_node\_list [in]**

Array containing the nodes for all side sets. Internal node IDs are used in this list (see Section 4.5).

**int\* side\_sets\_side\_list [out]**

Returned array containing the sides for all side sets.

The following code segment will convert side sets described by nodes to side sets described by local side numbers:

```
1 int error, exoid, ids[2], num_side_per_set[2],
2   num_nodes_per_set[2], elem_ind[2], node_ind[2],
3   elem_list[4], node_list[8], el_lst_len, *side_list;
4
5 ids[0] = 30          ; ids[1] = 31;
6 num_side_per_set[0] = 2; num_side_per_set[1] = 2;
7 num_nodes_per_set[0] = 4; num_nodes_per_set[1] = 4;
8
```

```

9  elem_ind[0] = 0; elem_ind[1] = 2;
10 node_ind[0] = 0; node_ind[1] = 4;
11
12 /* side set #1 */
13 elem_list[0] = 2; elem_list[1] = 2;
14 node_list[0] = 8; node_list[1] = 5;
15 node_list[2] = 6; node_list[3] = 7;
16
17 /* side set #2 */
18 elem_list[2] = 1; elem_list[3] = 2;
19 node_list[4] = 2; node_list[5] = 3;
20 node_list[6] = 7; node_list[7] = 8;
21
22 el_lst_len = ex_inquire_int(exoid, EX_INQ_SS_ELEM_LEN);
23
24 /* side set element list is same length as side list */
25 side_list = (int *) calloc (el_lst_len, sizeof(int));
26
27 ex_cvt_nodes_to_sides(exoid, num_side_per_set, num_nodes_per_set,
28                       elem_ind, node_ind, elem_list,
29                       node_list, side_list);

```

### 5.2.37 Write Property Arrays Names

The function `ex_put_prop_names` writes object property names and allocates space for object property arrays used to assign integer properties to element blocks, node sets, or side sets. The property arrays are initialized to zero (0). Although this function is optional, since `ex_put_prop` will allocate space within the data file if it hasn't been previously allocated, it is more efficient to use `ex_put_prop_names` if there is more than one property to store. See Appendix A for a discussion of efficiency issues.

In case of an error, `ex_put_prop_names` returns a negative number; a warning will return a positive number. Possible causes of errors include:

- data file not properly opened with call to `ex_create` or `ex_open`
- data file opened for read only.
- data file not initialized properly with call to `ex_put_init`.
- invalid object type specified.
- no object of the specified type is stored in the file.

```

int ex_put_prop_names (int exoid,
                      ex_entity_type obj_type,
                      int num_props,
                      char **prop_names)

```

`int exoid [in]`

EXODUS file ID returned from a previous call to `ex_create` or `ex_open`.

**ex\_entity\_type obj\_typ [in]**

Type of object; use one of the following options:

EX_NODE_SET	Node Set entity type
EX_EDGE_BLOCK	Edge Block entity type
EX_EDGE_SET	Edge Set entity type
EX_FACE_BLOCK	Face Block entity type
EX_FACE_SET	Face Set entity type
EX_ELEM_BLOCK	Element Block entity type
EX_ELEM_SET	Element Set entity type
EX_SIDE_SET	Side Set entity type
EX_ELEM_MAP	Element Map entity type
EX_NODE_MAP	Node Map entity type
EX_EDGE_MAP	Edge Map entity type
EX_FACE_MAP	Face Map entity type

**int num\_props [in]**

The number of integer properties to be assigned to all of the objects of the type specified (element blocks, node sets, or side sets).

**char\*\* prop\_names [in]**

Array containing `num_props` names (of maximum length of `MAX_STR_LENGTH`) of properties to be stored.

For instance, suppose a user wanted to assign the 1st, 3rd, and 5th element blocks (those element blocks stored 1st, 3rd, and 5th, regardless of their ID) to a group (property) called “TOP”, and the 2nd, 3rd, and 4th element blocks to a group called “LSIDE”. This could be accomplished with the following code:

```

1  #include "exodusII.h";
2
3  char* prop_names[2];
4  int top_part[] = {1,0,1,0,1};
5  int lside_part[] = {0,1,1,1,0};
6
7  int id[] = {10, 20, 30, 40, 50};
8
9  prop_names[0] = "TOP";
10 prop_names[1] = "LSIDE";
11
12 /* This call to ex_put_prop_names is optional, but more efficient */
13 ex_put_prop_names (exoid, EX_ELEM_BLOCK, 2, prop_names);
14
15 /* The property values can be output individually thus */
16 for (i=0; i < 5; i++) {
17     ex_put_prop (exoid, EX_ELEM_BLOCK, id[i], prop_names[0],
18                 top_part[i]);
19
20     ex_put_prop (exoid, EX_ELEM_BLOCK, id[i], prop_names[1],
21                 lside_part[i]);
22 }
23
24 /* Alternatively, the values can be output as an array thus*/
25 ex_put_prop_array (exoid, EX_ELEM_BLOCK, prop_names[0],

```



```

26         top_part);
27
28 ex_put_prop_array (exoid, EX_ELEM_BLOCK, prop_names[1],
29                  lside_part);

```

### 5.2.38 Read Property Arrays Names

The function `ex_get_prop_names` returns names of integer properties stored for an element block, node set, or side set. The number of properties (needed to allocate space for the property names) can be obtained via a call to `ex_inquire` or `ex_inquire_int`.

In case of an error, `ex_get_prop_names` returns a negative number; a warning will return a positive number. Possible causes of errors include:

- data file not properly opened with call to `ex_create` or `ex_open`
- invalid object type specified.

```

int ex_get_prop_names (int exoid,
                      ex_entity_type obj_type,
                      char **prop_names)

```

`int exoid [in]`

EXODUS file ID returned from a previous call to `ex_create` or `ex_open`.

`ex_entity_type obj_type [in]`

Type of object; use one of the following options:

EX_NODE_SET	Node Set entity type
EX_EDGE_BLOCK	Edge Block entity type
EX_EDGE_SET	Edge Set entity type
EX_FACE_BLOCK	Face Block entity type
EX_FACE_SET	Face Set entity type
EX_ELEM_BLOCK	Element Block entity type
EX_ELEM_SET	Element Set entity type
EX_SIDE_SET	Side Set entity type
EX_ELEM_MAP	Element Map entity type
EX_NODE_MAP	Node Map entity type
EX_EDGE_MAP	Edge Map entity type
EX_FACE_MAP	Face Map entity type

`char** prop_names [out]`

eturned array containing `num_props` (obtained from call to `ex_inquire` or `ex_inquire_int`) names (of maximum length `MAX_STR_LENGTH`) of properties to be stored. "ID", a reserved property name, will be the first name in the array.

As an example, the following code segment reads in properties assigned to node sets:

```

1 #include "exodusII.h";
2 int error, exoid, num_props, *prop_values;

```

```

3 char *prop_names[MAX_PROPS];
4
5 /* read node set properties */
6 num_props = ex_inquire_int(exoid, EX_INQ_NS_PROP);
7
8 for (i=0; i < num_props; i++) {
9     prop_names[i] = (char *) malloc ((MAX_STR_LENGTH+1), sizeof(char));
10    prop_values = (int *) malloc (num_node_sets, sizeof(int));
11 }
12
13 error = ex_get_prop_names(exoid, EX_NODE_SET, prop_names);
14
15 for (i=0; i < num_props; i++) {
16     error = ex_get_prop_array(exoid, EX_NODE_SET, prop_names[i],
17                             prop_values);
18 }

```

### 5.2.39 Write Object Property

The function `ex.put_prop` stores an integer object property value to a single element block, node set, or side set. Although it is not necessary to invoke `ex.put_prop_names`, since `ex.put_prop` will allocate space within the data file if it hasn't been previously allocated, it is more efficient to use `ex.put_prop_names` if there is more than one property to store. See Appendix A for a discussion of efficiency issues.

It should be noted that the interpretation of the values of the integers stored as properties is left to the application code. In general, a zero (0) means the object does not have the specified property (or is not in the specified group); a nonzero value means the object does have the specified property. When space is allocated for the properties using `ex.put_prop_names` or `ex.put_prop`, the properties are initialized to zero (0).

Because the ID of an element block, node set, or side set is just another property (named "ID"), this routine can be used to change the value of an ID. This feature must be used with caution, though, because changing the ID of an object to the ID of another object of the same type (element block, node set, or side set) would cause two objects to have the same ID, and thus only the first would be accessible. Therefore, `ex.put_prop` issues a warning if a user attempts to give two objects the same ID.

In case of an error, `ex.put_prop` returns a negative number; a warning will return a positive number. Possible causes of errors include:

- data file not properly opened with call to `ex.create` or `ex.open`
- data file opened for read only.
- data file not initialized properly with call to `ex.put_init`.
- invalid object type specified.
- a warning is issued if a user attempts to change the ID of an object to the ID of an existing object of the same type.

```
int ex_put_prop (int exoid,
                 ex_entity_type obj_type,
                 int obj_id,
                 char *prop_name,
                 int value)
```

**int exoid [in]**

EXODUS file ID returned from a previous call to `ex_create` or `ex_open`.

**ex\_entity\_type obj\_type [in]**

Type of object; use one of the following options:

EX_NODE_SET	Node Set entity type
EX_EDGE_BLOCK	Edge Block entity type
EX_EDGE_SET	Edge Set entity type
EX_FACE_BLOCK	Face Block entity type
EX_FACE_SET	Face Set entity type
EX_ELEM_BLOCK	Element Block entity type
EX_ELEM_SET	Element Set entity type
EX_SIDE_SET	Side Set entity type
EX_ELEM_MAP	Element Map entity type
EX_NODE_MAP	Node Map entity type
EX_EDGE_MAP	Edge Map entity type
EX_FACE_MAP	Face Map entity type

**int obj\_id [in]**

The element block, node set, or side set ID.

**char\* prop\_name [in]**

The name of the property for which the value will be stored. Maximum length of this string is `MAX_STR_LENGTH`.

**int value [in]**

the value of the property.

For an example of code to write out an object property, refer to the description for `ex_put_prop_names`.

#### 5.2.40 Read Object Property

The function `ex_get_prop` reads an integer object property value stored for a single element block, node set, or side set.

In case of an error, `ex_get_prop` returns a negative number; a warning will return a positive number. Possible causes of errors include:

- data file not properly opened with call to `ex_create` or `ex_open`
- invalid object type specified.
- a warning value is returned if a property with the specified name is not found.

```
int ex_get_prop (int exoid,
                 ex_entity_type obj_type,
                 int obj_id,
                 char *prop_name,
                 int value)
```

**int exoid [in]**

EXODUS file ID returned from a previous call to `ex_create` or `ex_open`.

**ex\_entity\_type obj\_type [in]**

Type of object; use one of the following options:

EX_NODE_SET	Node Set entity type
EX_EDGE_BLOCK	Edge Block entity type
EX_EDGE_SET	Edge Set entity type
EX_FACE_BLOCK	Face Block entity type
EX_FACE_SET	Face Set entity type
EX_ELEM_BLOCK	Element Block entity type
EX_ELEM_SET	Element Set entity type
EX_SIDE_SET	Side Set entity type
EX_ELEM_MAP	Element Map entity type
EX_NODE_MAP	Node Map entity type
EX_EDGE_MAP	Edge Map entity type
EX_FACE_MAP	Face Map entity type

**int obj\_id [in]**

The element block, node set, or side set ID.

**char\* prop\_name [in]**

The name of the property (maximum length is `MAX_STR_LENGTH`) for which the value is desired.

**int\* value [out]**

Returned value of the property.

For an example of code to read an object property, refer to the description for `ex_get_prop_names`.

### 5.2.41 Write Object Property Array

The function `ex_put_prop_array` stores an array of (`num_elem_blk`, `num_node_sets`, or `num_side_sets`) integer property values for all element blocks, node sets, or side sets. The order of the values in the array must correspond to the order in which the element blocks, node sets, or side sets were introduced into the file. For instance, if the parameters for element block with ID 20 were written to a file (via `ex_put_elem_block`), and then parameters for element block with ID 10, followed by the parameters for element block with ID 30, the first, second, and third elements in the property array would correspond to element block 20, element block 10, and element block 30, respectively.

One should note that this same functionality (writing properties to multiple objects) can be accomplished with multiple calls to `ex_put_prop`.

Although it is not necessary to invoke `ex_put_prop_names`, since `ex_put_prop_array` will allocate space

within the data file if it hasn't been previously allocated, it is more efficient to use `ex_put_prop_names` if there is more than one property to store. See Appendix A for a discussion of efficiency issues.

In case of an error, `ex_put_prop_array` returns a negative number; a warning will return a positive number. Possible causes of errors include:

- data file not properly opened with call to `ex_create` or `ex_open`
- data file opened for read only.
- data file not initialized properly with call to `ex_put_init`.
- invalid object type specified.

```
int ex_put_prop_array (int exoid,
                      ex_entity_type obj_type,
                      char *prop_name,
                      int *values)
```

`int exoid [in]`

EXODUS file ID returned from a previous call to `ex_create` or `ex_open`.

`ex_entity_type obj_type [in]`

Type of object; use one of the following options:

EX_NODE_SET	Node Set entity type
EX_EDGE_BLOCK	Edge Block entity type
EX_EDGE_SET	Edge Set entity type
EX_FACE_BLOCK	Face Block entity type
EX_FACE_SET	Face Set entity type
EX_ELEM_BLOCK	Element Block entity type
EX_ELEM_SET	Element Set entity type
EX_SIDE_SET	Side Set entity type
EX_ELEM_MAP	Element Map entity type
EX_NODE_MAP	Node Map entity type
EX_EDGE_MAP	Edge Map entity type
EX_FACE_MAP	Face Map entity type

`char* prop_name [in]`

The name of the property for which the values will be stored. Maximum length of this string is `MAX_STR_LENGTH`.

`int* values [in]`

An array of property values.

For an example of code to write an array of object properties, refer to the description for `ex_put_prop_names`.

### 5.2.42 Read Object Property Array

The function `ex_get_prop_array` reads an array of integer property values for all element blocks, node sets, or side sets. The order of the values in the array correspond to the order in which the element

blocks, node sets, or side sets were introduced into the file. Before this function is invoked, memory must be allocated for the returned array of (`num_elem_blk`, `num_node_sets`, or `num_side_sets`) integer values.

This function can be used in place of `ex_get_elem_blk_ids`, `ex_get_node_set_ids`, and `ex_get_side_set_ids` to get element block, node set, and side set IDs, respectively, by requesting the property name “ID.” One should also note that this same function can be accomplished with multiple calls to `ex_get_prop`.

In case of an error, `ex_get_prop_array` returns a negative number; a warning will return a positive number. Possible causes of errors include:

- data file not properly opened with call to `ex_create` or `ex_open`
- invalid object type specified.
- a warning value is returned if a property with the specified name is not found.

```
int ex_get_prop_array (int exoid,
                      ex_entity_type obj_type,
                      char *prop_name,
                      int *values)
```

`int exoid [in]`

EXODUS file ID returned from a previous call to `ex_create` or `ex_open`.

`ex_entity_type obj_type [in]`

Type of object; use one of the following options:

<code>EX_NODE_SET</code>	Node Set entity type
<code>EX_EDGE_BLOCK</code>	Edge Block entity type
<code>EX_EDGE_SET</code>	Edge Set entity type
<code>EX_FACE_BLOCK</code>	Face Block entity type
<code>EX_FACE_SET</code>	Face Set entity type
<code>EX_ELEM_BLOCK</code>	Element Block entity type
<code>EX_ELEM_SET</code>	Element Set entity type
<code>EX_SIDE_SET</code>	Side Set entity type
<code>EX_ELEM_MAP</code>	Element Map entity type
<code>EX_NODE_MAP</code>	Node Map entity type
<code>EX_EDGE_MAP</code>	Edge Map entity type
<code>EX_FACE_MAP</code>	Face Map entity type

`char* prop_name [in]`

The name of the property (maximum length of `MAX_STR_LENGTH`) for which the values are desired.

`int* values [out]`

Returned array of property values.

For an example of code to read an array of object properties, refer to the description for `ex_get_prop_names`.

## 5.3 Results Data

This section describes functions which read and write analysis results data and related entities. These include results variables (global, elemental, and nodal), element variable truth table, and simulation times.

### 5.3.1 Write Results Variables Parameters

The function `ex_put_variable_param` writes the number of global, nodal, or element variables that will be written to the database.

In case of an error, `ex_put_variable_param` returns a negative number; a warning will return a positive number. Possible causes of errors include:

- data file not properly opened with call to `ex_create` or `ex_open`
- data file opened for read only.
- invalid variable type specified.
- data file not initialized properly with call to `ex_put_init`.
- this routine has already been called with the same variable type; redefining the number of variables is not allowed.
- a warning value is returned if the number of variables is specified as zero.

```
int ex_put_variable_param (int exoid,
                          ex_entity_type var_type,
                          int num_vars)
```

`int exoid [in]`

EXODUS file ID returned from a previous call to `ex_create` or `ex_open`.

`ex_entity_type var_type [in]`

Variable indicating the type of variable which is described. Use one of the following options:

EX_GLOBAL	Global entity type
EX_NODAL	Nodal entity type
EX_NODE_SET	Node Set entity type
EX_EDGE_BLOCK	Edge Block entity type
EX_EDGE_SET	Edge Set entity type
EX_FACE_BLOCK	Face Block entity type
EX_FACE_SET	Face Set entity type
EX_ELEM_BLOCK	Element Block entity type
EX_ELEM_SET	Element Set entity type
EX_SIDE_SET	Side Set entity type

`int num_vars [in]`

The number of `var_type` variables that will be written to the database.

For example, the following code segment initializes the data file to store global variables:

```

1 int num_glo_vars, error, exoid;
2
3 /* write results variables parameters */
4 num_glo_vars = 3;
5
6 error = ex_put_variable_param (exoid, EX_GLOBAL, num_glo_vars);

```

### 5.3.2 Read Results Variables Parameters

The function `ex_get_variable_param` reads the number of global, nodal, or element variables stored in the database.

In case of an error, `ex_get_variable_param` returns a negative number; a warning will return a positive number. Possible causes of errors include:

- data file not properly opened with call to `ex_create` or `ex_open`
- invalid variable type specified.

```

int ex_get_variable_param (int exoid,
                          ex_entity_type var_type,
                          int *num_vars)

```

`int exoid [in]`

EXODUS file ID returned from a previous call to `ex_create` or `ex_open`.

`ex_entity_type var_type [in]`

Variable indicating the type of variable which is described. Use one of the following options:

EX_GLOBAL	Global entity type
EX_NODAL	Nodal entity type
EX_NODE_SET	Node Set entity type
EX_EDGE_BLOCK	Edge Block entity type
EX_EDGE_SET	Edge Set entity type
EX_FACE_BLOCK	Face Block entity type
EX_FACE_SET	Face Set entity type
EX_ELEM_BLOCK	Element Block entity type
EX_ELEM_SET	Element Set entity type
EX_SIDE_SET	Side Set entity type

`int* num_vars [out]`

Returned number of `var_type` variables that are stored in the database.

As an example, the following coding will determine the number of global variables stored in the data file:

```

1 int num_glo_vars, error, exoid;
2
3 /* read global variables parameters */
4 error = ex_get_variable_param(exoid, EX_GLOBAL, &num_glo_vars);

```



### 5.3.3 Write Results Variables Names

The function `ex_put_variable_names` writes the names of the results variables to the database. The names are `MAX_STR_LENGTH`-characters in length. The function `ex_put_variable_param` must be called before this function is invoked.

In case of an error, `ex_put_variable_names` returns a negative number; a warning will return a positive number. Possible causes of errors include:

- data file not properly opened with call to `ex_create` or `ex_open`
- data file not initialized properly with call to `ex_put_init`.
- invalid variable type specified.
- `ex_put_variable_param` was not called previously or was called with zero variables of the specified type.
- `ex_put_variable_names` has been called previously for the specified variable type.

```
int ex_put_variable_names (int exoid,
                          ex_entity_type var_type,
                          int num_vars,
                          char **var_names[])
```

`int exoid [in]`

EXODUS file ID returned from a previous call to `ex_create` or `ex_open`.

`ex_entity_type var_type [in]`

Variable indicating the type of variable which is described. Use one of the following options:

<code>EX_GLOBAL</code>	Global entity type
<code>EX_NODAL</code>	Nodal entity type
<code>EX_NODE_SET</code>	Node Set entity type
<code>EX_EDGE_BLOCK</code>	Edge Block entity type
<code>EX_EDGE_SET</code>	Edge Set entity type
<code>EX_FACE_BLOCK</code>	Face Block entity type
<code>EX_FACE_SET</code>	Face Set entity type
<code>EX_ELEM_BLOCK</code>	Element Block entity type
<code>EX_ELEM_SET</code>	Element Set entity type
<code>EX_SIDE_SET</code>	Side Set entity type

`int num_vars [in]`

The number of `var_type` variables that will be written to the database.

`char** var_names [in]`

Array of pointers to `num_vars` variable names.

The following coding will write out the names associated with the nodal variables:

```
1 int num_nod_vars, error, exoid;
2 char *var_names[2];
```

```

3
4 /* write results variables parameters and names */
5 num_nod_vars = 2;
6
7 var_names[0] = "disx";
8 var_names[1] = "disy";
9
10 error = ex_put_variable_param (exoid, EX_NODAL, num_nod_vars);
11 error = ex_put_variable_names (exoid, EX_NODAL, num_nod_vars, var_names);

```

### 5.3.4 Read Results Variable Names

The function `ex_get_variable_names` reads the names of the results variables from the database. Memory must be allocated for the name array before this function is invoked. The names are `MAX_STR_LENGTH`-characters in length.

In case of an error, `ex_get_variable_names` returns a negative number; a warning will return a positive number. Possible causes of errors include:

- data file not properly opened with call to `ex_create` or `ex_open`
- invalid variable type specified.
- a warning value is returned if no variables of the specified type are stored in the file.

```

int ex_get_variable_names (int exoid,
                           ex_entity_type var_type,
                           int num_vars,
                           char *var_names[])

```

`int exoid [in]`

EXODUS file ID returned from a previous call to `ex_create` or `ex_open`.

`ex_entity_type var_type [in]`

Variable indicating the type of variable which is described. Use one of the following options:

<code>EX_GLOBAL</code>	Global entity type
<code>EX_NODAL</code>	Nodal entity type
<code>EX_NODE_SET</code>	Node Set entity type
<code>EX_EDGE_BLOCK</code>	Edge Block entity type
<code>EX_EDGE_SET</code>	Edge Set entity type
<code>EX_FACE_BLOCK</code>	Face Block entity type
<code>EX_FACE_SET</code>	Face Set entity type
<code>EX_ELEM_BLOCK</code>	Element Block entity type
<code>EX_ELEM_SET</code>	Element Set entity type
<code>EX_SIDE_SET</code>	Side Set entity type

`int num_vars [in]`

The number of `var_type` variables that will be read from the database.

`char** var_names [out]`

Returned array of pointers to `num_vars` variable names.

As an example, the following code segment will read the names of the nodal variables stored in the data file:

```

1 #include "exodusII.h"
2 int error, exoid, num_nod_vars;
3 char *var_names[10];
4
5 /* read nodal variables parameters and names */
6 error = ex_get_variable_param(exoid, EX_NODAL, &num_nod_vars);
7 for (i=0; i < num_nod_vars; i++) {
8     var_names[i] = (char *) calloc ((MAX_STR_LENGTH+1), sizeof(char));
9 }
10 error = ex_get_variable_names(exoid, EX_NODAL, num_nod_vars, var_names);

```

### 5.3.5 Write Time Value for a Time Step

The function `ex_put_time` writes the time value for a specified time step.

Because time values are floating point values, the application code must declare the array passed to be the appropriate type (“float” or “double”) to match the compute word size passed in `ex_create` or `ex_open`.

In case of an error, `ex_put_time` returns a negative number; a warning will return a positive number. Possible causes of errors include:

- data file not properly opened with call to `ex_create` or `ex_open`
- data file opened for read only.

```

int ex_put_time (int exoid,
                 int time_step,
                 void *time_value)

```

`int exoid [in]`

EXODUS file ID returned from a previous call to `ex_create` or `ex_open`.

`int time_step [in]`

The time step number. This is essentially a counter that is incremented only when results variables are output to the data file. The first time step is 1.

`void* time_value [in]`

The time at the specified time step.

The following code segment will write out the simulation time value at simulation time step `n`:

```

1 int error, exoid, n;
2 float time_value;
3
4 /* write time value */
5 error = ex_put_time (exoid, n, &time_value);

```

### 5.3.6 Read Time Value for a Time Step

The function `ex_get_time` reads the time value for a specified time step.

Because time values are floating point values, the application code must declare the array passed to be the appropriate type (“float” or “double”) to match the compute word size passed in `ex_create` or `ex_open`.

In case of an error, `ex_get_time` returns a negative number; a warning will return a positive number. Possible causes of errors include:

- data file not properly opened with call to `ex_create` or `ex_open`
- no time steps have been stored in the file.

```
int ex_get_time (int exoid,
                int time_step,
                void *time_value)
```

`int exoid [in]`

EXODUS file ID returned from a previous call to `ex_create` or `ex_open`.

`int time_step [in]`

The time step number. This is essentially an index (in the time dimension) into the global, nodal, and element variables arrays stored in the database. The first time step is 1.

`void* time_value [out]`

Returned time at the specified time step.

As an example, the following coding will read the time value stored in the data file for time step n:

```
1 int n, error, exoid;
2 float time_value;
3
4 /* read time value at time step 3 */
5 n = 3;
6 error = ex_get_time (exoid, n, &time_value);
```

### 5.3.7 Read All Time Values

The function `ex_get_all_times` reads the time values for all time steps. Memory must be allocated for the time values array before this function is invoked. The storage requirements (equal to the number of time steps) can be determined by using the `ex_inquire` or `ex_inquire_int` routines.

Because time values are floating point values, the application code must declare the array passed to be the appropriate type (“float” or “double”) to match the compute word size passed in `ex_create` or `ex_open`.

In case of an error, `ex_get_all_times` returns a negative number; a warning will return a positive number. Possible causes of errors include:

- data file not properly opened with call to `ex_create` or `ex_open`

- no time steps have been stored in the file.

```
int ex_get_all_times (int exoid,
                     void *time_values)
```

**int exoid [in]**

EXODUS file ID returned from a previous call to `ex_create` or `ex_open`.

**void\* time\_values [out]**

Returned array of times. These are the time values at all time steps.

The following code segment will read the time values for all time steps stored in the data file:

```
1 #include "exodusII.h"
2 int error, exoid, num_time_steps;
3 float *time_values;
4
5 /* determine how many time steps are stored */
6 num_time_steps = ex_inquire_int(exoid, EX_INQ_TIME);
7
8 /* read time values at all time steps */
9 time_values = (float *) calloc(num_time_steps, sizeof(float));
10
11 error = ex_get_all_times(exoid, time_values);
```

### 5.3.8 Write Element Variable Truth Table

The function `ex_put_elem_var_tab` writes the EXODUS element variable truth table to the database. The element variable truth table indicates whether a particular element result is written for the elements in a particular element block. A 0 (zero) entry indicates that no results will be output for that element variable for that element block. A non-zero entry indicates that the appropriate results will be output.

Although writing the element variable truth table is optional, it is encouraged because it creates at one time all the necessary NetCDF variables in which to hold the EXODUS element variable values. This results in significant time savings. See Appendix A for a discussion of efficiency issues.

The function `ex_put_variable_param` must be called before this routine in order to define the number of element variables.

In case of an error, `ex_put_elem_var_tab` returns a negative number; a warning will return a positive number. Possible causes of errors include:

- data file not properly opened with call to `ex_create` or `ex_open`
- data file opened for read only.
- data file not initialized properly with call to `ex_put_init`.
- the specified number of element blocks is different than the number specified in a call to `ex_put_init`.
- `ex_put_elem_block` not called previously to specify element block parameters.

- `ex_put_variable_param` not called previously to specify the number of element variables or was called but with a different number of element variables.
- `ex_put_elem_var` previously called.

```
int ex_put_elem_var_tab (int exoid,
                        int num_elem_blk,
                        int num_elem_var,
                        int **elem_var_tab)
```

```
int exoid [in]
    EXODUS file ID returned from a previous call to ex_create or ex_open.
```

```
int num_elem_blk [in]
    The number of element blocks.
```

```
int num_elem_var [in]
    The number of element variables.
```

```
int elem_var_tab[num_elem_blk,num_elem_var] [in]
    A 2-dimensional array (with the num_elem_var index cycling faster) containing the element variable truth table.
```

The following coding will create, populate, and write an element variable truth table to an opened EXODUS file (NOTE: all element variables are valid for all element blocks in this example.):

```
1 int *truth_tab, num_elem_blk, num_ele_vars, error, exoid;
2
3 /* write element variable truth table */
4 truth_tab = (int *)calloc((num_elem_blk*num_ele_vars), sizeof(int));
5
6 for (i=0, k=0; i < num_elem_blk; i++) {
7     for (j=0; j < num_ele_vars; j++) {
8         truth_tab[k++] = 1;
9     }
10 }
11 error = ex_put_elem_var_tab(exoid, num_elem_blk, num_ele_vars,
12                             truth_tab);
```

### 5.3.9 Read Element Variable Truth Table

The function `ex_get_elem_var_tab` reads the EXODUS element variable truth table from the database. For a description of the truth table, see the usage of the function `ex_put_elem_var_tab`. Memory must be allocated for the truth table(`num_elem_blk × num_elem_var` in length) before this function is invoked. If the truth table is not stored in the file, it will be created based on information in the file and then returned.

In case of an error, `ex_get_elem_var_tab` returns a negative number; a warning will return a positive number. Possible causes of errors include:

- data file not properly opened with call to `ex_create` or `ex_open`

- data file not initialized properly with call to `ex_put_init`.
- the specified number of element blocks is different than the number specified in a call to `ex_put_init`.
- there are no element variables stored in the file or the specified number of element variables doesn't match the number specified in a call to `ex_put_variable_param`.

```
int ex_get_elem_var_tab (int exoid,
                        int num_elem_blk,
                        int num_elem_var,
                        int *elem_var_tab)
```

`int exoid [in]`  
exo file ID returned from a previous call to `ex_create` or `ex_open`.

`int num_elem_blk [in]`  
The number of element blocks.

`int num_elem_var [in]`  
The number of element variables.

`int elem_var_tab[num_elem_blk,num_elem_var] [out]`  
Returned 2-dimensional array (with the `num_elem_var` index cycling faster) containing the element variable truth table.

As an example, the following coding will read the element variable truth table from an opened EXODUS file:

```
1 int *truth_tab, num_elem_blk, num_ele_vars, error, exoid;
2
3 truth_tab = (int *) calloc ((num_elem_blk*num_ele_vars),
4                             sizeof(int));
5
6 error = ex_get_elem_var_tab (exoid, num_elem_blk, num_ele_vars,
7                             truth_tab);
```

### 5.3.10 Write Element Variable Values at a Time Step

The function `ex_put_elem_var` writes the values of a single element variable for one element block at one time step. It is recommended, but not required, to write the element variable truth table (with `ex_put_elem_var_tab` before this function is invoked for better efficiency. See Appendix A for a discussion of efficiency issues.

Because element variables are floating point values, the application code must declare the array passed to be the appropriate type ("float" or "double") to match the compute word size passed in `ex_create` or `ex_open`.

In case of an error, `ex_put_elem_var` returns a negative number; a warning will return a positive number. Possible causes of errors include:

- data file not properly opened with call to `ex_create` or `ex_open`
- data file opened for read only.
- data file not initialized properly with call to `ex_put_init`.
- invalid element block ID.
- `ex_put_elem_block` not called previously to specify parameters for this element block.
- `ex_put_variable_param` not called previously specifying the number of element variables.
- an element variable truth table was stored in the file but contains a zero (indicating no valid element variable) for the specified element block and element variable.

```
int ex_put_elem_var (int exoid,
                    int time_step,
                    int elem_var_index,
                    int elem_blk_id,
                    int num_elem_this_blk,
                    void *elem_var_vals)
```

`int exoid [in]`

EXODUS file ID returned from a previous call to `ex_create` or `ex_open`.

`int time_step [in]`

The time step number, as described under `ex_put_time`. This is essentially a counter that is incremented only when results variables are output. The first time step is 1.

`int elem_var_index [in]`

The index of the element variable. The first variable has an index of 1.

`int elem_blk_id [in]`

The element block ID.

`int num_elem_this_blk [in]`

The number of elements in the given element block.

`void* elem_var_vals [in]`

Array of `num_elem_this_blk` values of the `elem_var_index`<sup>th</sup> element variable for the element block with ID of `elem_blk_id` at the `time_step`<sup>th</sup> time step.

The following coding will write out all of the element variables for a single time step `n` to an open EXODUS file:

```
1 int num_ele_vars, num_elem_blk, *num_elem_in_block, error,
2   exoid, n, *ebids;
3
4 /* write element variables */
5 for (k=1; k <= num_ele_vars; k++) {
6   for (j=0; j < num_elem_blk; j++) {
```



```

7     float *elem_var_vals = (float *)
8         calloc(num_elem_in_block[j], sizeof(float));
9
10    for (m=0; m < num_elem_in_block[j]; m++) {
11        /* simulation code fills this in */
12        elem_var_vals[m] = 10.0;
13    }
14
15    error = ex_put_elem_var (exoid, n, k, ebids[j],
16                            num_elem_in_block[j], elem_var_vals);
17    free (elem_var_vals);
18 }
19 }

```

### 5.3.11 Read Element Variable Values at a Time Step

The function `ex_get_elem_var` reads the values of a single element variable for one element block at one time step. Memory must be allocated for the element variable values array before this function is invoked.

Because element variables are floating point values, the application code must declare the array passed to be the appropriate type (“float” or “double”) to match the compute word size passed in `ex_create` or `ex_open`.

In case of an error, `ex_get_elem_var` returns a negative number; a warning will return a positive number. Possible causes of errors include:

- data file not properly opened with call to `ex_create` or `ex_open`
- variable does not exist for the desired element block.
- invalid element block.

```

int ex_get_elem_var (int exoid,
                    int time_step,
                    int elem_var_index,
                    int elem_blk_id,
                    int num_elem_this_blk,
                    void *elem_var_vals)

```

`int exoid [in]`

EXODUS file ID returned from a previous call to `ex_create` or `ex_open`.

`int time_step [in]`

The time step number, as described under `ex_put_time`, at which the element variable values are desired. This is essentially an index (in the time dimension) into the element variable values array stored in the database. The first time step is 1.

`int elem_var_index [in]`

The index of the desired element variable. The first variable has an index of 1.

```
int elem_blk_id [in]
```

The desired element block ID.

```
int num_elem_this_blk [in]
```

The number of elements in this element block.

```
void* elem_var_vals [out]
```

Returned array of `num_elem_this_blk` values of the `elem_var_index`<sup>th</sup> element variable for the element block with ID of `elem_blk_id` at the `time_step`<sup>th</sup> time step.

As an example, the following code segment will read the `var_index`<sup>th</sup> element variable at one time step stored in an EXODUS file:

```
1 int num_elem_blk, error, exoid, *num_elem_in_block, step, var_ind;
2
3 int *ids = (int *) calloc(num_elem_blk, sizeof(int));
4 error = ex_get_elem_blk_ids (exoid, ids);
5
6 step = 1; /* read at the first time step */
7 for (i=0; i < num_elem_blk; i++) {
8     float *var_vals = (float *) calloc (num_elem_in_block[i], sizeof(float));
9     error = ex_get_elem_var (exoid, step, var_ind, ids[i],
10                             num_elem_in_block[i], var_vals);
11     free (var_vals);
12 }
```

### 5.3.12 Read Element Variable Values through Time

The function `ex_get_elem_var_time` reads the values of an element variable for a single element through a specified number of time steps. Memory must be allocated for the element variable values array before this function is invoked.

Because element variables are floating point values, the application code must declare the array passed to be the appropriate type (“float” or “double”) to match the compute word size passed in `ex_create` or `ex_open`.

In case of an error, `ex_get_elem_var_time` returns a negative number; a warning will return a positive number. Possible causes of errors include:

- data file not properly opened with call to `ex_create` or `ex_open`
- data file not initialized properly with call to `ex_put_init`.
- `ex_put_elem_block` not called previously to specify parameters for all element blocks.
- variable does not exist for the desired element or results haven’t been written.

```
int ex_get_elem_var_time (int exoid,
                          int elem_var_index,
                          int elem_number,
                          int beg_time_step,
                          int end_time_step,
                          void *elem_var_vals)
```

**int exoid [in]**

EXODUS file ID returned from a previous call to **ex\_create** or **ex\_open**.

**int elem\_var\_index [in]**

The index of the desired element variable. The first variable has an index of 1.

**int elem\_number [in]**

The internal ID (see Section 4.6) of the desired element. The first element is 1.

**int beg\_time\_step [in]**

The beginning time step for which an element variable value is desired. This is not a time value but rather a time step number, as described under **ex\_put\_time**. The first time step is 1.

**int end\_time\_step [in]**

The last time step for which an element variable value is desired. If negative, the last time step in the database will be used. The first time step is 1.

**void\* elem\_var\_vals [out]**

Returned array of  $(\text{end\_time\_step} - \text{beg\_time\_step} + 1)$  values of the  $\text{elem\_number}^{\text{th}}$  element for the  $\text{elem\_var\_index}^{\text{th}}$  element variable.

For example, the following coding will read the values of the  $\text{var\_index}^{\text{th}}$  element variable for element number 2 from the first time step to the last time step:

```

1  /* determine how many time steps are stored */
2  int num_time_steps = ex_inquire_int(exoid, EX_INQ_TIME);
3
4  /* read an element variable through time */
5  float *var_values = (float *) calloc (num_time_steps, sizeof(float));
6  int var_index = 2;
7
8  int elem_num = 2;
9
10 int beg_time = 1;
11 int end_time = -1;
12
13 int error = ex_get_elem_var_time (exoid, var_index, elem_num,
14                                   beg_time, end_time, var_values);

```

### 5.3.13 Write Global Variables Values at a Time Step

The function **ex\_put\_glob\_vars** writes the values of all the global variables for a single time step. The function **ex\_put\_variable\_param** must be invoked before this call is made.

Because global variables are floating point values, the application code must declare the array passed to be the appropriate type (“float” or “double”) to match the compute word size passed in **ex\_create** or **ex\_open**.

In case of an error, **ex\_put\_glob\_vars** returns a negative number; a warning will return a positive number. Possible causes of errors include:

- data file not properly opened with call to **ex\_create** or **ex\_open**

- data file opened for read only.
- `ex_put_variable_param` not called previously specifying the number of global variables.

```
int ex_put_glob_vars (int exoid,
                    int time_step,
                    int num_glob_vars,
                    void *glob_var_vals)
```

`int exoid [in]`

EXODUS file ID returned from a previous call to `ex_create` or `ex_open`.

`int time_step [in]`

The time step number, as described under `ex_put_time`. This is essentially a counter that is incremented when results variables are output. The first time step is 1.

`int num_glob_vars [in]`

The number of global variables to be written to the database.

`void* glob_var_vals [in]`

Array of `num_glob_vars` global variable values for the `time_step`<sup>th</sup> time step.

As an example, the following coding will write the values of all the global variables at one time step to an open EXODUS II file:

```
1 int num_glo_vars, error, exoid, time_step;
2
3 float *glob_var_vals
4
5 /* write global variables */
6 for (j=0; j < num_glo_vars; j++) {
7     /* application code fills this array */
8     glob_var_vals[j] = 10.0;
9 }
10 error = ex_put_glob_vars (exoid, time_step, num_glo_vars,
11                          glob_var_vals);
```

### 5.3.14 Read Global Variables Values at a Time Step

The function `ex_get_glob_vars` reads the values of all the global variables for a single time step. Memory must be allocated for the global variables values array before this function is invoked.

Because global variables are floating point values, the application code must declare the array passed to be the appropriate type (“float” or “double”) to match the compute word size passed in `ex_create` or `ex_open`.

In case of an error, `ex_get_glob_vars` returns a negative number; a warning will return a positive number. Possible causes of errors include:

- data file not properly opened with call to `ex_create` or `ex_open`

- no global variables stored in the file.
- a warning value is returned if no global variables are stored in the file.

```
int ex_get_glob_vars (int exoid,
                     int time_step,
                     int num_glob_vars,
                     void *glob_var_vals)
```

**int exoid [in]**

EXODUS file ID returned from a previous call to `ex_create` or `ex_open`.

**int time\_step [in]**

The time step, as described under `ex_put_time`, at which the global variable values are desired. This is essentially an index (in the time dimension) into the global variable values array stored in the database. The first time step is 1.

**int num\_glob\_vars [in]**

The number of global variables stored in the database.

**void\* glob\_var\_vals [out]**

Returned array of `num_glob_vars` global variable values for the `time_step`<sup>th</sup> time step.

The following is an example code segment that reads all the global variables at one time step:

```
1 int num_glo_vars, time_step;
2
3 int error = ex_get_variable_param (idexo, EX_GLOBAL, &num_glo_vars);
4 float *var_values = (float *) calloc (num_glo_vars, sizeof(float));
5 error = ex_get_glob_vars (idexo, time_step, num_glo_vars,
6                           var_values);
```

### 5.3.15 Read Global Variable Values through Time

The function `ex_get_glob_var_time` reads the values of a single global variable through a specified number of time steps. Memory must be allocated for the global variable values array before this function is invoked.

Because global variables are floating point values, the application code must declare the array passed to be the appropriate type (“float” or “double”) to match the compute word size passed in `ex_create` or `ex_open`.

In case of an error, `ex_get_glob_var_time` returns a negative number; a warning will return a positive number. Possible causes of errors include:

- data file not properly opened with call to `ex_create` or `ex_open`
- specified global variable does not exist.
- a warning value is returned if no global variables are stored in the file.

```
int ex_get_glob_var_time (int exoid,
                          int glob_var_index,
                          int beg_time_step,
                          int end_time_step,
                          void *glob_var_vals)
```

**int exoid [in]**

EXODUS file ID returned from a previous call to `ex_create` or `ex_open`.

**int glob\_var\_index [in]**

The index of the desired global variable. The first variable has an index of 1.

**int beg\_time\_step [in]**

The beginning time step for which a global variable value is desired. This is not a time value but rather a time step number, as described under `ex_put_time`. The first time step is 1.

**int end\_time\_step [in]**

The last time step for which a global variable value is desired. If negative, the last time step in the database will be used. The first time step is 1.

**void\* glob\_var\_vals [out]**

Returned array of (end'time'step - beg'time'step + 1) values for the `glob_var_indexth` global variable.

The following is an example of using this function:

```
1 #include "exodusII.h"
2 int error, exoid, num_time_steps, var_index;
3 int beg_time, end_time;
4
5 float *var_values;
6
7 /* determine how many time steps are stored */
8 num_time_steps = ex_inquire_int(exoid, EX_INQ_TIME);
9
10 /* read the first global variable for all time steps */
11 var_index = 1;
12 beg_time = 1;
13 end_time = -1;
14
15 var_values = (float *) calloc (num_time_steps, sizeof(float));
16
17 error = ex_get_glob_var_time(exoid, var_index, beg_time,
18                             end_time, var_values);
```

### 5.3.16 Write Nodal Variable Values at a Time Step

The function `ex_put_nodal_var` writes the values of a single nodal variable for a single time step. The function `ex_put_variable_param` must be invoked before this call is made.

Because nodal variables are floating point values, the application code must declare the array passed to be the appropriate type (“float” or “double”) to match the compute word size passed in `ex_create` or `ex_open`.

In case of an error, `ex_put_nodal_var` returns a negative number; a warning will return a positive number. Possible causes of errors include:

- data file not properly opened with call to `ex_create` or `ex_open`
- data file opened for read only.
- data file not initialized properly with call to `ex_put_init`.
- `ex_put_variable_param` not called previously specifying the number of nodal variables.

```
int ex_put_nodal_var (int exoid,
                     int time_step,
                     int nodal_var_index,
                     int num_nodes,
                     void *nodal_var_vals)
```

`int exoid [in]`

EXODUS file ID returned from a previous call to `ex_create` or `ex_open`.

`int time_step [in]`

The time step number, as described under `ex_put_time`. This is essentially a counter that is incremented when results variables are output. The first time step is 1.

`int nodal_var_index [in]`

The index of the nodal variable. The first variable has an index of 1.

`int num_nodes [in]`

The number of nodal points.

`void* nodal_var_vals [in]`

Array of `num_nodes` values of the `nodal_var_index`<sup>th</sup> nodal variable for the `time_step`<sup>th</sup> time step.

As an example, the following code segment writes all the nodal variables for a single time step:

```
1 int num_nod_vars, num_nodes, error, exoid, time_step;
2
3 /* write nodal variables */
4 float *nodal_var_vals = (float *) calloc(num_nodes, sizeof(float));
5 for (k=1; k <= num_nod_vars; k++) {
6     for (j=0; j < num_nodes; j++) {
7         /* application code fills in this array */
8         nodal_var_vals[j] = 10.0;
9     }
10    error = ex_put_nodal_var(exoid, time_step, k, num_nodes,
11                             nodal_var_vals);
12 }
```

### 5.3.17 Read Nodal Variable Values at a Time Step

The function `ex_get_nodal_var` reads the values of a single nodal variable for a single time step. Memory must be allocated for the nodal variable values array before this function is invoked.

Because nodal variables are floating point values, the application code must declare the array passed to be the appropriate type (“float” or “double”) to match the compute word size passed in `ex_create` or `ex_open`.

In case of an error, `ex_get_nodal_var` returns a negative number; a warning will return a positive number. Possible causes of errors include:

- data file not properly opened with call to `ex_create` or `ex_open`
- specified nodal variable does not exist.
- a warning value is returned if no nodal variables are stored in the file.

```
int ex_get_nodal_var (int exoid,
                     int time_step,
                     int nodal_var_index,
                     int num_nodes,
                     void *nodal_var_vals)
```

`int exoid [in]`

EXODUS file ID returned from a previous call to `ex_create` or `ex_open`.

`int time_step [in]`

The time step, as described under `ex_put_time`, at which the nodal variable values are desired. This is essentially an index (in the time dimension) into the nodal variable values array stored in the database. The first time step is 1.

`int nodal_var_index [in]`

The index of the desired nodal variable. The first variable has an index of 1.

`int num_nodes [in]`

The number of nodal points.

`void* nodal_var_vals [out]`

Returned array of `num_nodes` values of the `nodal_var_index`<sup>th</sup> nodal variable for the `time_step`<sup>th</sup> time step.

For example, the following demonstrates how this function would be used:

```
1 /* read the second nodal variable at the first time step */
2 int time_step = 1;
3 int var_index = 2;
4
5 float *var_values = (float *) calloc (num_nodes, sizeof(float));
6 error = ex_get_nodal_var(exoid, time_step, var_index, num_nodes,
7                          var_values);
```



### 5.3.18 Read Nodal Variable Values through Time

The function `ex_get_nodal_var_time` reads the values of a nodal variable for a single node through a specified number of time steps. Memory must be allocated for the nodal variable values array before this function is invoked.

Because nodal variables are floating point values, the application code must declare the array passed to be the appropriate type (“float” or “double”) to match the compute word size passed in `ex_create` or `ex_open`.

In case of an error, `ex_get_nodal_var_time` returns a negative number; a warning will return a positive number. Possible causes of errors include:

- specified nodal variable does not exist.
- a warning value is returned if no nodal variables are stored in the file.

```
int ex_get_nodal_var_time (int exoid,
                          int nodal_var_index,
                          int node_number,
                          int beg_time_step,
                          int end_time_step,
                          void *nodal_var_vals)
```

`int exoid [in]`

EXODUS file ID returned from a previous call to `ex_create` or `ex_open`.

`int nodal_var_index [in]`

The index of the desired nodal variable. The first variable has an index of 1.

`int node_number [in]`

The internal ID (see Section 4.5) of the desired node. The first node is 1.

`int beg_time_step [in]`

The beginning time step for which a nodal variable value is desired. This is not a time value but rather a time step number, as described under `ex_put_time`. The first time step is 1.

`int end_time_step [in]`

The last time step for which a nodal variable value is desired. If negative, the last time step in the database will be used. The first time step is 1.

`void* nodal_var_vals [out]`

Returned array of  $(\text{end\_time\_step} - \text{beg\_time\_step} + 1)$  values of the `node_number`<sup>th</sup> node for the `nodal_var_index`<sup>th</sup> nodal variable.

For example, the following code segment will read the values of the first nodal variable for node number one for all time steps stored in the data file:

```
1 #include "exodusII.h"
2 int node_num, beg_time, end_time, error, exoid;
```

```
3
4  /* determine how many time steps are stored */
5  int num_time_steps = ex_inquire_int(exoid, EX_INQ_TIME);
6
7  /* read a nodal variable through time */
8  float *var_values = (float *) calloc (num_time_steps, sizeof(float));
9
10 int var_index = 1; node_num = 1; beg_time = 1; end_time = -1;
11 error = ex_get_nodal_var_time(exoid, var_index, node_num, beg_time,
12                               end_time, var_values);
```

## Chapter 6

# References

- [1] W. C. Mills-Curran, A. P. Gilkey, and D. P. Flanagan, “EXODUS: A Finite Element File Format for Pre- and Post-processing,” Technical Report SAND87-2977, Sandia National Laboratories, Albuquerque, New Mexico, September 1988.
- [2] G. D. Sjaardema, “Overview of the Sandia National Laboratories Engineering Analysis Code Access System,” Technical Report SAND92-2292, Sandia National Laboratories, Albuquerque, New Mexico, January 1993.
- [3] R. K. Rew, G. P. Davis, and S. Emmerson, “NetCDF User’s Guide: An Interface for Data Access,” Version 2.3, University Corporation for Atmospheric Research, Boulder, Colorado, April 1993.
- [4] Sun Microsystems, “External Data Representation Standard: Protocol Specification,” RFC 1014; Information Sciences Institute, May 1988.
- [5] PDA Engineering, “PATRAN Plus User Manual,” Publication No. 2191024, Costa Mesa, California, January 1990.

\*

## Appendix A

# Implementation of EXODUS with NetCDF

### A.1 Description

The NetCDF software is an I/O library, callable from C or Fortran, which stores and retrieves scientific data structures in self-describing, machine-independent files. *Self-describing means that a file includes information defining the data it contains.* Machine-independent means that a file is represented in a form that can be accessed by computers with different ways of storing integers, characters, and floating-point numbers. It is available via the web from <http://www.unidata.ucar.edu>. The current version is 3.6.2 although version 4.0 is expected to be released soon.

For the EXODUS implementation, the standard NetCDF distribution is used except that the following defined constants in the include file *netcdf.h* are modified to the values shown:

```
1 #define NC_MAX_DIMS 65536
2 #define NC_MAX_VARS 524288
3 #define NC_MAX_VAR_DIMS 8
```

### A.2 Efficiency Issues

There are some efficiency concerns with using NetCDF as the low level data handler. The main one is that whenever a new object is introduced, the file is put into *define mode*, *the new object is defined*, and then the file is taken out of *define mode*. A result of going in and out of *define mode* is that all of the data that was output previous to the introduction of the new object is copied to a new file. Obviously, this copying of data to a new file is very inefficient. We have attempted to minimize the number of times the data file is put into *define mode* by accumulating objects within a single EXODUS API function. Thus using optional features such as the element variable truth table, concatenated node and side sets, and writing all property array names with *ex.put\_prop\_names* will increase efficiency significantly.

## Appendix B

# Deprecated Functions

### `ex_get_concat_node_sets`

Use `ex_get_concat_sets(exoid, EX_NODE_SET, set_specs)` [See Section ??]

### `ex_get_concat_side_sets`

Use `ex_get_concat_sets(exoid, EX_SIDE_SET, set_specs)` [See Section ??]

### `ex_get_elem_attr`

Use `ex_get_attr(exoid, EX_ELEM_BLOCK, elem_blk_id, attrib)` [See Section ??]

### `ex_get_elem_attr_names`

Use `ex_get_attr_names(exoid, EX_ELEM_BLOCK, elem_blk_id, names)` [See Section ??]

### `ex_get_elem_blk_ids`

Use `ex_get_ids(exoid, EX_ELEM_BLOCK, ids)` [See Section ??]

### `ex_get_elem_block`

Use

`ex_get_block(exoid, EX_ELEM_BLOCK, elem_blk_id, elem_type, num_elem_this_blk, num_nodes_per_elem, num_attr)`  
[See Section ??]

### `ex_get_elem_conn`

Use `ex_get_conn(exoid, EX_ELEM_BLOCK, elem_blk_id, connect, 0, 0)` [See Section ??]

### `ex_get_elem_map`

Use `ex_get_num_map(exoid, EX_ELEM_MAP, map_id, elem_map)` [See Section ??]

### `ex_get_elem_var`

Use

`ex_get_var(exoid, time_step, EX_ELEM_BLOCK, elem_var_index, elem_blk_id, num_elem_this_blk, elem_var_vals)`  
[See Section ??]

### `ex_get_elem_var_tab`

Use

`ex_get_truth_table(exoid, EX_ELEM_BLOCK, num_elem_blk, num_elem_var, elem_var_tab)`

[See Section ??]

#### `ex_get_elem_var_time`

Use

`ex_get_var_time(exoid, EX_ELEM_BLOCK, elem_var_index, elem_number, beg_time_step, end_time_step, elem_var_`

[See Section ??]

#### `ex_get_elem_varid`

Use `ex_get_varid(exoid, EX_ELEM_BLOCK, varid)` [See Section ??]

#### `ex_get_map`

Use `ex_get_num_map` [See Section ??]

#### `ex_get_node_map`

Use `ex_get_num_map(exoid, EX_NODE_MAP, map_id, node_map)` [See Section ??]

#### `ex_get_node_set`

Use `ex_get_set(exoid, EX_NODE_SET, node_set_id, node_set_node_list, NULL)` [See Section ??]

#### `ex_get_node_set_dist_fact`

Use `ex_get_set_dist_fact(exoid, EX_NODE_SET, node_set_id, node_set_dist_fact)` [See Section ??]

#### `ex_get_node_set_ids`

Use `ex_get_ids(exoid, EX_NODE_SET, ids)` [See Section ??]

#### `ex_get_node_set_param`

Use `ex_get_set_param(exoid, EX_NODE_SET, node_set_id, num_nodes_in_set, num_df_in_set)` [See Section ??]

#### `ex_get_nset_var`

Use

`ex_get_var(exoid, time_step, EX_NODE_SET, nset_var_index, nset_id, num_node_this_nset, nset_var_vals)`

[See Section ??]

#### `ex_get_nset_var_tab`

Use `ex_get_truth_table(exoid, EX_NODE_SET, num_nodesets, num_nset_var, nset_var_tab)` [See Section ??]

#### `ex_get_nset_varid`

Use `ex_get_varid(exoid, EX_NODE_SET, varid)` [See Section ??]

#### `ex_get_one_elem_attr`

Use `ex_get_one_attr(exoid, EX_ELEM_BLOCK, elem_blk_id, attrib_index, attrib)` [See Section ??]

#### `ex_get_side_set`

Use `ex_get_set(exoid, EX_SIDE_SET, side_set_id, side_set_elem_list, side_set_side_list)` [See

Section ??]

#### `ex_get_side_set_dist_fact`

Use `ex_get_set_dist_fact` (`exoid`, `EX_SIDE_SET`, `side_set_id`, `side_set_dist_fact`) [See Section ??]

#### `ex_get_side_set_ids`

Use `ex_get_ids` (`exoid`, `EX_SIDE_SET`, `ids`) [See Section ??]

#### `ex_get_side_set_param`

Use  
`ex_get_set_param`(`exoid`, `EX_SIDE_SET`, `side_set_id`, `num_side_in_set`, `num_dist_fact_in_set`)  
[See Section ??]

#### `ex_get_sset_var`

Use  
`ex_get_var`(`exoid`, `time_step`, `EX_SIDE_SET`, `sset_var_index`, `sset_id`, `num_side_this_sset`, `sset_var_vals`)  
[See Section ??]

#### `ex_get_sset_var_tab`

Use `ex_get_truth_table` (`exoid`, `EX_SIDE_SET`, `num_sidesets`, `num_sset_var`, `sset_var_tab`) [See Section ??]

#### `ex_get_sset_varid`

Use `ex_get_varid` (`exoid`, `EX_SIDE_SET`, `varid`) [See Section ??]

#### `ex_get_var_name`

use `ex_get_variable_name` (`exoid`, `obj_type`, `var_num`, `*var_name`) [See Section ??]

#### `ex_get_var_names`

Use `ex_get_variable_names` (`exoid`, `obj_type`, `num_vars`, `var_names`) [See Section 5.3.4]

#### `ex_get_var_param`

Use `ex_get_variable_param` (`exoid`, `obj_type`, `*num_vars`) [See Section 5.3.2]

#### `ex_get_var_tab`

Use `ex_get_truth_table` (`exoid`, `obj_type`, `num_blk`, `num_var`, `var_tab`) [See Section ??]

#### `ex_put_concat_node_sets`

Use `ex_put_concat_sets` (`exoid`, `EX_NODE_SET`, `&set_specs`) [See Section ??]

#### `ex_put_concat_side_sets`

Use `ex_put_concat_sets` (`exoid`, `EX_SIDE_SET`, `set_specs`) [See Section ??]

#### `ex_put_concat_var_param`

Use `ex_put_all_var_param` (`exoid`, `num_g`, `num_n`, `num_e`, `elem_var_tab`, `0`, `0`, `0`, `0`) [See Section ??]



**ex\_put\_elem\_attr**

Use `ex_put_attr(exoid, EX_ELEM_BLOCK, elem_blk_id, attrib)` [See Section ??]

**ex\_put\_elem\_attr\_names**

Use `ex_put_attr_names(exoid, EX_ELEM_BLOCK, elem_blk_id, names)` [See Section ??]

**ex\_put\_elem\_block**

Use

`ex_put_block(exoid, EX_ELEM_BLOCK, elem_blk_id, elem_type, num_elem_this_blk, num_nodes_per_elem, 0, 0, num)`  
[See Section ??]

**ex\_put\_elem\_conn**

Use `ex_put_conn(exoid, EX_ELEM_BLOCK, elem_blk_id, connect, 0, 0)` [See Section ??]

**ex\_put\_elem\_map**

Use `ex_put_num_map(exoid, EX_ELEM_MAP, map_id, elem_map)` [See Section ??]

**ex\_put\_elem\_num\_map**

Use `ex_put_id_map(exoid, EX_ELEM_MAP, elem_map)` [See Section ??]

**ex\_put\_elem\_var**

Use

`ex_put_var(exoid, time_step, EX_ELEM_BLOCK, elem_var_index, elem_blk_id, num_elem_this_blk, elem_var_vals)`  
[See Section ??]

**ex\_put\_elem\_var\_tab**

Use

`ex_put_truth_table(exoid, EX_ELEM_BLOCK, num_elem_blk, num_elem_var, elem_var_tab)`  
[See Section ??]

**ex\_put\_glob\_vars**

Use `ex_put_var(exoid, time_step, EX_GLOBAL, 1, 0, num_glob_vars, glob_var_vals)` [See Section ??]

**ex\_put\_map**

Use `ex_put_num_map` [See Section ??]

**ex\_put\_node\_map**

Use `ex_put_num_map(exoid, EX_NODE_MAP, map_id, node_map)` [See Section ??]

**ex\_put\_node\_num\_map**

Use `ex_put_id_map(exoid, EX_NODE_MAP, node_map)` [See Section ??]

**ex\_put\_node\_set**

Use `ex_put_set(exoid, EX_NODE_SET, node_set_id, node_set_node_list, NULL)` [See Section ??]

**ex\_put\_node\_set\_dist\_fact**

Use `ex_put_set_dist_fact(exoid, EX_NODE_SET, node_set_id, node_set_dist_fact)` [See

Section ??]

#### `ex_put_node_set_param`

Use `ex_put_set_param(exoid, EX_NODE_SET, node_set_id, num_nodes_in_set, num_dist_in_set)`  
[See Section ??]

#### `ex_put_nset_var`

Use  
`ex_put_var(exoid, time_step, EX_NODE_SET, nset_var_index, nset_id, num_nodes_this_nset, nset_var_vals)`  
[See Section ??]

#### `ex_put_nset_var_tab`

Use `ex_put_truth_table(exoid, EX_NODE_SET, num_nset, num_nset_var, nset_var_tab)` [See Section ??]

#### `ex_put_one_elem_attr`

Use `ex_put_one_attr(exoid, EX_ELEM_BLOCK, elem_blk_id, attrib_index, attrib)` [See Section ??]

#### `ex_put_side_set`

Use `ex_put_set(exoid, EX_SIDE_SET, side_set_id, side_set_elem_list, side_set_side_list)` [See Section ??]

#### `ex_put_side_set_dist_fact`

Use `ex_put_set_dist_fact(exoid, EX_SIDE_SET, side_set_id, side_set_dist_fact)` [See Section ??]

#### `ex_put_side_set_param`

Use  
`ex_put_set_param(exoid, EX_SIDE_SET, side_set_id, num_side_in_set, num_dist_fact_in_set)`  
[See Section ??]

#### `ex_put_sset_var`

Use  
`ex_put_var(exoid, time_step, EX_SIDE_SET, sset_var_index, sset_id, num_faces_this_sset, sset_var_vals)`  
[See Section ??]

#### `ex_put_sset_var_tab`

Use `ex_put_truth_table(exoid, EX_SIDE_SET, num_sset, num_sset_var, sset_var_tab)` [See Section ??]

#### `ex_put_var_name`

use `ex_put_variable_name(exoid, obj_type, var_num, *var_name)` [See Section ??]

#### `ex_put_var_names`

Use `ex_put_variable_names(exoid, obj_type, num_vars, var_names)` [See Section 5.3.3]

#### `ex_put_var_param`

Use `ex_put_variable_param(exoid, obj_type, num_vars)` [See Section 5.3.1]

`ex_put_var_tab`

Use `ex_put_truth_table(exoid, obj_type, num_blk, num_var, var_tab)` [See Section ??]

# Appendix C

## Sample Code

This appendix contains examples of C programs that use the EXODUS API.

### C.1 Write Example Code

The following is a C program that creates and populates an EXODUS file:

```
1 #include "exodusII.h"
2 #include <stdio.h>
3 #include <stdlib.h>
4
5 int main(int argc, char **argv)
6 {
7     int    exoid, num_dim, num_nodes, num_elem, num_elem_blk;
8     int    num_elem_in_block[10], num_nodes_per_elem[10];
9     int    num_face_in_sset[10], num_nodes_in_nset[10];
10    int    num_node_sets, num_side_sets;
11    int    i, j, k, m, *elem_map, *connect;
12    int    node_list[100], elem_list[100], side_list[100];
13    int    ebids[10], ssids[10], nsids[10];
14    int    num_qa_rec, num_info;
15    int    num_glo_vars, num_nod_vars, num_ele_vars, num_sset_vars, num_nset_vars;
16    int    *truth_tab;
17    int    whole_time_step, num_time_steps;
18    int    CPU_word_size, IO_word_size;
19    int    prop_array[2];
20
21    float  *glob_var_vals, *nodal_var_vals, *elem_var_vals;
22    float  *sset_var_vals, *nset_var_vals;
23    float  time_value;
24    float  x[100], y[100], z[100];
25    float  attrib[1], dist_fact[100];
26    char  * coord_names[3], *qa_record[2][4], *info[3], *var_names[3];
27    char  * block_names[10], *nset_names[10], *sset_names[10];
28    char  * prop_names[2], *attrib_names[2];
29
30    ex_opts(EX_VERBOSE | EX_ABORT);
```

```

31
32  /* Specify compute and i/o word size */
33
34  CPU_word_size = 0; /* sizeof(float) */
35  IO_word_size  = 4; /* (4 bytes) */
36
37  /* create EXODUS II file */
38  exoid = ex_create("test.exo",      /* filename path */
39                  EX_CLOBBER,        /* create mode */
40                  &CPU_word_size,    /* CPU float word size in bytes */
41                  &IO_word_size);    /* I/O float word size in bytes */
42  printf("CPU word size: %d IO word size: %d\n", CPU_word_size, IO_word_size);
43
44  /* initialize file with parameters */
45  num_dim      = 3;
46  num_nodes    = 33;
47  num_elem     = 7;
48  num_elem_blk = 7;
49  num_node_sets = 2;
50  num_side_sets = 5;
51
52  ex_put_init(exoid, "This is a test", num_dim, num_nodes, num_elem, num_elem_blk, num_node_sets,
53              num_side_sets);
54
55  /* write nodal coordinates values and names to database */
56
57  /* Quad #1 */
58  x[0] = 0.0;
59  y[0] = 0.0;
60  z[0] = 0.0;
61  x[1] = 1.0;
62  y[1] = 0.0;
63  z[1] = 0.0;
64  x[2] = 1.0;
65  y[2] = 1.0;
66  z[2] = 0.0;
67  x[3] = 0.0;
68  y[3] = 1.0;
69  z[3] = 0.0;
70
71  /* Quad #2 */
72  x[4] = 1.0;
73  y[4] = 0.0;
74  z[4] = 0.0;
75  x[5] = 2.0;
76  y[5] = 0.0;
77  z[5] = 0.0;
78  x[6] = 2.0;
79  y[6] = 1.0;
80  z[6] = 0.0;
81  x[7] = 1.0;
82  y[7] = 1.0;
83  z[7] = 0.0;
84
85  /* Hex #1 */

```

```
86  x[8]  = 0.0;
87  y[8]  = 0.0;
88  z[8]  = 0.0;
89  x[9]  = 10.0;
90  y[9]  = 0.0;
91  z[9]  = 0.0;
92  x[10] = 10.0;
93  y[10] = 0.0;
94  z[10] = -10.0;
95  x[11] = 1.0;
96  y[11] = 0.0;
97  z[11] = -10.0;
98  x[12] = 1.0;
99  y[12] = 10.0;
100 z[12] = 0.0;
101 x[13] = 10.0;
102 y[13] = 10.0;
103 z[13] = 0.0;
104 x[14] = 10.0;
105 y[14] = 10.0;
106 z[14] = -10.0;
107 x[15] = 1.0;
108 y[15] = 10.0;
109 z[15] = -10.0;
110
111 /* Tetra #1 */
112 x[16] = 0.0;
113 y[16] = 0.0;
114 z[16] = 0.0;
115 x[17] = 1.0;
116 y[17] = 0.0;
117 z[17] = 5.0;
118 x[18] = 10.0;
119 y[18] = 0.0;
120 z[18] = 2.0;
121 x[19] = 7.0;
122 y[19] = 5.0;
123 z[19] = 3.0;
124
125 /* Wedge #1 */
126 x[20] = 3.0;
127 y[20] = 0.0;
128 z[20] = 6.0;
129 x[21] = 6.0;
130 y[21] = 0.0;
131 z[21] = 0.0;
132 x[22] = 0.0;
133 y[22] = 0.0;
134 z[22] = 0.0;
135 x[23] = 3.0;
136 y[23] = 2.0;
137 z[23] = 6.0;
138 x[24] = 6.0;
139 y[24] = 2.0;
140 z[24] = 2.0;
```

```

141     x[25] = 0.0;
142     y[25] = 2.0;
143     z[25] = 0.0;
144
145     /* Tetra #2 */
146     x[26] = 2.7;
147     y[26] = 1.7;
148     z[26] = 2.7;
149     x[27] = 6.0;
150     y[27] = 1.7;
151     z[27] = 3.3;
152     x[28] = 5.7;
153     y[28] = 1.7;
154     z[28] = 1.7;
155     x[29] = 3.7;
156     y[29] = 0.0;
157     z[29] = 2.3;
158
159     /* 3d Tri */
160     x[30] = 0.0;
161     y[30] = 0.0;
162     z[30] = 0.0;
163     x[31] = 10.0;
164     y[31] = 0.0;
165     z[31] = 0.0;
166     x[32] = 10.0;
167     y[32] = 10.0;
168     z[32] = 10.0;
169
170     ex_put_coord(exoid, x, y, z);
171
172     coord_names[0] = "x";
173     coord_names[1] = "y";
174     coord_names[2] = "z";
175
176     ex_put_coord_names(exoid, coord_names);
177
178     /* Add nodal attributes */
179     ex_put_attr_param(exoid, EX_NODAL, 0, 2);
180
181     ex_put_one_attr(exoid, EX_NODAL, 0, 1, x);
182     ex_put_one_attr(exoid, EX_NODAL, 0, 2, y);
183
184     attrib_names[0] = "Node_attr_1";
185     attrib_names[1] = "Node_attr_2";
186     ex_put_attr_names(exoid, EX_NODAL, 0, attrib_names);
187
188     /* write element order map */
189
190     elem_map = (int *)calloc(num_elem, sizeof(int));
191
192     for (i = 1; i <= num_elem; i++) {
193         elem_map[i - 1] = i;
194     }
195

```

```

196     ex_put_map(exoid, elem_map);
197     free(elem_map);
198
199     /* write element block parameters */
200     block_names[0] = "block_1";
201     block_names[1] = "block_2";
202     block_names[2] = "block_3";
203     block_names[3] = "block_4";
204     block_names[4] = "block_5";
205     block_names[5] = "block_6";
206     block_names[6] = "block_7";
207
208     num_elem_in_block[0] = 1;
209     num_elem_in_block[1] = 1;
210     num_elem_in_block[2] = 1;
211     num_elem_in_block[3] = 1;
212     num_elem_in_block[4] = 1;
213     num_elem_in_block[5] = 1;
214     num_elem_in_block[6] = 1;
215
216     num_nodes_per_elem[0] = 4; /* elements in block #1 are 4-node quads */
217     num_nodes_per_elem[1] = 4; /* elements in block #2 are 4-node quads */
218     num_nodes_per_elem[2] = 8; /* elements in block #3 are 8-node hexes */
219     num_nodes_per_elem[3] = 4; /* elements in block #4 are 4-node tetras */
220     num_nodes_per_elem[4] = 6; /* elements in block #5 are 6-node wedges */
221     num_nodes_per_elem[5] = 8; /* elements in block #6 are 8-node tetras */
222     num_nodes_per_elem[6] = 3; /* elements in block #7 are 3-node tris */
223
224     ebids[0] = 10;
225     ebids[1] = 11;
226     ebids[2] = 12;
227     ebids[3] = 13;
228     ebids[4] = 14;
229     ebids[5] = 15;
230     ebids[6] = 16;
231
232     ex_put_elem_block(exoid, ebids[0], "quad", num_elem_in_block[0], num_nodes_per_elem[0], 1);
233     ex_put_elem_block(exoid, ebids[1], "quad", num_elem_in_block[1], num_nodes_per_elem[1], 1);
234     ex_put_elem_block(exoid, ebids[2], "hex", num_elem_in_block[2], num_nodes_per_elem[2], 1);
235     ex_put_elem_block(exoid, ebids[3], "tetra", num_elem_in_block[3], num_nodes_per_elem[3], 1);
236
237     /* Use alternative function to do same thing... */
238     ex_put_block(exoid, EX_ELEM_BLOCK, ebids[4], "wedge", num_elem_in_block[4], num_nodes_per_e
239         0, 0, 1);
240     ex_put_block(exoid, EX_ELEM_BLOCK, ebids[5], "tetra", num_elem_in_block[5], num_nodes_per_e
241         0, 0, 1);
242     ex_put_block(exoid, EX_ELEM_BLOCK, ebids[6], "tri", num_elem_in_block[6], num_nodes_per_e
243         0, 0, 1);
244
245     /* Write element block names */
246     ex_put_names(exoid, EX_ELEM_BLOCK, block_names);
247
248     /* write element block properties */
249
250     /*
251         12345678901234567890123456789012
252     */

```



```

251 prop_names[0] = "MATERIAL_PROPERTY_LONG_NAME_32CH";
252 prop_names[1] = "DENSITY";
253 ex_put_prop_names(exoid, EX_ELEM_BLOCK, 2, prop_names);
254 ex_put_prop(exoid, EX_ELEM_BLOCK, ebids[0], prop_names[0], 10);
255 ex_put_prop(exoid, EX_ELEM_BLOCK, ebids[1], prop_names[0], 20);
256 ex_put_prop(exoid, EX_ELEM_BLOCK, ebids[2], prop_names[0], 30);
257 ex_put_prop(exoid, EX_ELEM_BLOCK, ebids[3], prop_names[0], 40);
258 ex_put_prop(exoid, EX_ELEM_BLOCK, ebids[4], prop_names[0], 50);
259 ex_put_prop(exoid, EX_ELEM_BLOCK, ebids[5], prop_names[0], 60);
260 ex_put_prop(exoid, EX_ELEM_BLOCK, ebids[6], prop_names[0], 70);
261
262 /* write element connectivity */
263 connect = (int *)calloc(8, sizeof(int));
264 connect[0] = 1;
265 connect[1] = 2;
266 connect[2] = 3;
267 connect[3] = 4;
268 ex_put_conn(exoid, EX_ELEM_BLOCK, ebids[0], connect, 0, 0);
269
270 connect[0] = 5;
271 connect[1] = 6;
272 connect[2] = 7;
273 connect[3] = 8;
274 ex_put_conn(exoid, EX_ELEM_BLOCK, ebids[1], connect, 0, 0);
275
276 connect[0] = 9;
277 connect[1] = 10;
278 connect[2] = 11;
279 connect[3] = 12;
280 connect[4] = 13;
281 connect[5] = 14;
282 connect[6] = 15;
283 connect[7] = 16;
284 ex_put_conn(exoid, EX_ELEM_BLOCK, ebids[2], connect, 0, 0);
285
286 connect[0] = 17;
287 connect[1] = 18;
288 connect[2] = 19;
289 connect[3] = 20;
290 ex_put_conn(exoid, EX_ELEM_BLOCK, ebids[3], connect, 0, 0);
291
292 connect[0] = 21;
293 connect[1] = 22;
294 connect[2] = 23;
295 connect[3] = 24;
296 connect[4] = 25;
297 connect[5] = 26;
298 ex_put_conn(exoid, EX_ELEM_BLOCK, ebids[4], connect, 0, 0);
299
300 connect[0] = 17;
301 connect[1] = 18;
302 connect[2] = 19;
303 connect[3] = 20;
304 connect[4] = 27;
305 connect[5] = 28;

```

```

306 connect[6] = 30;
307 connect[7] = 29;
308 ex_put_conn(exoid, EX_ELEM_BLOCK, ebids[5], connect, 0, 0);
309
310 /* Use "old" API function just to show syntax */
311 connect[0] = 31;
312 connect[1] = 32;
313 connect[2] = 33;
314 ex_put_elem_conn(exoid, ebids[6], connect);
315
316 free(connect);
317
318 /* write element block attributes */
319 attrib[0] = 3.14159;
320 ex_put_attr(exoid, EX_ELEM_BLOCK, ebids[0], attrib);
321 ex_put_attr(exoid, EX_ELEM_BLOCK, ebids[0], attrib);
322
323 attrib[0] = 6.14159;
324 ex_put_attr(exoid, EX_ELEM_BLOCK, ebids[1], attrib);
325 ex_put_attr(exoid, EX_ELEM_BLOCK, ebids[2], attrib);
326 ex_put_attr(exoid, EX_ELEM_BLOCK, ebids[3], attrib);
327 ex_put_attr(exoid, EX_ELEM_BLOCK, ebids[4], attrib);
328 ex_put_attr(exoid, EX_ELEM_BLOCK, ebids[5], attrib);
329 ex_put_attr(exoid, EX_ELEM_BLOCK, ebids[6], attrib);
330
331 attrib_names[0] = "THICKNESS";
332 for (i = 0; i < 7; i++) {
333     ex_put_attr_names(exoid, EX_ELEM_BLOCK, ebids[i], attrib_names);
334 }
335
336 /* write individual node sets */
337 num_nodes_in_nset[0] = 5;
338 num_nodes_in_nset[1] = 3;
339
340 nsids[0] = 20;
341 nsids[1] = 21;
342
343 ex_put_set_param(exoid, EX_NODE_SET, nsids[0], 5, 5);
344
345 node_list[0] = 100;
346 node_list[1] = 101;
347 node_list[2] = 102;
348 node_list[3] = 103;
349 node_list[4] = 104;
350 ex_put_set(exoid, EX_NODE_SET, nsids[0], node_list, 0);
351
352 dist_fact[0] = 1.0;
353 dist_fact[1] = 2.0;
354 dist_fact[2] = 3.0;
355 dist_fact[3] = 4.0;
356 dist_fact[4] = 5.0;
357 ex_put_set_dist_fact(exoid, EX_NODE_SET, nsids[0], dist_fact);
358
359 ex_put_set_param(exoid, EX_NODE_SET, nsids[1], 3, 3);
360

```

```

361 node_list[0] = 200;
362 node_list[1] = 201;
363 node_list[2] = 202;
364 ex_put_set(exoid, EX_NODE_SET, nsids[1], node_list, 0);
365
366 dist_fact[0] = 1.1;
367 dist_fact[1] = 2.1;
368 dist_fact[2] = 3.1;
369 ex_put_set_dist_fact(exoid, EX_NODE_SET, nsids[1], dist_fact);
370
371 /* Write node set names */
372 nset_names[0] = "nset_1";
373 nset_names[1] = "nset_2";
374
375 ex_put_names(exoid, EX_NODE_SET, nset_names);
376 ex_put_prop(exoid, EX_NODE_SET, nsids[0], "FACE", 4);
377 ex_put_prop(exoid, EX_NODE_SET, nsids[1], "FACE", 5);
378
379 prop_array[0] = 1000;
380 prop_array[1] = 2000;
381
382 ex_put_prop_array(exoid, EX_NODE_SET, "VELOCITY", prop_array);
383
384 /* Add nodeset attributes */
385 ex_put_attr_param(exoid, EX_NODE_SET, nsids[0], 1);
386 ex_put_attr(exoid, EX_NODE_SET, nsids[0], x);
387
388 attrib_names[0] = "Nodeset_attribute";
389 ex_put_attr_names(exoid, EX_NODE_SET, nsids[0], attrib_names);
390
391 /* write individual side sets */
392 num_face_in_sset[0] = 2;
393 num_face_in_sset[1] = 2;
394 num_face_in_sset[2] = 7;
395 num_face_in_sset[3] = 8;
396 num_face_in_sset[4] = 10;
397
398 ssids[0] = 30;
399 ssids[1] = 31;
400 ssids[2] = 32;
401 ssids[3] = 33;
402 ssids[4] = 34;
403
404 /* side set #1 - quad */
405 ex_put_set_param(exoid, EX_SIDE_SET, ssids[0], 2, 4);
406
407 elem_list[0] = 2;
408 elem_list[1] = 2;
409 side_list[0] = 4;
410 side_list[1] = 2;
411 dist_fact[0] = 30.0;
412 dist_fact[1] = 30.1;
413 dist_fact[2] = 30.2;
414 dist_fact[3] = 30.3;
415

```

```

416 ex_put_set(exoid, EX_SIDE_SET, 30, elem_list, side_list);
417 ex_put_set_dist_fact(exoid, EX_SIDE_SET, 30, dist_fact);
418
419 /* side set #2 - quad, spanning 2 elements */
420 ex_put_set_param(exoid, EX_SIDE_SET, 31, 2, 4);
421
422 elem_list[0] = 1;
423 elem_list[1] = 2;
424 side_list[0] = 2;
425 side_list[1] = 3;
426 dist_fact[0] = 31.0;
427 dist_fact[1] = 31.1;
428 dist_fact[2] = 31.2;
429 dist_fact[3] = 31.3;
430
431 ex_put_set(exoid, EX_SIDE_SET, 31, elem_list, side_list);
432 ex_put_set_dist_fact(exoid, EX_SIDE_SET, 31, dist_fact);
433
434 /* side set #3 - hex */
435 ex_put_set_param(exoid, EX_SIDE_SET, 32, 7, 0);
436
437 elem_list[0] = 3;
438 elem_list[1] = 3;
439 elem_list[2] = 3;
440 elem_list[3] = 3;
441 elem_list[4] = 3;
442 elem_list[5] = 3;
443 elem_list[6] = 3;
444
445 side_list[0] = 5;
446 side_list[1] = 3;
447 side_list[2] = 3;
448 side_list[3] = 2;
449 side_list[4] = 4;
450 side_list[5] = 1;
451 side_list[6] = 6;
452
453 ex_put_set(exoid, EX_SIDE_SET, 32, elem_list, side_list);
454
455 /* side set #4 - tetras */
456 ex_put_set_param(exoid, EX_SIDE_SET, 33, 8, 0);
457
458 elem_list[0] = 4;
459 elem_list[1] = 4;
460 elem_list[2] = 4;
461 elem_list[3] = 4;
462 elem_list[4] = 6;
463 elem_list[5] = 6;
464 elem_list[6] = 6;
465 elem_list[7] = 6;
466
467 side_list[0] = 1;
468 side_list[1] = 2;
469 side_list[2] = 3;
470 side_list[3] = 4;

```

```

471 side_list[4] = 1;
472 side_list[5] = 2;
473 side_list[6] = 3;
474 side_list[7] = 4;
475
476 ex_put_set(exoid, EX_SIDE_SET, 33, elem_list, side_list);
477
478 /* side set #5 - wedges and tris */
479 ex_put_set_param(exoid, EX_SIDE_SET, 34, 10, 0);
480
481 elem_list[0] = 5;
482 elem_list[1] = 5;
483 elem_list[2] = 5;
484 elem_list[3] = 5;
485 elem_list[4] = 5;
486 elem_list[5] = 7;
487 elem_list[6] = 7;
488 elem_list[7] = 7;
489 elem_list[8] = 7;
490 elem_list[9] = 7;
491
492 side_list[0] = 1;
493 side_list[1] = 2;
494 side_list[2] = 3;
495 side_list[3] = 4;
496 side_list[4] = 5;
497 side_list[5] = 1;
498 side_list[6] = 2;
499 side_list[7] = 3;
500 side_list[8] = 4;
501 side_list[9] = 5;
502
503 ex_put_set(exoid, EX_SIDE_SET, 34, elem_list, side_list);
504
505 /* Write side set names */
506 sset_names[0] = "sset_1";
507 sset_names[1] = "sset_2";
508 sset_names[2] = "sset_3";
509 sset_names[3] = "sset_4";
510 sset_names[4] = "sset_5";
511
512 ex_put_names(exoid, EX_SIDE_SET, sset_names);
513 ex_put_prop(exoid, EX_SIDE_SET, 30, "COLOR", 100);
514 ex_put_prop(exoid, EX_SIDE_SET, 31, "COLOR", 101);
515
516 /* write QA records; test empty and just blank-filled records */
517 num_qa_rec = 2;
518
519 qa_record[0][0] = "TESTWT";
520 qa_record[0][1] = "testwt";
521 qa_record[0][2] = "07/07/93";
522 qa_record[0][3] = "15:41:33";
523 qa_record[1][0] = "";
524 qa_record[1][1] = "____________________";
525 qa_record[1][2] = "";

```

```

526 qa_record[1][3] = "aaaaaaaaaaaaaaaaaaaaaaaa";
527
528 ex_put_qa(exoid, num_qa_rec, qa_record);
529
530 /* write information records; test empty and just blank-filled records */
531 num_info = 3;
532
533 info[0] = "This is the first information record.";
534 info[1] = "";
535 info[2] = "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa";
536
537 ex_put_info(exoid, num_info, info);
538
539 /* write results variables parameters and names */
540 num_glo_vars = 1;
541 var_names[0] = "glo_vars";
542 ex_put_variable_param(exoid, EX_GLOBAL, num_glo_vars);
543 ex_put_variable_names(exoid, EX_GLOBAL, num_glo_vars, var_names);
544
545 num_nod_vars = 2;
546 /*      12345678901234567890123456789012 */
547 var_names[0] = "node_variable_a_very_long_name_0";
548 var_names[1] = "nod_var1";
549
550 ex_put_variable_param(exoid, EX_NODAL, num_nod_vars);
551 ex_put_variable_names(exoid, EX_NODAL, num_nod_vars, var_names);
552
553 num_ele_vars = 3;
554 var_names[0] = "ele_var0";
555 var_names[1] = "ele_var1";
556 var_names[2] = "ele_var2";
557
558 ex_put_variable_param(exoid, EX_ELEM_BLOCK, num_ele_vars);
559 ex_put_variable_names(exoid, EX_ELEM_BLOCK, num_ele_vars, var_names);
560
561 num_nset_vars = 3;
562 var_names[0] = "ns_var0";
563 var_names[1] = "ns_var1";
564 var_names[2] = "ns_var2";
565
566 ex_put_variable_param(exoid, EX_NODE_SET, num_nset_vars);
567 ex_put_variable_names(exoid, EX_NODE_SET, num_nset_vars, var_names);
568
569 num_sset_vars = 3;
570 var_names[0] = "ss_var0";
571 var_names[1] = "ss_var1";
572 var_names[2] = "ss_var2";
573
574 ex_put_variable_param(exoid, EX_SIDE_SET, num_sset_vars);
575 ex_put_variable_names(exoid, EX_SIDE_SET, num_sset_vars, var_names);
576
577 /* write element variable truth table */
578 truth_tab = (int *)calloc((num_elem_blk * num_ele_vars), sizeof(int));
579
580 k = 0;

```

```

581     for (i = 0; i < num_elem_blk; i++) {
582         for (j = 0; j < num_ele_vars; j++) {
583             truth_tab[k++] = 1;
584         }
585     }
586
587     ex_put_truth_table(exoid, EX_ELEM_BLOCK, num_elem_blk, num_ele_vars, truth_tab);
588     free(truth_tab);
589
590     /* for each time step, write the analysis results;
591      * the code below fills the arrays glob_var_vals,
592      * nodal_var_vals, and elem_var_vals with values for debugging purposes;
593      * obviously the analysis code will populate these arrays
594      */
595
596     whole_time_step = 1;
597     num_time_steps = 10;
598
599     glob_var_vals = (float *)calloc(num_glo_vars, CPU_word_size);
600     nodal_var_vals = (float *)calloc(num_nodes, CPU_word_size);
601     elem_var_vals = (float *)calloc(4, CPU_word_size);
602     sset_var_vals = (float *)calloc(10, CPU_word_size);
603     nset_var_vals = (float *)calloc(10, CPU_word_size);
604
605     for (i = 0; i < num_time_steps; i++) {
606         time_value = (float)(i + 1) / 100.;
607
608         /* write time value */
609         ex_put_time(exoid, whole_time_step, &time_value);
610
611         /* write global variables */
612         for (j = 0; j < num_glo_vars; j++) {
613             glob_var_vals[j] = (float)(j + 2) * time_value;
614         }
615
616         ex_put_var(exoid, whole_time_step, EX_GLOBAL, 1, 0, num_glo_vars, glob_var_vals);
617
618         /* write nodal variables */
619         for (k = 1; k <= num_nod_vars; k++) {
620             for (j = 0; j < num_nodes; j++) {
621                 nodal_var_vals[j] = (float)k + ((float)(j + 1) * time_value);
622             }
623             ex_put_var(exoid, whole_time_step, EX_NODAL, k, 1, num_nodes, nodal_var_vals);
624         }
625
626         /* write element variables */
627         for (k = 1; k <= num_ele_vars; k++) {
628             for (j = 0; j < num_elem_blk; j++) {
629                 for (m = 0; m < num_elem_in_block[j]; m++) {
630                     elem_var_vals[m] = (float)(k + 1) + (float)(j + 2) + ((float)(m + 1) * time_value);
631                 }
632                 ex_put_var(exoid, whole_time_step, EX_ELEM_BLOCK, k, ebids[j], num_elem_in_block[j],
633                     elem_var_vals);
634             }
635         }

```

```

636
637     /* write sideset variables */
638     for (k = 1; k <= num_sset_vars; k++) {
639         for (j = 0; j < num_side_sets; j++) {
640             for (m = 0; m < num_face_in_sset[j]; m++) {
641                 sset_var_vals[m] = (float)(k + 2) + (float)(j + 3) + ((float)(m + 1) * time_value);
642             }
643             ex_put_var(exoid, whole_time_step, EX_SIDE_SET, k, ssids[j], num_face_in_sset[j],
644                 sset_var_vals);
645         }
646     }
647
648     /* write nodeset variables */
649     for (k = 1; k <= num_nset_vars; k++) {
650         for (j = 0; j < num_node_sets; j++) {
651             for (m = 0; m < num_nodes_in_nset[j]; m++) {
652                 nset_var_vals[m] = (float)(k + 3) + (float)(j + 4) + ((float)(m + 1) * time_value);
653             }
654             ex_put_var(exoid, whole_time_step, EX_NODE_SET, k, nsids[j], num_nodes_in_nset[j],
655                 nset_var_vals);
656         }
657     }
658
659     whole_time_step++;
660
661     /* update the data file; this should be done at the end of every
662      * time step to ensure that no data is lost if the analysis dies
663      */
664     ex_update(exoid);
665 }
666 free(glob_var_vals);
667 free(nodal_var_vals);
668 free(elem_var_vals);
669 free(sset_var_vals);
670 free(nset_var_vals);
671
672 /* close the EXODUS files */
673 ex_close(exoid);
674 return 0;
675 }

```



## C.2 Read Example Code

The following C program reads data from an EXODUS file:

```

1  #include "exodusII.h"
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <string.h>
5
6  int main(int argc, char **argv)
7  {
8      int    exoid, num_dim, num_nodes, num_elem, num_elem_blk, num_node_sets;
9      int    num_side_sets, error;
10     int    i, j;
11     int    *elem_map, *connect, *node_list, *node_ctr_list, *elem_list, *side_list;
12     int    *ids;
13     int    num_qa_rec, num_info;
14     int    num_glo_vars, num_nod_vars, num_ele_vars;
15     int    num_nset_vars, num_sset_vars;
16     int    *truth_tab;
17     int    num_time_steps;
18     int    *num_elem_in_block, *num_nodes_per_elem, *num_attr;
19     int    num_nodes_in_set, num_elem_in_set;
20     int    num_sides_in_set, num_df_in_set;
21     int    list_len, elem_list_len, df_list_len;
22     int    node_num, time_step, var_index, beg_time, end_time, elem_num;
23     int    CPU_word_size, IO_word_size;
24     int    num_props, prop_value, *prop_values;
25     int    idum;
26
27     ex_set_specs ss_specs, ns_specs;
28
29     float    time_value, *time_values, *var_values;
30     float    *x, *y, *z;
31     float    *attrib, *dist_fact;
32     float    version;
33
34     char    *coord_names[3], *qa_record[2][4], *info[3], *var_names[3];
35     char    *block_names[10], *nset_names[10], *sset_names[10];
36     char    *attrib_names[10];
37     char    name[MAX_STR_LENGTH + 1];
38     char    title[MAX_LINE_LENGTH + 1], elem_type[MAX_STR_LENGTH + 1];
39     char    *cdum;
40     char    *prop_names[3];
41
42     cdum = 0;
43
44     CPU_word_size = 0; /* sizeof(float) */
45     IO_word_size  = 0; /* use what is stored in file */
46
47     ex_opts(EX_VERBOSE | EX_ABORT);
48
49     /* open EXODUS II files */
50     exoid = ex_open("test.exo",      /* filename path */
51                    EX_READ,          /* access mode = READ */
52                    &CPU_word_size, /* CPU word size */

```

```

53         &IO_word_size, /* IO word size */
54         &version);      /* ExodusII library version */
55
56     if (exoid < 0)
57         exit(1);
58
59     printf("test.exo is an EXODUSII file; version %4.2f\n", version);
60     printf("IO word size %1d\n", IO_word_size);
61     ex_inquire(exoid, EX_INQ_API_VERS, &idum, &version, cdum);
62     printf("EXODUSII API; version %4.2f\n", version);
63
64     ex_inquire(exoid, EX_INQ_LIB_VERS, &idum, &version, cdum);
65     printf("EXODUSII Library API; version %4.2f (%d)\n", version, idum);
66
67     /* read database parameters */
68     ex_get_init(exoid, title, &num_dim, &num_nodes, &num_elem, &num_elem_blk, &num_node_sets,
69                &num_side_sets);
70
71     printf("database parameters:\n");
72     printf("title = '%s'\n", title);
73     printf("num_dim = %3d\n", num_dim);
74     printf("num_nodes = %3d\n", num_nodes);
75     printf("num_elem = %3d\n", num_elem);
76     printf("num_elem_blk = %3d\n", num_elem_blk);
77     printf("num_node_sets = %3d\n", num_node_sets);
78     printf("num_side_sets = %3d\n", num_side_sets);
79
80     /* read nodal coordinates values and names from database */
81     x = (float *)calloc(num_nodes, sizeof(float));
82     y = (float *)calloc(num_nodes, sizeof(float));
83     if (num_dim >= 3)
84         z = (float *)calloc(num_nodes, sizeof(float));
85     else
86         z = 0;
87
88     ex_get_coord(exoid, x, y, z);
89     free(x);
90     free(y);
91     if (num_dim >= 3)
92         free(z);
93
94     for (i = 0; i < num_dim; i++) {
95         coord_names[i] = (char *)calloc((MAX_STR_LENGTH + 1), sizeof(char));
96     }
97
98     ex_get_coord_names(exoid, coord_names);
99
100    for (i = 0; i < num_dim; i++)
101        free(coord_names[i]);
102
103    {
104        int num_attrs = 0;
105        ex_get_attr_param(exoid, EX_NODAL, 0, &num_attrs);
106        if (num_attrs > 0) {
107            for (j = 0; j < num_attrs; j++) {

```

```

108     attrib_names[j] = (char *)calloc((MAX_STR_LENGTH + 1), sizeof(char));
109 }
110 error = ex_get_attr_names(exoid, EX_NODAL, 0, attrib_names);
111
112 if (error == 0) {
113     attrib = (float *)calloc(num_nodes, sizeof(float));
114     for (j = 0; j < num_attrs; j++) {
115         ex_get_one_attr(exoid, EX_NODAL, 0, j + 1, attrib);
116         free(attrib_names[j]);
117     }
118     free(attrib);
119 }
120 }
121 }
122
123 /* read element order map */
124 elem_map = (int *)calloc(num_elem, sizeof(int));
125 ex_get_map(exoid, elem_map);
126 free(elem_map);
127
128 /* read element block parameters */
129 if (num_elem_blk > 0) {
130     ids = (int *)calloc(num_elem_blk, sizeof(int));
131     num_elem_in_block = (int *)calloc(num_elem_blk, sizeof(int));
132     num_nodes_per_elem = (int *)calloc(num_elem_blk, sizeof(int));
133     num_attr = (int *)calloc(num_elem_blk, sizeof(int));
134
135     ex_get_ids(exoid, EX_ELEM_BLOCK, ids);
136
137     for (i = 0; i < num_elem_blk; i++) {
138         block_names[i] = (char *)calloc((MAX_STR_LENGTH + 1), sizeof(char));
139     }
140
141     ex_get_names(exoid, EX_ELEM_BLOCK, block_names);
142
143     for (i = 0; i < num_elem_blk; i++) {
144         ex_get_name(exoid, EX_ELEM_BLOCK, ids[i], name);
145         /* 'name' should equal 'block_names[i]' at this point */
146
147         ex_get_elem_block(exoid, ids[i], elem_type, &(num_elem_in_block[i]), &(num_nodes_per_e
148             &(num_attr[i]));
149         free(block_names[i]);
150     }
151
152     /* read element block properties */
153     num_props = ex_inquire_int(exoid, EX_INQ_EB_PROP);
154     for (i = 0; i < num_props; i++) {
155         prop_names[i] = (char *)calloc((MAX_STR_LENGTH + 1), sizeof(char));
156     }
157
158     ex_get_prop_names(exoid, EX_ELEM_BLOCK, prop_names);
159
160     for (i = 1; i < num_props; i++) { /* Prop 1 is id; skip that here */
161         for (j = 0; j < num_elem_blk; j++) {
162             ex_get_prop(exoid, EX_ELEM_BLOCK, ids[j], prop_names[i], &prop_value);

```

```

163     }
164 }
165
166 for (i = 0; i < num_props; i++)
167     free(prop_names[i]);
168 }
169
170 /* read element connectivity */
171 for (i = 0; i < num_elem_blk; i++) {
172     if (num_elem_in_block[i] > 0) {
173         connect = (int *)calloc((num_nodes_per_elem[i] * num_elem_in_block[i]), sizeof(int));
174
175         ex_get_conn(exoid, EX_ELEM_BLOCK, ids[i], connect, 0, 0);
176         free(connect);
177     }
178 }
179
180 /* read element block attributes */
181 for (i = 0; i < num_elem_blk; i++) {
182     if (num_elem_in_block[i] > 0) {
183         for (j = 0; j < num_attr[i]; j++)
184             attrib_names[j] = (char *)calloc((MAX_STR_LENGTH + 1), sizeof(char));
185
186         attrib = (float *)calloc(num_attr[i] * num_elem_in_block[i], sizeof(float));
187         error = ex_get_attr(exoid, EX_ELEM_BLOCK, ids[i], attrib);
188
189         if (error == 0) {
190             ex_get_attr_names(exoid, EX_ELEM_BLOCK, ids[i], attrib_names);
191         }
192         free(attrib);
193         for (j = 0; j < num_attr[i]; j++)
194             free(attrib_names[j]);
195     }
196 }
197
198 if (num_elem_blk > 0) {
199     free(ids);
200     free(num_nodes_per_elem);
201     free(num_attr);
202 }
203
204 /* read individual node sets */
205 if (num_node_sets > 0) {
206     ids = (int *)calloc(num_node_sets, sizeof(int));
207
208     ex_get_ids(exoid, EX_NODE_SET, ids);
209
210     for (i = 0; i < num_node_sets; i++) {
211         nset_names[i] = (char *)calloc((MAX_STR_LENGTH + 1), sizeof(char));
212     }
213
214     /* Get all nodeset names in one call */
215     ex_get_names(exoid, EX_NODE_SET, nset_names);
216
217     for (i = 0; i < num_node_sets; i++) {

```

```

218     /* Can also get the names one at a time... */
219     ex_get_name(exoid, EX_NODE_SET, ids[i], name);
220     /* 'name' should equal 'block_names[i]' at this point */
221
222     ex_get_set_param(exoid, EX_NODE_SET, ids[i], &num_nodes_in_set, &num_df_in_set);
223     free(nset_names[i]);
224     node_list = (int *)calloc(num_nodes_in_set, sizeof(int));
225     dist_fact = (float *)calloc(num_nodes_in_set, sizeof(float));
226
227     ex_get_set(exoid, EX_NODE_SET, ids[i], node_list, 0);
228
229     if (num_df_in_set > 0) {
230         ex_get_set_dist_fact(exoid, EX_NODE_SET, ids[i], dist_fact);
231     }
232
233     free(node_list);
234     free(dist_fact);
235
236     {
237         int num_attrs = 0;
238         ex_get_attr_param(exoid, EX_NODE_SET, ids[i], &num_attrs);
239         if (num_attrs > 0) {
240             for (j = 0; j < num_attrs; j++) {
241                 attrib_names[j] = (char *)calloc((MAX_STR_LENGTH + 1), sizeof(char));
242             }
243
244             error = ex_get_attr_names(exoid, EX_NODE_SET, ids[i], attrib_names);
245             if (error == 0) {
246                 attrib = (float *)calloc(num_nodes_in_set, sizeof(float));
247                 for (j = 0; j < num_attrs; j++) {
248                     ex_get_one_attr(exoid, EX_NODE_SET, ids[i], j + 1, attrib);
249                     free(attrib_names[j]);
250                 }
251                 free(attrib);
252             }
253         }
254     }
255     free(ids);
256
257     /* read node set properties */
258     num_props = ex_inquire_int(exoid, EX_INQ_NS_PROP);
259
260     for (i = 0; i < num_props; i++) {
261         prop_names[i] = (char *)calloc((MAX_STR_LENGTH + 1), sizeof(char));
262     }
263     prop_values = (int *)calloc(num_node_sets, sizeof(int));
264
265     ex_get_prop_names(exoid, EX_NODE_SET, prop_names);
266
267     for (i = 0; i < num_props; i++) {
268         ex_get_prop_array(exoid, EX_NODE_SET, prop_names[i], prop_values);
269     }
270     for (i = 0; i < num_props; i++)
271         free(prop_names[i]);
272

```

```

273     free(prop_values);
274
275     /* read concatenated node sets; this produces the same information as
276      * the above code which reads individual node sets
277      */
278     {
279         num_node_sets          = ex_inquire_int(exoid, EX_INQ_NODE_SETS);
280         ns_specs.sets_ids      = (int *)calloc(num_node_sets, sizeof(int));
281         ns_specs.num_entries_per_set = (int *)calloc(num_node_sets, sizeof(int));
282         ns_specs.num_dist_per_set  = (int *)calloc(num_node_sets, sizeof(int));
283         ns_specs.sets_entry_index  = (int *)calloc(num_node_sets, sizeof(int));
284         ns_specs.sets_dist_index   = (int *)calloc(num_node_sets, sizeof(int));
285
286         list_len               = ex_inquire_int(exoid, EX_INQ_NS_NODE_LEN);
287         ns_specs.sets_entry_list = (int *)calloc(list_len, sizeof(int));
288
289         list_len               = ex_inquire_int(exoid, EX_INQ_NS_DF_LEN);
290         ns_specs.sets_dist_fact = (float *)calloc(list_len, sizeof(float));
291         ns_specs.sets_extra_list = NULL;
292
293         ex_get_concat_sets(exoid, EX_NODE_SET, &ns_specs);
294     }
295 }
296
297 /* read individual side sets */
298 if (num_side_sets > 0) {
299     ids = (int *)calloc(num_side_sets, sizeof(int));
300
301     ex_get_ids(exoid, EX_SIDE_SET, ids);
302     for (i = 0; i < num_side_sets; i++) {
303         sset_names[i] = (char *)calloc((MAX_STR_LENGTH + 1), sizeof(char));
304     }
305
306     ex_get_names(exoid, EX_SIDE_SET, sset_names);
307
308     for (i = 0; i < num_side_sets; i++) {
309         ex_get_name(exoid, EX_SIDE_SET, ids[i], name);
310
311         ex_get_set_param(exoid, EX_SIDE_SET, ids[i], &num_sides_in_set, &num_df_in_set);
312         free(sset_names[i]);
313
314         /* Note: The # of elements is same as # of sides! */
315         num_elem_in_set = num_sides_in_set;
316         elem_list       = (int *)calloc(num_elem_in_set, sizeof(int));
317         side_list       = (int *)calloc(num_sides_in_set, sizeof(int));
318         node_ctr_list   = (int *)calloc(num_elem_in_set, sizeof(int));
319         node_list       = (int *)calloc(num_elem_in_set * 21, sizeof(int));
320         dist_fact       = (float *)calloc(num_df_in_set, sizeof(float));
321
322         ex_get_set(exoid, EX_SIDE_SET, ids[i], elem_list, side_list);
323         ex_get_side_set_node_list(exoid, ids[i], node_ctr_list, node_list);
324
325         if (num_df_in_set > 0) {
326             ex_get_set_dist_fact(exoid, EX_SIDE_SET, ids[i], dist_fact);
327         }

```

```

328     free(elem_list);
329     free(side_list);
330     free(node_ctr_list);
331     free(node_list);
332     free(dist_fact);
333 }
334
335
336 /* read side set properties */
337 num_props = ex_inquire_int(exoid, EX_INQ_SS_PROP);
338
339 for (i = 0; i < num_props; i++) {
340     prop_names[i] = (char *)calloc((MAX_STR_LENGTH + 1), sizeof(char));
341 }
342
343 ex_get_prop_names(exoid, EX_SIDE_SET, prop_names);
344
345 for (i = 0; i < num_props; i++) {
346     for (j = 0; j < num_side_sets; j++) {
347         ex_get_prop(exoid, EX_SIDE_SET, ids[j], prop_names[i], &prop_value);
348     }
349 }
350 for (i = 0; i < num_props; i++)
351     free(prop_names[i]);
352 free(ids);
353
354 num_side_sets = ex_inquire_int(exoid, EX_INQ_SIDE_SETS);
355
356 if (num_side_sets > 0) {
357     elem_list_len = ex_inquire_int(exoid, EX_INQ_SS_ELEM_LEN);
358     df_list_len   = ex_inquire_int(exoid, EX_INQ_SS_DF_LEN);
359 }
360
361 /* read concatenated side sets; this produces the same information as
362  * the above code which reads individual side sets
363  */
364
365 /* concatenated side set read */
366
367 if (num_side_sets > 0) {
368     ss_specs.sets_ids           = (int *)calloc(num_side_sets, sizeof(int));
369     ss_specs.num_entries_per_set = (int *)calloc(num_side_sets, sizeof(int));
370     ss_specs.num_dist_per_set   = (int *)calloc(num_side_sets, sizeof(int));
371     ss_specs.sets_entry_index   = (int *)calloc(num_side_sets, sizeof(int));
372     ss_specs.sets_dist_index    = (int *)calloc(num_side_sets, sizeof(int));
373     ss_specs.sets_entry_list    = (int *)calloc(elem_list_len, sizeof(int));
374     ss_specs.sets_extra_list    = (int *)calloc(elem_list_len, sizeof(int));
375     ss_specs.sets_dist_fact     = (float *)calloc(df_list_len, sizeof(float));
376
377     ex_get_concat_sets(exoid, EX_SIDE_SET, &ss_specs);
378 }
379 }
380 /* end of concatenated side set read */
381
382 /* read QA records */

```

```

383 num_qa_rec = ex_inquire_int(exoid, EX_INQ_QA);
384
385 for (i = 0; i < num_qa_rec; i++) {
386     for (j = 0; j < 4; j++) {
387         qa_record[i][j] = (char *)calloc((MAX_STR_LENGTH + 1), sizeof(char));
388     }
389 }
390
391 ex_get_qa(exoid, qa_record);
392
393 /* read information records */
394 num_info = ex_inquire_int(exoid, EX_INQ_INFO);
395 for (i = 0; i < num_info; i++) {
396     info[i] = (char *)calloc((MAX_LINE_LENGTH + 1), sizeof(char));
397 }
398
399 ex_get_info(exoid, info);
400
401 for (i = 0; i < num_info; i++) {
402     free(info[i]);
403 }
404
405 /* read global variables parameters and names */
406 ex_get_variable_param(exoid, EX_GLOBAL, &num_glo_vars);
407
408 for (i = 0; i < num_glo_vars; i++) {
409     var_names[i] = (char *)calloc((MAX_STR_LENGTH + 1), sizeof(char));
410 }
411
412 ex_get_variable_names(exoid, EX_GLOBAL, num_glo_vars, var_names);
413
414 for (i = 0; i < num_glo_vars; i++) {
415     free(var_names[i]);
416 }
417
418 /* read nodal variables parameters and names */
419 num_nod_vars = 0;
420 if (num_nodes > 0) {
421     ex_get_variable_param(exoid, EX_NODAL, &num_nod_vars);
422
423     for (i = 0; i < num_nod_vars; i++) {
424         var_names[i] = (char *)calloc((MAX_STR_LENGTH + 1), sizeof(char));
425     }
426
427     ex_get_variable_names(exoid, EX_NODAL, num_nod_vars, var_names);
428     for (i = 0; i < num_nod_vars; i++) {
429         free(var_names[i]);
430     }
431 }
432
433 /* read element variables parameters and names */
434 num_ele_vars = 0;
435 if (num_elem > 0) {
436     ex_get_variable_param(exoid, EX_ELEM_BLOCK, &num_ele_vars);
437

```



```

438     for (i = 0; i < num_ele_vars; i++) {
439         var_names[i] = (char *)calloc((MAX_STR_LENGTH + 1), sizeof(char));
440     }
441
442     ex_get_variable_names(exoid, EX_ELEM_BLOCK, num_ele_vars, var_names);
443     for (i = 0; i < num_ele_vars; i++) {
444         free(var_names[i]);
445     }
446
447     /* read element variable truth table */
448     if (num_ele_vars > 0) {
449         truth_tab = (int *)calloc((num_elem_blk * num_ele_vars), sizeof(int));
450
451         ex_get_truth_table(exoid, EX_ELEM_BLOCK, num_elem_blk, num_ele_vars, truth_tab);
452         free(truth_tab);
453     }
454 }
455
456 /* read nodeset variables parameters and names */
457 num_nset_vars = 0;
458 if (num_node_sets > 0) {
459     ex_get_variable_param(exoid, EX_NODE_SET, &num_nset_vars);
460
461     if (num_nset_vars > 0) {
462         for (i = 0; i < num_nset_vars; i++) {
463             var_names[i] = (char *)calloc((MAX_STR_LENGTH + 1), sizeof(char));
464         }
465
466         ex_get_variable_names(exoid, EX_NODE_SET, num_nset_vars, var_names);
467
468         for (i = 0; i < num_nset_vars; i++) {
469             free(var_names[i]);
470         }
471
472         /* read nodeset variable truth table */
473         if (num_nset_vars > 0) {
474             truth_tab = (int *)calloc((num_node_sets * num_nset_vars), sizeof(int));
475
476             ex_get_truth_table(exoid, EX_NODE_SET, num_node_sets, num_nset_vars, truth_tab);
477             free(truth_tab);
478         }
479     }
480 }
481
482 /* read sideset variables parameters and names */
483 num_sset_vars = 0;
484 if (num_side_sets > 0) {
485     ex_get_variable_param(exoid, EX_SIDE_SET, &num_sset_vars);
486
487     if (num_sset_vars > 0) {
488         for (i = 0; i < num_sset_vars; i++) {
489             var_names[i] = (char *)calloc((MAX_STR_LENGTH + 1), sizeof(char));
490         }
491
492         ex_get_variable_names(exoid, EX_SIDE_SET, num_sset_vars, var_names);

```

```

493     for (i = 0; i < num_sset_vars; i++) {
494         free(var_names[i]);
495     }
496
497     /* read sideset variable truth table */
498     if (num_sset_vars > 0) {
499         truth_tab = (int *)calloc((num_side_sets * num_sset_vars), sizeof(int));
500
501         ex_get_truth_table(exoid, EX_SIDE_SET, num_side_sets, num_sset_vars, truth_tab);
502         free(truth_tab);
503     }
504 }
505 }
506 }
507
508 /* determine how many time steps are stored */
509 num_time_steps = ex_inquire_int(exoid, EX_INQ_TIME);
510
511 /* read time value at one time step */
512 time_step = 3;
513 ex_get_time(exoid, time_step, &time_value);
514
515 /* read time values at all time steps */
516 time_values = (float *)calloc(num_time_steps, sizeof(float));
517
518 ex_get_all_times(exoid, time_values);
519
520 free(time_values);
521
522 /* read all global variables at one time step */
523 var_values = (float *)calloc(num_glo_vars, sizeof(float));
524
525 ex_get_var(exoid, time_step, EX_GLOBAL, 1, 0, num_glo_vars, var_values);
526 free(var_values);
527
528 /* read a single global variable through time */
529 var_index = 1;
530 beg_time = 1;
531 end_time = -1;
532
533 var_values = (float *)calloc(num_time_steps, sizeof(float));
534
535 ex_get_var_time(exoid, EX_GLOBAL, var_index, 1, beg_time, end_time, var_values);
536 free(var_values);
537
538 /* read a nodal variable at one time step */
539 if (num_nodes > 0) {
540     var_values = (float *)calloc(num_nodes, sizeof(float));
541
542     ex_get_var(exoid, time_step, EX_NODAL, var_index, 1, num_nodes, var_values);
543     free(var_values);
544
545     /* read a nodal variable through time */
546     var_values = (float *)calloc(num_time_steps, sizeof(float));
547

```

```

548     node_num = 1;
549     ex_get_var_time(exoid, EX_NODAL, var_index, node_num, beg_time, end_time, var_values);
550     free(var_values);
551 }
552
553 /* read an element variable at one time step */
554 if (num_elem_blk > 0) {
555     ids = (int *)calloc(num_elem_blk, sizeof(int));
556
557     ex_get_elem_blk_ids(exoid, ids);
558
559     for (i = 0; i < num_elem_blk; i++) {
560         if (num_elem_in_block[i] > 0) {
561             var_values = (float *)calloc(num_elem_in_block[i], sizeof(float));
562
563             ex_get_var(exoid, time_step, EX_ELEM_BLOCK, var_index, ids[i], num_elem_in_block[i],
564                       var_values);
565
566             free(var_values);
567         }
568     }
569     free(num_elem_in_block);
570     free(ids);
571 }
572 /* read an element variable through time */
573 if (num_ele_vars > 0) {
574     var_values = (float *)calloc(num_time_steps, sizeof(float));
575
576     var_index = 2;
577     elem_num = 2;
578     ex_get_var_time(exoid, EX_ELEM_BLOCK, var_index, elem_num, beg_time, end_time, var_values);
579     free(var_values);
580 }
581
582 /* read a sideset variable at one time step */
583 if (num_sset_vars > 0) {
584     for (i = 0; i < num_side_sets; i++) {
585         var_values = (float *)calloc(ss_specs.num_entries_per_set[i], sizeof(float));
586
587         ex_get_var(exoid, time_step, EX_SIDE_SET, var_index, ss_specs.sets_ids[i],
588                   ss_specs.num_entries_per_set[i], var_values);
589
590         free(var_values);
591     }
592 }
593
594 /* read a nodeset variable at one time step */
595 if (num_nset_vars > 0) {
596     for (i = 0; i < num_node_sets; i++) {
597         var_values = (float *)calloc(ns_specs.num_entries_per_set[i], sizeof(float));
598
599         ex_get_var(exoid, time_step, EX_NODE_SET, var_index, ns_specs.sets_ids[i],
600                   ns_specs.num_entries_per_set[i], var_values);
601
602         free(var_values);

```

```
603     }  
604 }  
605 ex_close(exoid);  
606 return 0;  
607 }
```