Planetary Data System
Object Access Library User's Guide

OAL Version 1.2

December 1997

Randy Davis and Steve Monk

Laboratory for Atmospheric and Space Physics
University of Colorado

Table of Contents

Part I — User's Guide

Part I — User's Guide

Chapter 1. OAL Overview

1.1 Contents of this Manual

This document is the User's Guide for the Object Access (OA) Library. The OA Library is a set of software routines for accessing data that meet the standards of the NASA Planetary Data System (PDS).

This document is divided into two parts:

- *User's Guide.* This is the User's Guide for scientists and other end-users who write applications using the OA library. Most users will find this information and the descriptions of object layer routines sufficient for their needs, and not need to go into the *Advanced User's Guide*.

- *Advanced User's Guide.* For readers who plan to write their own Object Access routines, this part contains a detailed description of the inner workings of the OA library, including all stream layer and structure layer routines.

1.2 Background

The Planetary Data System acquires, archives and distributes much of the data that NASA collects about objects in our solar system (other than the Earth) and about the interplanetary medium. Soon after the development of a prototype PDS began in 1982, it became clear that there were virtually no standards for the data produced by planetary missions and planetary scientists. This meant that the future PDS would have to accept, store, and deliver datasets in many different formats.

The developers of the prototype PDS responded to this challenge in several ways. First, a simple keyword/value language was devised that would permit a dataset or file to be labeled with information identifying its format and content. This language eventually evolved into the Object Description Language (ODL). Today, PDS labels written in ODL are attached to virtually every piece of data that flows into or out of the PDS. A PDS label in ODL can be read by both humans and computers. For example, a scientist can read a PDS label to learn about a file she receives, and at the same time the software she uses for data analysis can parse the label and use the information in the label to identify, locate and retrieve data from the file.

The planetary community's second response to the data format challenge was to develop a Planetary Science Data Dictionary (PSDD) and to formally define the standard types of data objects that are manipulated by planetary scientists (like images, spectra and data tables). The PSDD holds both the definitions of standard PDS data objects and the definitions of elementary terms used to describe planetary data (for example, the terms *latitude* and *longitude*). Each object definition specifies a set of elementary terms that represent the attributes of the object type. An attribute describes or qualifies an aspect of the format or content of the object. Some attributes are mandatory — a value must be assigned for every object instance — and some optional.

Application programs that process and analyze planetary data often require access to a combination of standard data objects, information from PDS labels, and information from the PSDD. Here are some examples:

- When a scientist provides a file, volume or dataset to the PDS, the data must have associated PDS labels. Information from the PSDD is used to ensure that the labels submitted with data are valid, consistent and complete.

- When the PDS produces a file, volume or dataset for scientists, information from the PSDD can be used to create the PDS labels that accompany the data. Values for an object's attributes (at a minimum the mandatory attributes) are filled in to describe each specific object instance. The attributes and their assigned values are then encoded in ODL to form the PDS label.

While PDS labels and the PSDD make it possible for software to identify, locate and retrieve data objects within a data stream or file, there has been an unfortunate lack of general-purpose software for accessing and manipulating PDS data. There are no standards or guidelines for software that manipulate PDS data objects, so routines for handling one type of PDS data object may work very differently from the software for other types of objects. Some PDS data objects — notably images and tables — have quite a bit of access and manipulation software available, while other types of objects have very little. The software that is available often works with only a subset of the possible objects that can be generated from an object type. For example, some of the routines and programs that handle PDS images ignore the attributes that specify whether or not the image data is band or line interleaved, which means that the software cannot handle multi-band images. Additionally, much of the current software can only handle data objects that are formatted for certain machines — for example, with numeric data that is represented in VAX integer and floating point formats — and with data objects encoded into files in certain ways (for example, data coded as fixed-length records but not variable length records).

The OA Library effort attempts to solve some of the problems listed above. This effort provides some standards and guidelines to follow when developing software for accessing PDS data objects. It provides routines for reading and writing PDS data objects from a variety of file formats and is able to translate data into the proper representations for a user's computer. It also provides routines for accessing the most important types of PDS data objects. The OA Library provides the basic capabilities that will make it easier for scientists to develop their own software for accessing PDS data.

1.3 Software Organization

The OA Library consists of three layers of software. The first layer, called the *stream layer*, provides functions needed to read in and write out a PDS data object — or part of an object — from or to a file. (The term *data object* is fully defined and discussed in Section 1.3.1 below). The second layer, called the *structure layer*, deals with data representation issues (for example, the need to convert numeric data from one format to another when moving across computer platforms). The third layer, called the *object layer*, provides routines for end users (predominately planetary scientists) of PDS data objects. These three layers are discussed in greater detail in Section 1.3.2 below.

The OA Library serves as a *starter set* of software for reading, manipulating and writing PDS data objects. We expect that users of the OA Library will augment the OA Library by developing their own object access (OA) routines. These user-supplied routines will augment the OA Library's object layer and will provide new operations on PDS data objects. At the same time, we want to encourage uniformity in OA software so that the OA Library and user-supplied routines will be of greatest use to the entire planetary science community. Therefore this document provides a general set of guidelines and requirements for software that accesses PDS data objects. These are given in Part III. While it will not be possible to force users to follow these guidelines, we believe that by following the guidelines users will be able to develop software for manipulating PDS data objects faster and better (because they will have the OA Library and other existing OA routines to build upon) and they will be able to make their software useful to a wider audience.

1.3.1 What is a PDS Data Object?

A *data object* is simply a chunk of data of a recognizable type. This may be something as simple as a single integer number or as complex as a multi-spectral image. A *PDS data object* is a data object that is of a type defined in the Planetary Science Data Dictionary and for which an appropriate PDS label exists (or can be created) to provide the values of the attributes of the data object.

Our use of the word *object* deserve some discussion. PDS data standards are *object based* but not *object oriented*. By object-based, we mean that the PDS has defined a set of data types suitable for describing a wide variety of the data objects that scientists know and manipulate routinely — like images, tables, spectra, and so on. For each of these types of data, a set of attributes is defined in the PSDD that describe the data in a way that both a scientist and a computer program can understand. To this degree, PDS data objects are similar to the types of data defined in an object-oriented data system. But from an object-oriented perspective, PDS standards lack a few essential ingredients:

- Inheritance — a mechanism for explicitly relating one type of object to all other types of objects;

- Methods — a mechanism for associating software with specific types of data objects.

In developing the OA Library, it is not our intention to attempt to move the PDS standards from an object-based to a fully object-oriented implementation. We do intend to develop the OA Library in such a way that it can be used in traditional (non-object oriented) computing environments and — with some extensions as noted above — in object-oriented environments as well.

A PDS data object consists of two parts:

- *Object attributes* — A PDS label (or portion of a label) that contains an object description with the values of the attributes for the data object. This information can be in an external format (that is, in ODL within a file) or in an internal format (that is, in some memory-resident representation that provides software with access to the label information). The standard memory-resident representation of a PDS label is an *ODL Tree*. An ODL tree is a linked data structure that contains the same information as the corresponding PDS label, but in a format that is easier for computers to access and manipulate. The PDS has produced a library of routines to read a PDS label from a file and create the corresponding ODL tree, to manipulate the tree, and to convert an ODL tree into a PDS label and write that label to a file. This library is called L3 (Label Library Lite), and is included in the OA Library distribution.

- *Object data* — A sequence of bytes that form the primary data value of the object. The format of the data are given by the associated PDS label. This part of a PDS data object may be null; that is, it is possible for a PDS data object to consist of nothing but the object attribute values defined within a PDS label. However, the opposite is not true: a piece of data without a PDS label is *not* considered a PDS data object. (It may, however, be turned into a PDS data object by generating the appropriate PDS label information for the data).

1.3.2 OA Library Architecture

The OA Library consists of three layers of software:

- Stream Layer — This layer provides software for dealing with data as a stream of contiguous bytes within a file. The stream layer is able to extract a data stream from fixed or variable length records and from non-record oriented files, and do so on a variety of host computers. The stream layer is stand-alone so it can be used independently of the rest of the OA library, if desired.

- Structure Layer — This layer provides software for dealing with data as a sequence of atomic data types (for example, integer numbers or ASCII characters). The atomic data types for the PDS are described in the PDS Standards Reference Document (Chapter 3 and Appendix C), and in the PSDD. As an example, consider a simple binary data table where each row of the table consists of a series of fields, each of which is a single atomic value. The structure layer provides software for dealing with the fact that the representations of these atomic data types differ across computer platforms. Referring back to our example, the structure layer is able to convert the table fields from VAX format to Sun format, etc. The possible representations of the base data types are defined by the values of the element DATA_TYPE, in the PDS Standards Reference. Another issue that is dealt with at the structure layer is alignment of data. Again looking at our table example, some computers allow data fields to be byte aligned, but some machines may require certain fields to be aligned on word or even multi-word boundaries. The structure layer provides the mechanisms for assuring that data are aligned properly within a computer's memory.

- Object Layer — The object layer provides object access (OA) routines for reading, manipulating and writing PDS data objects. This is the layer scientists and other end-users use in their applications. The routines in the object layer carry out well-defined operations on PDS data objects. In this regard, OA routines are similar to the *methods* defined for an object in an object-oriented data system. A tutorial on implementing new OA routines is given in part II of this manual, the Advanced User's Guide.

- L3 — The Label Library Lite (L3) is utilized by all layers of the OA Library. L3 provides routines for parsing PDS labels and accessing label information in memory. L3 routines are also used to write label information from an ODL tree in memory to a label file. User interfaces utilize L3 to navigate in ODL trees, and to find nodes to pass to OA routines. The OA routine *OaParseLabelFile* should be used to read a PDS label into memory; it utilizes L3 and the OA stream layer, and works for all file record formats.

1.4 Glossary

This section provides a glossary for key terms used in this document.

Attribute — A data element — taken from the Planetary Science Data Dictionary — that describes some aspect of a PDS data object's form or content. Examples are the attributes LINES and LINE_SAMPLES, which respectively specify the number of lines and the numbers of pixels per line of an image. In the context of ODL trees, **keyword** often refers to the attribute's name and value, and **keyword value** refers only to its value.

Data Object — A chunk of data of a recognizable type. This may be something as simple as a single integer number or as complex as a multi-spectral image.

Object Access (OA) Routine — A routine that performs a single well-defined operation upon a PDS data object. These are the routines that are used by scientists and other end-users to operate upon PDS data. The OA Library provides a starter set of OA routines, but developers can build upon the OA Library's stream and structure layers to create their own OA routines. The *Advanced User's Guide*, Chapter 9 of this document provides information on implementing new OA routines.

Object Data — The primary data that is associated with a PDS data object. For example, for an image the object data is the set of lines and samples that form the image. Not all types of PDS objects have object data: some consist only of a set of attributes, such as the HISTORY or CATALOG objects.

Object Descriptor — A data structure that is passed to and from OA routines that represents a specific PDS data object. It contains a pointer to the object data, if any, and another to the ODL tree with the attributes for the object.

Object Description Language (ODL) — The language in which PDS labels are written. The ODL syntax and semantics are specified in Chapter 12 of the PDS Standards Reference.

Object Layer — The collection of Object Access (OA) routines that are provided as part of the OA Library.

ODL Tree — A data structure that contains the information from a PDS label, but in a form that computer programs can readily manipulate. ODL trees are created when PDS labels are read into memory using the routine *OaParseLabelFile*, which uses stream layer and the PDS Label Lite Library. Every PDS data object that has been read into memory using an OA routine will have an ODL tree that provides the values for all of the attributes of the object. When an OA routine writes out a PDS data object to a file, the ODL tree for the object is appended to the ODL tree for the file, which is in turn converted to a PDS label and written out to describe the file contents.

PDS Data Object — A data object that is of a type defined in the Planetary Science Data Dictionary and for which an appropriate PDS label exists (or can be created) to provide the values of the attributes of the data object.

PDS Label — A description of all of the PDS data objects within a file. PDS labels are written in ODL. The conventions for encoding PDS labels are given in Section 5 of the PDS Standard Reference.

Planetary Science Data Dictionary (PSDD) — The data dictionary that defines all PDS data objects and all of the elements that can be used as attributes for a PDS data object. It is defined in the Planetary Science Data Dictionary document.

Stream Decomposition Tree (SDT) — An augmented ODL tree that contains information that is used by structure layer routines to transfer and translate data.

Stream Descriptor — A data structure used by stream layer routines to keep track of access to a file.

Stream Layer — The set of routines provided as part of the OA Library for reading and writing data from or to files. It is used to read in or write out the object data associated with a PDS data object, and it can handle all the file and record formats described in the PDS Standards Reference.

Structure Layer — The set of routines provided as part of the OA Library for translating data from the format used by one supported platform (like a VAX) to another supported platform (like a Sun workstation).

1.5 Reference Documents

R. Davis and S. Monk; Object Access User's Guide; OAL Version 1.1; December 1996.

R. Davis and S. Monk; Object Access User's Guide; OAL Version 1.0; 14 February 1996.

R. Davis and S. Monk; Object Access Library Concept, Requirements and Design document; Beta Release Version; 15 March 1995.

R. Davis and S. Monk; Object Access Library Concept, Requirements and Design document; Pre-Alpha Release Version; 10 August 1994.

R. Davis and S. Monk; Object Access Library Concept and Requirements Document; Draft, 31 January 1994.

S. Hughes and D. Bernath; Label Library Light (L3) User's Guide; January 15, 1995. Supplied in the OA Library distribution.

PDS; Planetary Data System Standards Reference; Version 3.1, 3 Aug 1994. Available at: http://pds.jpl.nasa.gov/stdref/stdref.htm

PDS; Planetary Science Data Dictionary; Revision C, 20 Nov 1992.

ISO; ISO 9660 Standard on Volume and File Structure for CD-ROM for Information Interchange; First edition, 15 April 1988.

Chapter 2. Using OAL

This chapter provides information on compiling the OA library, the include files you need, and on customizing certain OA routines to fit your user interface and memory management scheme.

2.1 The Source Files You Need

The OA library distribution includes all the OA and Label Library Lite (L3) source code, include files and a Makefile for UNIX platforms. When developing an application in C, the only include file (*.h* file) you need to include in your code is *oal.h*; this has all the typedefs, structure definitions, enumerated types and function prototypes you need to develop your application.

2.2 Compiling OAL

The OA library code requires the user to explicitly define a platform (machine) specific symbol (macro) at compile time - the code does **not** rely on symbols predefined by certain compilers. This should be done via a compiler switch or prefix file, rather than adding it to all the source code files. The symbol is used in the stream layer and L3 to compile different code for dealing with platform-specific issues such as file path formats and record formats, and in the structure layer to select a profile matching the platform's native binary data types. For example, defining VAX or ALPHA_VMS compiles VMS-specific I/O code in the stream layer, and selects the data types profile containing VMS numeric data types. Similarily, defining ULTRIX, ALPHA_OSF or IBM_PC selects the profile containing the LSB integers and byte-reversed IEEE floats native to these platforms. The data types profile is a data structure which tells the structure layer which native binary data types to convert foreign binary data types or ASCII data to.

The define values for the supported platforms and compilers are listed below, along with compiler flags which must be set .

1) VAX/VMS
    VAX C compiler; /define=VAX

2) Sun Sparc/SunOS and Solaris
    cc, /usr/5bin/cc; -DSUN4 -D_NO_PROTO
    gcc, CC (Sparc Works C++ compiler); -DSUN4

3) Silicon Graphics/Irix
    gcc and cc; -DSGI

4) Dec 3100/Ultrix
    cc; -DULTRIX -Dstd1

5) Dec Alpha/OSF-1
    cc; -DALPHA_OSF

6) Dec Alpha/OpenVMS
    OpenVMS C compiler; /define=ALPHA_VMS

7) Macintosh II
    Symantec ThinkC Version 6.0
    Project/Set Project Type/Far DATA, Far CODE
    Edit/Options/ThinkC/Compiler Settings/ 4-byte ints, 8-byte doubles
    Edit/Options/ThinkC/Prefix:
        #include <MacHeaders>
        #define MAC
    Add ANSI and unix libraries to the project, and recompile with same switches.

8) Macintosh PowerPC
    Metrowerks CodeWarrier
    Include the standard libraries for a PowerPC application:
    MWC_RuntimeLib
    Mathlib
    InterfaceLib
    ANSI C.PPC.Lib
    SIOUX.PPC.Lib (for console I/O used by OAL test procedures)
    Edit a file, e.g. "prefix.h", and add these lines:
        #include <MacHeaders>
        #define MAC
    then add "prefix.h" to the project with Edit/Preferences/C Language/Prefix File.

9) IBM-PC/Dos
    Borland C++ Version 4.5
    command line compiler: bcc -DIBM_PC=1 -DMS_DOS=1 -mh -f
    command line linker: tlink /Tde C0H.OBJ MATHH.LIB EMU.LIB CH.LIB
  For Windows 32-bit applications, define only IBM_PC.

2.3 Customizing the Error Reporting Routine

The OA Library does all its error reporting by setting the global variable *oa_errno*, then passing to the function*OaReportError* a string containing an error message. As supplied in the release, *OaReportError* prints the string to the standard error output. You can replace the fprintf statement in *OaReportError* (located in *rprt_err.c*) by code your user interface uses to alert the user of an error. You can also filter the messages which are presented to the user by selecting only those *oa_errno*'s you wish the user to see. For example, most users (except label verifiers) don't care about the warning message "Implicit SPARE detected" with *oa_errno* = 900, so you could modify *OaReportError* to not print the error string when *oa_errno* has this value. You can also screen messages by category, for example, all informational messages, which have *oa_errno*'s in the range 950 - 999. The OA library's error reporting mechanism is described in detail in “Error Reporting Mechanism” on page 40, and a list of *oa_errno* error code values is in Appendix A.

2.4 Customizing the Memory Management Routines

The OA Library and L3 do all dynamic memory allocation using the routines *OaMalloc*, *OaRealloc* and *OaFree* (located in *oamalloc.c*). As supplied in the release, *OaMalloc* calls *malloc()* (or *farmalloc()* on an IBM-PC running DOS), *OaRealloc* uses *realloc()* and *OaFree* uses *free().* If needed, you can modify the code inside of *OaMalloc*, *OaRealloc* and *OaFree* to call your own memory allocation routines.

2.5 Optional Libraries and Utilities

2.5.1 Unzip

In the future, PDS archives may contain compressed data files using Zip compression. On most platforms, OAL will automatically unzip such files, provided that the Unzip executable is present and accessible (in the path of the environment your OAL application runs in). Alternatively, the user can do the unzip manually before calling an OA routine which accesses the file. Either way, the user should install Unzip, as described in the Unzip documentation in order to access zipped files. Unzip (and Zip) documentation and executables for all platforms supported by OAL are available from ftp.cdrom.com in pub/infozip, and from various mirror sites. InfoZip’s home page is http://www.cdrom.com/pub/infozip/

Chapter 3. Object Layer

3.1 Overview

Object Layer routines (from here on referred to as Object Access or OA routines) can be broken down into three broad categories: (1) routines that read PDS data objects from a file; (2) routines that manipulate PDS data objects in memory; and (3) routines that write PDS data objects.

3.1.1 Reading PDS Data Objects from a File

- **Read the Label into an ODL Tree:**
  OA routines that read objects from files take as their principal input an ODL tree describing the contents of the entire file. This means that the PDS label for the file must first have been read into memory using *OaParseLabelFile*, and converted to the latest PDS standards by *OaConvertLabel*, prior to calling the OA read routine.

- **Locate the Desired ODL Tree Node:**
  The actual input to the OA read routine is a pointer to the specific node in the file's ODL tree that describes the object to be read. L3 routines may be used to navigate in the ODL tree to find the desired node. If the node that is pointed to does not have the object class expected by the OA routine, then it issues an error message and returns a null pointer.

- **Read in the Object:**
  The OA routine *OaReadObject* can be used to read any object into memory. Some objects, particularly Images and Tables, are occasionally so large that it is not practical to have the entire object in memory at one time. The OA Library provides OA routines for reading a selected portion of an Image, Table or Qube.

  Some types of objects — like Catalog, Directory and Volume objects — consist only of the attributes found in the ODL tree. OA routines for reading these kinds of objects can call *OaParseLabelFile*, then simply copy out the pertinent part of the file's ODL tree.

  Most OA routines for reading from files rely upon the OA Library's stream layer to fetch the object's data and on the structure layer to translate the object data into the proper binary format for the host computer. This means that the details of file format and data format within a file are usually transparent to the OA routine, having been handled before the data even gets to the OA routine.

3.1.2 Manipulating PDS Objects in Memory

- **Extracting Subobjects:**
  Routines are provided to extract and combine parts of objects. For Tables, routines are provided to add and delete rows or columns, and to deal with nested objects like Containers. For example, the OA routine*OaGetSubTable* can be used to extract a column of a Table; it returns a new Table object with a single column. Each routine returns a new object, consisting of object data and an ODL tree with the attributes of the new object.

- **Converting Object Data:**
  Routines are provided convert an object's data to ASCII or to different binary types. The OA routine *OaConvertObject* can be used to convert an object's data to ASCII or to any other combination of profile settings. The OA routine *OaConvertObjecttoOneType* converts all the data atoms in an object to the specified type, so that it can be accessed as a C array. The object's ODL tree is always updated with the new attributes of the converted data.

- **Exporting Object Data:**
  The OA routine *OaExportObject* strips an object of it's ODL tree and object descriptor, and returns a pointer to the object data.

3.1.3 Writing PDS Objects to a File

Users can call routines provided in the OA Library to write most types of PDS data objects from memory to a file. The OA routine *OaOpenOutputFile* opens an output file with the specified attributes. Subsequent calls to the routine *OaWriteObject* write an object's data to the file. As each object's data is written to the file, the object's ODL tree is copied and appended to the output file's ODL tree, and a pointer keyword for the object is added to the root node of the output file's ODL tree. After all the objects have been written, a call to *OaCloseOutputFile* closes the data file and writes out the ODL tree as a detached PDS label. If desired, the data file and label file can then be combined into an attached label file by a call to *OaCreateAttachedLabel*.

3.2 Object Layer Data Structures

Most object layer routines return a pointer value of type *OA_OBJECT*. This is a pointer to a data structure called an *object descriptor* that contains pointers to an object's data and to the ODL tree specifying the object's attributes. It also provides for a pointer to a stream descriptor, which is used when reading in or writing out PDS data objects to or from file. Users should not change any of these fields.

```
typedef struct OA_Object {
  ODLTREE               odltree;
  PTR                   data_ptr;
  long                  size;
  struct OaStreamStruct *stream_id;
  int                   is_in_memory;
  void                  *appl1;
  struct oa_profile     profile;
} *OA_OBJECT;
```

odltree	A pointer to an ODL tree which contains the attributes for a data object.
data_ptr	Pointer to the object's data. This pointer has the value NULL if there is no data for the object (i.e., the object consists only of attributes that are provided through the object's ODL tree) or if the object's data is not in memory. PTR is a typedef equivalent to char * on most platforms, and char huge * on the IBM PC.
size	Size of the object's data, in bytes. This is set to zero if the object data is not in memory or if there is no object data.
stream_id	Pointer to a stream descriptor for the data stream through which the object's data is accessed. If the object has no data associated with it, or if the data is in memory and not in a file, then this value is NULL.
is_in_memory	Flag indicating whether the object's data is in memory or in a file.
appl1	Used by *OaOpenImage* and *OaReadImagePixels* to point to an oa_image_handle structure.
profile	The profile which was used when the object data was created or last modified, used internally by OAL.

The root node of an object descriptor's ODL tree is always a "top-level" object class, such as a Table, Image, Array or History object; it is never a stand-alone Column, Bit Column or Bit Element node: these are only found as sub-objects under a Table or Array. For example, an object descriptor with a single Column worth of data is thought of as a Table with one Column, not as a stand-alone Column, and its ODL tree consists of a Table node with a single Column node below it.

3.3 OA Routines for Reading and Converting Label Files

OaParseLabelFile utilizes the OA stream layer and L3 to read a label from a file into memory. Since it can handle variable-length record files, it's use is preferable to the L3 routine *OdlParseLabelFile*, which cannot. Once in memory, the label (ODL tree) should be converted to the latest version of ODL by calling *OaConvertLabel*. The OA library depends on all ODL trees following the PDS Version 3 standards.

3.3.1 OaParseLabelFile

This routine is like L3's *OdlParseLabelFile*, except that it uses the stream layer to read the file, which makes it possible to read labels from variable-length record files.

```
ODLTREE OaParseLabelFile (char           *filespec,
                          char           *errfilespec,
                          MASK            expand,
                          unsigned short nomsgs)
```

filespec	in	A character string representing a file name or path.
errfilespec	in	A character string representing a file name or path to write error messages to during parsing.
expand	in	A bit mask which specifies what to expand, if anything. Options are ODL_EXPAND_STRUCTURES and/or ODL_EXPAND_CATALOG.
nomsgs	in	TRUE means parser error messages should be written to errfilespec.

In order to facilitate expansion of ^STRUCTURE pointers in labels, provide a full path name in *filespec* if possible. The statements,

```
ODLTREE odltree;
odltree = OaParseLabelFile( "/mydisk/labels/X.LBL", "PARSER_ERRORS.TXT",
                            ODL_EXPAND_STRUCTURES, TRUE);
```

reads label file X.LBL into an ODL tree in memory, expands any ^STRUCTURE pointers, and if there were any parser error messages, appends them to the file PARSER_ERRORS.TXT.

3.3.2 OaConvertLabel

This routine converts an ODL tree to comply with the latest version of PDS standards, currently Version 3. Users must call this routine after reading in a label with *OaParseLabelFile*, as structure layer routines depend on it having been called.

```
ODLTREE OaConvertLabel (ODLTREE root_node)
```

root_node	in	Pointer to the root node of an ODL tree read from a file.

The ODL tree may be modified by this routine. A pointer to the modified ODL tree's root node is returned as the function value. Upon successful conversion, the routine inserts the PDS_VERSION_ID keyword into the root node and sets it's value to the latest version of PDS, i.e. PDS3.

3.4 OA Routines for All Objects

The following OA routines can handle more than one type (class) of object. With the exception of *OaReadObject*, these routines implement functions that are performed in the same way for all supported object types. For example, the operations of copying or deleting objects don't require any object type-specific information.

3.4.1 OaReadObject

This OA routine provides the simplest way to read different types of objects. To do so it determines the type of object by inspecting the ODL tree node pointed to by the input parameter, and then calls the appropriate type-specific routine for reading the object from file. (The type-specific read routines which exist for every object type are not all described in this document.)

```
OA_OBJECT OaReadObject (ODLTREE object_node)
```

object_node	in	A pointer to the node of an ODL tree that describes the object to be read. The object may be any of the supported types listed below.

The routine reads the object indicated by the input parameter from a file into memory, by default converting the object into the proper binary format for the current platform. It returns a pointer to an object descriptor that contains a pointer to the object data and a pointer to the ODL tree with the object's attribute information. The input ODL tree is unchanged.

This routine currently handles the following types of PDS data objects:

- Array
- Collection
- Histogram
- History (this object is all ODL tree, no object data)
- Image (if multi-band, only the first band is read)
- Table, Series, Spectrum, Palette, Gazetteer

As OA routines to read more types of objects become available, the routine will be expanded to include them. The code fragment,

```
ODLTREE odltree, image_node;
OA_OBJECT image_object;

odltree = OaParseLabelFile( "/mydisk/labels/X.LBL", "PARSER_ERRORS.TXT",
```

```
                            ODL_EXPAND_STRUCTURES, TRUE);
image_node = OdlFindObjDesc( odltree, "*IMAGE", "*", 0L,
                             ODL_RECURSIVE_DOWN);
image_object = OaReadObject( image_node);
```

reads in a label, finds the first image object, and reads the image into memory.

3.4.2 OaCopyObject

This OA routine makes a copy of all the attributes and data associated with an object.

```
OA_OBJECT OaCopyObject (OA_OBJECT object)
```

object	in	A pointer to an object descriptor for the PDS data object to be copied.

The routine returns a pointer to the object descriptor of the copied object.

This routine handles all objects, including those which have only attributes and no object data; in this case the pointer to the object data in the output object descriptor is set to NULL (the same as in the input object).

3.4.3 OaDeleteObject

This routine deletes an object, including the object's in-memory data, if any, the ODL tree associated with the object, and finally the object descriptor.

```
int OaDeleteObject (OA_OBJECT object)
```

object	in/out	A pointer to the object descriptor for the object to be deleted.

The routine always returns a status value of zero as its function value. After this call is made, the input OA_OBJECT pointer should no longer be accessed.

3.4.4 OaExportObject

This OA routine strips off the object descriptor and ODL tree associated with the input object and returns a pointer to the object's data.

```
PTR OaExportObject (OA_OBJECT object)
```

object	in	A pointer to the object descriptor for the object to be exported.

The routine returns a pointer to the object data. PTR is a typedef equivalent to char * on most platforms and char huge * on the IBM PC. The return value is NULL if an error occurs, or if an attempt is made to export an object for which there is no data (for example, a Catalog type object for which there are only attributes encoded in the object's ODL tree). The code fragment,

```
ODLTREE image_node;
OA_OBJECT image_object1, image_object2;
long lines, pixels;
unsigned short *image;

image_object1 = OaReadObject( image_node);
OaKwdValuetoLong( "LINES", image_node, &lines);
OaKwdValuetoLong( "LINE_SAMPLES", image_node, &pixels);
image_object2 = OaConvertObjecttoOneType( image_object1,
                                          "unsigned short",0,0);
OaDeleteObject( image_object1);
image = (unsigned short *) OaExportObject( image_object2);
```

reads in an image, gets the number of lines and pixels, and converts the object's data to unsigned shorts. The return value of *OaExportObject* is then casted to an unsigned short, and can now be accessed through the *image* variable; *lines* and *pixels* give the size of the data.

3.4.5 OaConvertObject

This OA routine copies an object and converts the object's data to a different interchange format, alignment type or binary data types, using the current Oa_profile settings. This routine is intended for converting an object from binary to ASCII or ASCII to binary, or for converting data to a different platform's data types. Another OA routine, *OaConvertObjecttoOneType*, can be used to convert uniform data to a single type, in order to access it through a C array or pointer. The input object data may consist of various different data types (as in a Table with multiple columns), and these will be converted, as specified in the data types profile, to other data types. The caller should set Oa_profile to specify the desired target platform or ASCII/binary prior to calling this function, and restore it to its original values after the call.

```
OA_OBJECT OaConvertObject (OA_OBJECT object)
```

object	in	A pointer to the object descriptor for the object to be converted.

The routine returns a pointer to the new OA_Object structure, which contains an ODL tree describing the converted data, and a pointer to the converted data. If there was an error, a NULL pointer is returned. The input object is unchanged.

The following types of PDS data objects are supported by this routine:

- Array
- Collection
- Histogram
- Image
- Table, Series, Spectrum, Palette, Gazetteer

The code fragment,

```
OA_OBJECT binary_table, ASCII_table;
int saved_interchange_format;

saved_interchange_format = Oa_profile.dst_format_for_binary_src;
Oa_profile.dst_format_for_binary_src = OA_ASCII_INTERCHANGE_FORMAT
ASCII_table = OaConvertObject( binary_table);
OaDeleteObject( binary_table);
Oa_profile.dst_format_for_binary_src = saved_interchange_format;
```

converts every column in a table to ASCII values, deletes the old binary table, and restores global variable Oa_profile to its original values.

3.4.6 OaConvertObjecttoOneType

This OA routine converts all the object data of an in-memory OA_OBJECT from one numeric data type to another numeric type. The input object data must contain values of only one data type, e.g. an Image or a Table with a single column.

```
OA_OBJECT OaConvertObjecttoOneType (OA_OBJECT  object,
                                     char      *data_type
                                     int        bytes,
                                     int        rescale)
```

object	in	A pointer to the object descriptor for the object to be converted.
data_type	in	Pointer to a string containing the name of the C type or PDS data type into which the object data is to be translated. If a PDS data type, this can be any of the standard values allowed for the DATA_TYPE keyword for numeric types. If a C type, allowed values are: "char", "unsigned char", "short", "unsigned short", "int", "unsigned int", "long", "unsigned long", "float", "double".
bytes	in	The number of bytes in the specified data_type. Ignored if data_type is a C type.
rescale	in	A value of TRUE (non-zero) means the SCALING_FACTOR and OFFSET keyword values will be applied to the data, if these keywords are present in the oa_object's ODL tree. The computation done is: new_data = (SCALING_FACTOR * data) + OFFSET

The routine returns a pointer to the new OA_Object structure, which contains an ODL tree describing the converted data, and a pointer to the converted data. If there was a fatal error, a NULL pointer is returned. The input object is unchanged. An error message is issued with *oa_errno*=904 if there were any integer truncations, attempts to convert a negative integer to an unsigned integer, or loss-of-precision errors in floating point numbers. This code fragment reads the first column of a Table and converts it to be accessed as a C double array.

```
OA_OBJECT column_obj1, column_obj2;
ODLTREE table_node, column_node;
long rows;
double *dbl_arr;

column_node = LeftmostChild( table_node);
OaKwdValuetoLong( "ROWS", table_node, &rows);
column_obj1 = OaReadSubTable( table_node, 1, rows, &column_node, 1);
column_obj2 = OaConvertObjecttoOneType( column_obj1, "double",0,0);
OaDeleteObject( column_obj1);
dbl_arr = (double *) OaExportObject( column_obj2);
```

3.5 OA Routines for Writing Objects to a File

The OA routines described in this section can be used to create files that contain one or more PDS data objects, write the associated label file, and create attached label files. The first step is to call *OaOpenOutputFile*, which creates the data file with the specified record attributes, and returns an object descriptor that is used by subsequent routines to identify the file. The second step is to write a PDS data object to the file using the routine *OaWriteObject*. This routine writes the object data to the file, and updates the file object's ODL tree. More objects can be added to the file by additional calls to *OaWriteObject.* Single objects too big to store in memory at one time can be written by multiple calls to *OaWriteObject*. When the file is completed, a call to *OaCloseOutputFile* closes the data file and writes out the file object's ODL tree to a detached PDS. If desired, the label file and data file can be combined with a call to *OaCreateAttachedLabel*.

3.5.1 OaOpenOutputFile

This routine opens an output file with the specified file characteristics.

```
OA_OBJECT OaOpenOutputFile (char *data_filename,
                            int   record_type,
                            long  record_bytes)
```

data_filename	in	Pointer to a string containing the name for the output file.
record_type	in	Indicator for the type of records in the output file. The allowed values are defined by the enumeration type *oa_record_type_enum* .
record_bytes	in	The record length for the output file. If record_type = OA_FIXED_LENGTH, then record_bytes must be an even number.

This routine creates an object descriptor, and attaches an ODL tree containing file attributes to describe the file. The ODL tree initially consists of a single node containing file keywords. The routine also attaches a stream structure to the object descriptor to keep track of the open file.

The routine returns a pointer to the file object's descriptor. The statements,

```
OA_OBJECT file_object;

file_object = OaOpenOutputFile( "MIRANDA.IMG", OA_FIXED_LENGTH, 200);
```

creates the file *MIRANDA.IMG* as a 200-byte fixed-length record file, and returns the file object descriptor in the *file_object* variable.

3.5.2 OaWriteObject

This function writes the input object's data to the data file. The file object's descriptor must have been previously created by the routine *OaOpenOutputFile.*

```
int OaWriteObject (OA_OBJECT file_object,
                   OA_OBJECT object)
```

file_object	in/out	An object descriptor for the file to which the object is to be written.
object	in	A pointer to an object descriptor for the object to be written.

This routine updates the ODL tree describing the contents of the output file by attaching a copy of the input object's ODL tree to the file_object's tree and adding a pointer to the object's location within the output file. The routine returns a status value of 0 if successful. The input object is unchanged. For fixed-length record files, if the object data size is not a multiple of RECORD_BYTES, then pads will be inserted after the object data to make the last record be RECORD_BYTES in length. The statements,

```
int result;
OA_OBJECT file_object, image;

result = OaWriteObject( file_object, image);
```

writes the object data from the *image* object to the file, and adds the image object's ODL tree below the root of *file_object*'s ODL tree.

This routine can also be used to write an object too big to store in memory at one time. To do this, the first call to *OaWriteObject* should provide the ODL tree for the full-size object, and the first chunk of object data. Subsequent calls should provide object data (whose size in bytes is given by the OA_OBJECT's *size* field), with the input object's ODL tree set to NULL. Care should be taken when writing to a fixed-length record file that the data size of each chunk of object data is a multiple of RECORD_BYTES, otherwise pad data will be written at the end of each chunk (see note above on fixed-length record files). The following code fragment transfers the first 10 bands of a Qube from an input file to an output file, setting up the output file's ODL tree to describe a multi-band Image.

```
OA_OBJECT file_object, image_object;
ODLTREE odltree, qube_node;
int i;

odltree = OaParseLabelFile( "QUBE.LBL", "ODL_ERRORS.TXT", 0, 0);
qube_node = OdlFindObjDesc( odltree, "*QUBE", NULL, NULL, 0L, 0);
file_object = OaOpenOutputFile( "MULTIBAND.IMG",
                                OA_UNDEFINED_RECORD_TYPE, 0);
image_object = OaReadImageFromQube( qube_node, 1);
OaStrtoKwdValue( "BAND_STORAGE_TYPE", image_object->odltree,
                 "BAND_SEQUENTIAL");
```

```
OaStrtoKwdValue( "BANDS", image_object->odltree, "10");
OaWriteObject( file_object, image_object);
OaDeleteObject( image_object);
for (i=2; i<=10; i++)
{
  image_object = OaReadImageFromQube( qube_node, i);
  OdlFreeTree( image_object->odltree);
  image_object->odltree = NULL;
  OaWriteObject( file_object, image_object);
  OaDeleteObject( image_object);
}
OaCloseOutputFile( file_object, "MULTIBAND.LBL");
```

3.5.3 OaCloseOutputFile

This OA routine closes an output file and writes a detached PDS label describing the file contents. The file should be created using *OaOpenOutputFile* and written to using *OaWriteObject*.

```
int OaCloseOutputFile (OA_OBJECT file_object,
                       char      *label_filename)
```

file_object	in/out	Pointer to the object descriptor for the output file.
label_filename	in	A character string containing the name of a new file into which the detached PDS label describing the output file will be written.

The statement,

```
OA_OBJECT file_object;

OaCloseOutputFile( file_object, "MULTIBAND.LBL");
```

closes the file and writes *file_object*'s ODL tree to the file*MULTIBAND.LBL*.

3.5.4 OaCreateAttachedLabel

This OA routine takes a detached label file and the data file referenced in it, and combines them into one file, an attached label file. The attached label file contains the label in ASCII, followed by the data, usually binary. Object ^POINTER keywords in the label portion giving object start offsets are updated accordingly.

```
int OaCreateAttachedLabel (ODLTREE odltree,
                           char *label_filespec,
                           char *attached_lbl_filespec)
```

odltree	in	An ODL tree containing the label which describes the data file. Ignored if NULL.

label_filespec	in	A character string giving the path or file name of the label file. Ignored if NULL. Either *odltree* or *label_filespec*, but not both, must be specified.
attached_lbl_filespec	in	A character string giving the path or file name of the attached label file to be created. Must be different from *label_filespec*.

The routine returns 0 if successful, otherwise 1 and an error message. The input label file or *odltree*, and the data file are unchanged. The attached label file has the same record type and record length as the data file. The label file or *odltree* must describe only one data file, and the routine expects every object ^POINTER keyword value in the root node to reference the data file's name, i.e. no FILE objects allowed.

3.6 OA Routines for Image Objects

The PDS Image object type covers a wide variety of image formats. For example, images may be compressed or uncompressed; they may possess prefixes and/or suffixes; and they may contain a single or multiple image planes or bands. This leads to a problem: when object routines are created to manipulate image objects, how does a user know: (1) which variations on images the routine can handle and which it cannot handle; and (2) what variant of image is required for the inputs and returned as the output of the routines?

To resolve this issue, we have broken the PDS Image object type into a number of subtypes as shown in Figure 1. Table 1 below provides a definition for each of these subtypes.

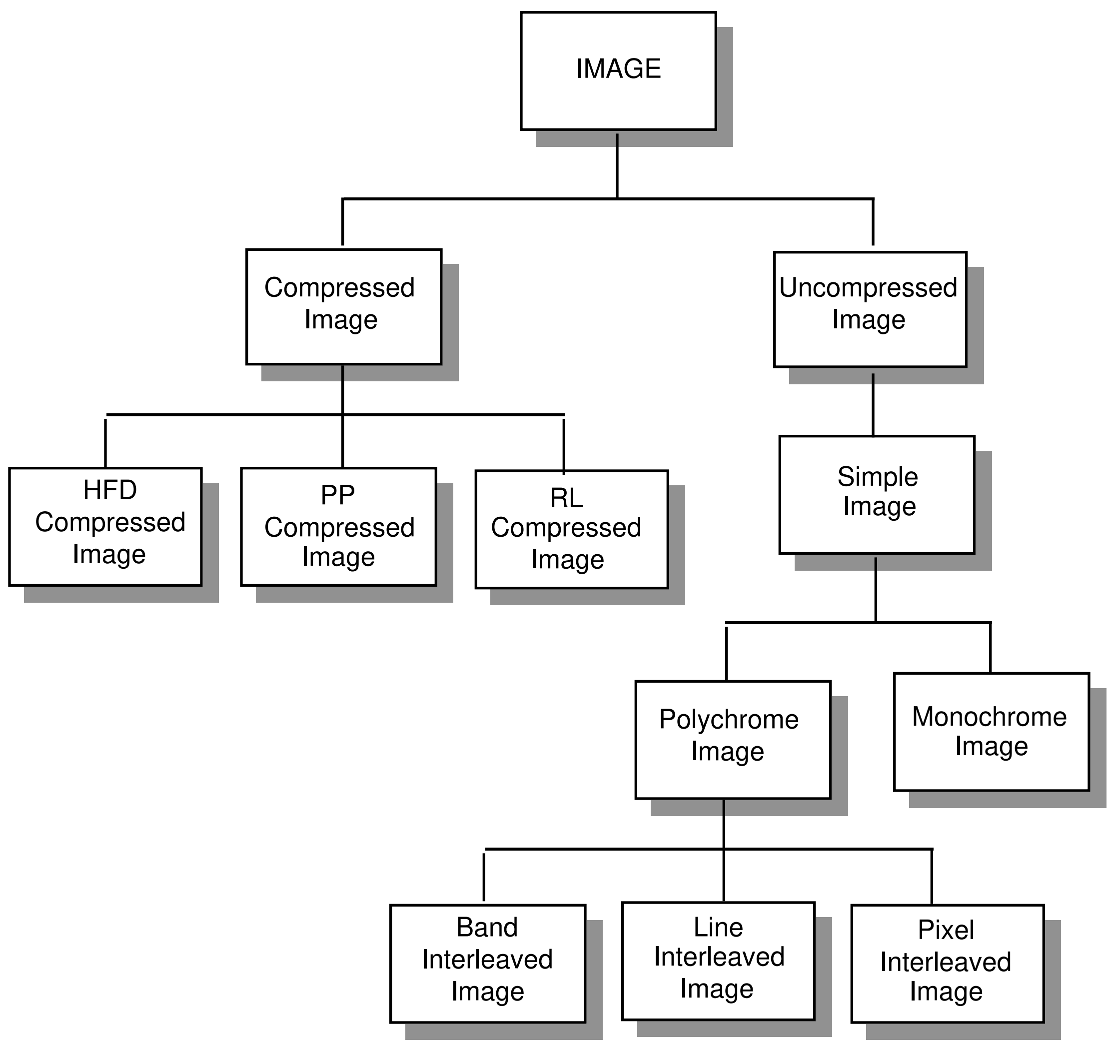

Figure 1 — Subtypes of Image Objects

Every object routine for images takes in one or more of these subtypes and returns an object of the same or a different subtype. An example is a decompression routine that takes in an HFD Compressed Image object and returns an Uncompressed Image object. Another example might be a routine to convert a band interleaved image to pixel interleaving. This last-mentioned routine would expect a Band Interleaved Image object as its input: that is, an uncompressed polychrome image, without prefixes or suffixes (because a Band Interleaved Image object is a subtype of Simple Image) and with band interleaving. This can easily be verified by checking the proper attributes in the input image's ODL tree (namely ENCODING_TYPE, LINE_PREFIX_BYTES, LINE_SUFFIX_BYTES, BANDS and BAND_STORAGE_TYPE). If the input image is of the proper type, then the conversion can be carried out, leaving a Pixel Interleaved Image object which has the same attribute values except for BAND_STORAGE_TYPE, which will be changed to SAMPLE_INTERLEAVED).

Image	This is the standard PDS image type, which covers all images. OA routines should advertise that they can accept this type for their input only if they are able to handle *all* possible variations of PDS images.
Compressed Image	Images that are compressed using one of the standard compression schemes recognized by the PDS. Compressed images are indicated by having a non-null value for the attribute ENCODING_TYPE.
HFD Compressed Image	Images compressed using the Huffman First Difference (HFD) technique. Images of this subtype are indicated by having the value of ENCODING_TYPE equal to HUFFMAN_FIRST_DIFFERENCE.
PP Compressed Image	Images compressed using the previous-pixel scheme. Images of this subtype are indicated by having a value of ENCODING_TYPE equal to PREVIOUS_PIXEL.
Clementine Compressed Image	Images from the Clementine mission compressed using a Direct Cosine Transform compression scheme. Images of this subtype are indicated by having a value of ENCODING_TYPE starting with "CLEM-JPEG".
RL Compressed Image	Images compressed using run-length encoding. Images of this subtype are indicated by having a value of ENCODING_TYPE equal to RUN_LENGTH.
Uncompressed Image	Images that are not compressed. Images of this subtype are indicated by a null value for the attribute ENCODING_TYPE. This subtype should be used for uncompressed images that have prefixes or suffixes; images that have neither prefixes and suffixes are better modeled as belonging to the Simple Image type described below.

Simple Image	Images that do not have a prefix or suffix. Images of this subtype are indicated by having null or zero values for the two attributes LINE_PREFIX_BYTES and LINE_SUFFIX_BYTES. Simple Image objects may be monochrome or polychrome, as described below.
Monochrome Image	Images that have only a single band. Images of this subtype are indicated by a null value for the attribute BANDS. They should also have a null value for the attribute BAND_STORAGE_TYPE.
Polychrome Image	Images that have more than one band. Images of this subtype are indicated by having a non-zero value for the attribute BANDS and a non-null value for the attribute BAND_STORAGE_TYPE.
Band Interleaved Image	Images of this subtype are indicated by having the value of BAND_STORAGE_TYPE equal to BAND_INTERLEAVED.
Line Interleaved Image	Images of this subtype are indicated by having the value of BAND_STORAGE_TYPE equal to LINE_INTERLEAVED.
Pixel Interleaved Image	Images of this subtype are indicated by having the value of BAND_STORAGE_TYPE equal to SAMPLE_INTERLEAVED.

Table 1 — Definitions of Image Object Subtypes

Unless otherwise noted in the individual routine descriptions, all OAL image reading routines support the following encoding types:

- HUFFMAN_FIRST_DIFFERENCE
- PREVIOUS_PIXEL
- CLEM-JPEG The Clementine decompression software's data structures take about 300 Kbytes of memory, plus the size of the decompressed image (~111 Kbytes for the typical 288 x 384 image) and may take several minutes to run on slow processors. Clementine image decompression is supported only by *OaReadImage* .

All the OA library's image read routines support multi-band images which are not compressed and have no line prefix or suffix bytes.

The image read routines return a monochromatic image with no line prefix or suffix bytes, or a spectrum (multiband images only).

3.6.1 OaReadImage

This OA routine reads a PDS image object and returns a monochromatic image that resides in memory, with no prefix or suffix bytes .

- If the image is stored in a compressed form, it is decompressed.
- If there are a prefix or suffix for the image, they are removed.
- If the image is multi-banded, the band given by the *band* argument is read in.

Note: *OaReadObject* calls *OaReadImage* with *band*=1 if the input object is an Image type.

```
OA_OBJECT OaReadImage (ODLTREE image_node,
                       int     band)
```

image_node	in	A pointer to a node of an ODL tree that describes an Image object in a file.
band	in	The band to read in. 1 <= band <= BANDS

If the image is encoded using the HFD method, the routine searches the file's ODL tree for a sibling of the input image node which describes the encoding histogram. It then uses this histogram to decompress the image.

The routine creates an ODL tree for the image's attributes and attaches it to the object descriptor. The input ODL tree is unchanged.

3.6.2 OaOpenImage

This routine sets up an image handle object for subsequent calls to *OaReadPartialImage* or *OaReadImagePixels*, routines used for reading images too large to be stored in memory at once. It initializes a stream descriptor, opens the file, and positions the file pointer to the start of the image. It initializes an image handle structure, and attaches it to the *appl1* field of the object descriptor. It then initializes the object descriptor's ODL tree, which is used internally to describe the partial image stored in the image handle's buffer.

```
OA_OBJECT OaOpenImage (ODLTREE image_node,
                       int     band)
```

image_node	in	A pointer to a node of an ODL tree that describes an Image object in a file.
band	in	The band of a multi-band image to read. Ignored for monochromatic images.

The routine returns a pointer to an object descriptor, which can be passed into *OaReadPartialImage* and *OaReadImagePixels.*

The routine supports the following encoding types: HUFFMAN_FIRST_DIFFERENCE and PREVIOUS_PIXEL. The routine also supports multi-band images which are uncompressed and have no line prefix or suffix bytes.

3.6.3 OaReadPartialImage

This routine reads a portion of an Image object from a file into memory. The result is a monochromatic image with no line prefix or suffix bytes.

```
OA_OBJECT OaReadPartialImage (OA_OBJECT   image_handle_object,
                              long        start_line,
                              long        stop_line,
                              long        start_sample,
                              long        stop_sample)
```

image_handle_object in/out	Pointer to an object descriptor returned by *OaOpenImage*.
start_line in	The first line of the image to be included in the output image, in the range 1..LINES.
stop_line in	The last line of the input image to be included, in the range start_line..LINES.
start_sample in	The first sample of each line of the input image to be included in the output image, in the range 1..LINE_SAMPLES.
stop_sample in	The last sample of each line of the input image to be included in the output image, in the range start_sample..LINE_SAMPLES.

The processing performed by this routine is similar to the processing described for *OaReadImage* above. The routine can be called repeatedly to get different portions of a single monochromatic image (or different portions of a single band of a multi-band image); the input file remains open, and the image handle keeps track of the position in the file and the state of decompression, when applicable. This is useful when the image is too large to fit into memory all at once. The first call to *OaReadPartialImage* must be preceeded by a call to *OaOpenImage*, and the last call to *OaReadPartialImage* should be followed by a call to *OaCloseImage*. The value returned is a pointer to an object descriptor for a monochromatic image with no line prefix or suffix bytes.

3.6.4 OaReadImagePixels

This is a low-level object layer routine which is normally not called by users. *OaReadPartialImage*, described above, is built on top of *OaReadImagePixels* and is the preferred routine to use. *OaReadImagePixels* returns pixels (samples) from an image starting at the specified line/sample location. The samples are decompressed, when applicable, and converted to native binary format according to *oa_profile*. The file is left open. On exit, the image handle's ODL tree describes the number of pixels read and where they came from in the image, and its *data_ptr* points to the pixels. The caller should not modify or delete the image handle object - this is done by a call to *OaCloseImage*.

```
int OaReadImagePixels (OA_OBJECT image_handle_object,
                       long      start_line,
                       long      start_sample)
```

image_handle_object in/out	Pointer to an object descriptor returned by *OaOpenImage*.
start_line in	The line at which to start returning samples, in the range 1..LINES.
start_sample in	The sample number at which to start returning samples, in the range 1..LINE_SAMPLES.

The routine returns the number of pixels read. This is variable, depending on the start location, the line length and the compression type. The first call to *OaReadImagePixels* must be preceeded by a call to *OaOpenImage*, and the last call to *OaReadImagePixels* should be followed by a call to *OaCloseImage*.

3.6.5 OaCloseImage

This routine closes the data file opened by *OaOpenImage* and frees all components of the image handle object. This should be called after the last call to*OaReadPartialImage* or *OaReadImagePixels*.

```
int OaCloseImage (OA_OBJECT image_handle_object)
```

image_handle_object in/out	Pointer to an object descriptor returned by *OaOpenImage*.

The routine always returns zero.

3.6.6 OaImportImage

This routine takes a pointer to image data in memory, creates an ODL node of class Image, adds keywords corresponding to the input arguments, then packages the ODL tree and data pointer into an object descriptor.

```
OA_OBJECT OaImportImage (PTR   data_ptr,
                         long  lines,
                         long  line_samples,
                         char  *sample_type,
                         int   sample_bits)
```

data_ptr in	Pointer to memory containing the image to be imported. PTR is a typedef equivalent to char * on most platforms and char huge * on the IBM PC.
lines in	Number of lines in the image.

line_samples	in	Number of samples per line.
sample_type	in	PDS data type of each image sample.
sample_bits	in	Length of each sample, in bits.

The resulting ODL tree for the Image object has only the four required attributes corresponding to the input arguments. The user can add additional attributes using L3 routines, if desired.

3.6.7 OaGetPartialImage

This routine is similar to *OaReadPartialImage* described above, but the input image must already be in memory. This routine can be called without calling *OaOpenImage* and *OaCloseImage*, since no file is involved.

```
OA_OBJECT OaGetPartialImage (OA_OBJECT simple_image,
                             long      start_line,
                             long      stop_line,
                             long      start_sample,
                             long      stop_sample)
```

simple_image	in	A pointer to an object descriptor for the source image object, a monochromatic image with no line prefix or suffix bytes.
start_line	in	The first line of the image to be included in the output image, in range 1..LINES.
stop_line	in	The last line of the source image to be included, in the range start_line..LINES.
start_sample	in	The first sample of each line of the source image to be included in the output image, in the range 1..LINE_SAMPLES.
stop_sample	in	The last sample of each line of the source image to be included in the output image, in the range start_sample..LINE_SAMPLES.

3.6.8 OaConvertImagetoArray

This routine converts an Image object into an Array object.

```
OA_OBJECT OaConvertImagetoArray (OA_OBJECT simple_image)
```

simple_image	in	Pointer to the object descriptor for the image to be converted.

The routine does not modify the object data in any way. An ODL tree is created for the Array object and the attributes for the Image object are translated into the corresponding attributes of an Array object and placed into the ODL tree. An object descriptor is then returned for the Array object. The input ODL tree must describe a monochromatic image with no prefix or suffix bytes.

3.6.9 OaReadSpectrumFromImage

This OA routine reads a spectrum from a multiband image file at the given line/sample location, and returns a Spectrum object in memory. It creates an ODL tree for the Spectrum object's attributes and attaches the ODL tree and the object data to an object descriptor that is returned as the function value. The resulting OA_OBJECT is a SPECTRUM-class object.

```
OA_OBJECT OaReadSpectrumFromImage (ODLTREE image_node,
                                   int     line,
                                   int     sample)
```

image_node	in	A pointer to a node of an ODL tree describing an Image type object, with multiple bands.
line	in	The line number of the spectrum.
sample	in	The sample number of the spectrum.

The routine supports multi-band images which are uncompressed and have no line prefix or suffix bytes.

3.7 OA Routines for Table Objects

Like Image objects, Table objects have some options that can complicate their representation in memory as well as their interpretation. To simplify the handling of tables in memory, we define a new subtype called Simple Table which is a Table object without any prefix or suffix. The two read routines for tables described below always return a Simple Table object.

All the routines for Table objects also work on Series, Spectrum, Palette and Gazetteer objects, except for routines specific to column-major Tables.

3.7.1 OaReadTable

This OA routine reads a Table type object from a file into memory, translating the data into the format specified by the host profile. It creates an ODL tree for the object's attributes and attaches the ODL tree and the object data to an object descriptor that is then returned as the function value. Note: *OaReadTable* is called by*OaReadObject* if the input object is a Table type.

```
OA_OBJECT OaReadTable (ODLTREE table_node)
```

table_node	in	A pointer to a node of an ODL tree describing a Table type object.

If the table has any prefix or suffix, they are stripped off. The routine returns a pointer to the Simple Table object's descriptor. The input ODL tree is unchanged.

This routine can not handle column-major tables that contain Container objects, nor can it handle column-major tables that have row-prefix bytes or row-suffix bytes.

3.7.2 OaReadSubTable

This OA routine reads a portion of a Table type object from a file into memory, translating the data into the format specified by the host profile. It creates an ODL tree for the object's attributes and attaches the ODL tree and the object data to an object descriptor that is then returned as the function value.

```
OA_OBJECT OaReadSubTable (ODLTREE table_node,
                          long    start_row,
                          long    stop_row,
                          ODLTREE subobject_nodes[],
                          int     n_subobject_nodes)
```

table_node	in	A pointer to a node of an ODL tree describing a Table type object.
start_row	in	The first row of the table to be included in the output table.

stop_row	in	The last row of the table to be included in the output table.
subobject_nodes	in	The columns and containers to be included in the output table.
n_subobject_nodes	in	The number of elements in the subobject_nodes array.

If the table has any prefix or suffix, they are stripped off. The routine returns a pointer to the Simple Table object's descriptor. The input ODL tree is unchanged.

The subobject nodes can be a mixture of Columns and Containers; all must be directly under the top-level Table object, not nested under another Container. This routine cannot handle column-major tables. The code fragment,

```
OA_OBJECT partial_table;
ODLTREE table_node, column_nodes[2];
long rows;

OaKwdValuetoLong( "ROWS", table_node, &rows);
column_nodes[0] = LeftmostChild( table_node);
column_nodes[1] = RightSibling( column_nodes[0]);
partial_table = OaReadPartialTable( table_node, 1, rows,
                                    column_nodes, 2);
```

reads the first two columns of the input table and returns the new table object in *partial_table*.

3.7.3 OaGetSubTable

This OA routine extracts a portion of a Table type object from a table in memory. It creates an ODL tree for the object's attributes and attaches the ODL tree and the object data to an object descriptor that is then returned as the function value. No data translations are done.

```
OA_OBJECT OaGetSubTable (OA_OBJECT table_object,
                         long      start_row,
                         long      stop_row,
                         ODLTREE   subobject_nodes[],
                         int       n_subobject_nodes)
```

table_object	in	A pointer to an object descriptor for an in-memory Table object.
start_row	in	The first row of the table to be included in the output table.
stop_row	in	The last row of the table to be included in the output table.
subobject_nodes	in	The columns and containers to be included in the output table.
n_subobject_nodes	in	The number of elements in the subobject_nodes array.

If the table has any prefix or suffix, they are stripped off. The routine returns a pointer to the Simple Table object's descriptor. The input ODL tree is unchanged.

The subobject nodes can be a mixture of Columns and Containers; all must be directly under the top-level Table object, not nested under another Container. This routine can not handle column-major tables.

3.7.4 OaJoinTables

This OA routine appends *table_B* to the end of *table_A*. If option is OA_ADD_ROWS, then all of *table_B's* rows are added after the last row of *table_A*; *table_B* must have the same column structure and ROW_BYTES as *table_A*. If *option* is OA_ADD_COLUMNS, then *table_B's* columns are added after the last column of *table_A*; the number of rows in *table_A* and *table_B* must be the same. In either case, *table_A* is returned changed, and *table_B* is unchanged. *Table_A* has a new ODL tree, and a new data_ptr and size. *Table_A's* object descriptor pointer is then returned as the function value.

```
OA_OBJECT OaJoinTables (OA_OBJECT table_A,
                        OA_OBJECT table_B,
                        int       option)
```

table_A	in/out	Pointer to an object descriptor for the Table object to append to.
table_B	in	Pointer to an object descriptor for the Table object to append.
option	in	OA_ADD_ROWS or OA_ADD_COLUMNS.

Both tables must have the same interchange format and be row-major.

3.7.5 OaDeleteRow

This OA routine removes a row of data from the object data of a Table, and updates the object's ROWS keyword.

```
OA_OBJECT OaDeleteRow (OA_OBJECT  table_object,
                       long       row)
```

table_object	in/out	Pointer to an object descriptor for a Table object.
row	in	The row to delete, in the range 1..ROWS.

If successful, the function returns the table_object pointer, otherwise NULL. This function will return an error if you attempt to delete the only row in a table.

3.7.6 OaDeleteColumn

This OA routine removes a column or container of data from the object data of a Table, and removes the ODL tree node from the object's ODL tree.

```
OA_OBJECT OaDeleteColumn (OA_OBJECT table_object,
                          ODLTREE   input_node)
```

table_object	in/out	Pointer to an object descriptor for a Table object.
input_node	in	Pointer to the node to delete in the table_object's ODL tree.

If successful, the function returns the *table_object* pointer, otherwise NULL. The *input_node* must be a Column or Container directly under the top-level Table object, not nested under another Container, and cannot be the only Column or Container in the table.

3.7.7 OaTransposeTable

This OA routine converts the data and ODL tree of an in-memory Table object from one TABLE_STORAGE_TYPE to another. If the table is ROW_MAJOR, it is converted to COLUMN_MAJOR, and if it is COLUMN_MAJOR, it is converted to ROW_MAJOR.

```
OA_OBJECT OaTransposeTable (OA_OBJECT table_object)
```

table_object	in/out	Pointer to an object descriptor for a Table object.

If successful, the function returns the *table_object* pointer, otherwise NULL. The table's old data has been freed and replaced with the transposed data, and has a new ODL tree. The input Table object must not have any Containers as sub-objects.

3.7.8 OaAddLineTerminatorstoTable

This OA routine copies an ASCII table object's data, appends a <CR><LF> to the end of each row, frees the old data and updates the ODL tree.

```
OA_OBJECT OaAddLineTerminatorstoTable (OA_OBJECT table_object)
```

table_object	in/out	Pointer to an object descriptor for a Table object.

If successful, the function returns the table_object pointer, otherwise NULL. PDS recommends every row of an ASCII table be terminated with a carriage-return/line-feed (<CR><LF>) pair.

3.7.9 OaAddContainerAroundTable

This OA routine only effects the object's ODL tree; its data is unchanged. It takes all the columns and containers in a table, and adds a new Container object to enclose them. The REPETITIONS keyword value in the Container is set equal to the table's ROWS keyword value, and the ROWS keyword value changed to one. The resulting ODL tree is Table node with ROWS = 1, with a single Container node below it. Below the Container node are all the nodes which were originally below the Table node. The input Table must be row-major.

```
OA_OBJECT OaAddContainerAroundTable (OA_OBJECT table_object)
```

table_object	in/out	Pointer to an object descriptor for a Table object.

If successful, the function returns the table_object pointer, otherwise NULL. The table must be row-major. This routine is useful in conjunction with *OaJoinTables* when building up a table in memory which has Container sub-object(s): be sure you modify ROWS and REPETITIONS appropriately before calling *OaJoinTable*.

3.7.10 OaUnravelContainer

This OA routine only effects the object's ODL tree; its object data is unchanged. It takes a table whose only sub-object is a Container, removes the container object, and places all the container's sub-objects directly under the table node. It then multiplies the table's ROWS keyword value by the container's REPETITIONS keyword value.

```
OA_OBJECT OaUnravelContainer (OA_OBJECT table_object)
```

table_object	in/out	Pointer to an object descriptor for a Table object.

If successful, the function returns the table_object pointer, otherwise NULL. The table must be row-major. This routine is useful in the process of extracting a column of data from a container, or getting at nested containers and their columns. The input Table must be row-major.

3.7.11 OaImportColumn

This routine converts a C data array into a PDS Table object which contains a single column.

```
OA_OBJECT OaImportColumn (PTR   data_ptr,
                          long  rows,
                          long  items,
                          long  item_bytes,
                          char  *data_type,
                          int   interchange_format,
                          char  *name)
```

data_ptr	in	Pointer to the C data array that contains the column's data. PTR is a typedef equivalent to char * on most platforms and char huge * on the IBM PC under DOS.
rows	in	The number of rows in the column.
items	in	The number of repetitions within each row of the column.
item_bytes	in	The length in bytes of each item within the column.
data_type	in	An indicator of the type of data in the column. The values are defined by the enumeration type *oa_PDS_data_types_enum*.
interchange_format	in	Indicator of whether the data are binary or ASCII coded. The choices are defined by the enumeration type *oa_interchange_format_enum*.
name	in	A pointer to a character string that gives the name that is to be associated with the column in the NAME keyword.

This function creates an OA_OBJECT and sets its data_ptr to the input pointer *data_ptr*. The data pointed to by *data_ptr* should have been dynamically allocated by the caller using*OaMalloc*, and be layed out consistent with the input keyword values. The routine creates an ODL tree and attaches it to the object. The root node of the ODL tree is a Table node, with a single Column node below it.

3.8 OA Routines for Qube Objects

The Qube is a complicated object, which already has specialized software to deal with it, so the OA library currently only handles a small subset of Qubes, and operations are restricted to reads. These OA routines handle Standard ISIS Qubes whose AXIS_NAME keyword is "(SAMPLE, LINE, BAND)" (i.e. band-sequential storage organization), and which have no bottom planes. This accounts for the vast majority of existing Qubes of general interest to planetary scientists.

3.8.1 OaReadImageFromQube

This OA routine reads a single band from a Qube file and returns a Monochrome Image object in memory. It creates an ODL tree for the object's attributes and attaches the ODL tree and the object data to an object descriptor that is then returned as the function value. The resulting OA_OBJECT is an Image-class object, which can then be passed to OA image routines.

```
OA_OBJECT OaReadImageFromQube (ODLTREE qube_node,
                               int     band)
```

qube_node	in	A pointer to a node of an ODL tree describing a Qube type object in a file.
band	in	The number of the band to read, starting with 1.

3.8.2 OaReadSpectrumFromQube

This OA routine reads a spectrum from a Qube file corresponding to the given line/sample, and returns a Spectrum object in memory. It creates an ODL tree for the object's attributes and attaches the ODL tree and the object data to an object descriptor that is then returned as the function value. The resulting OA_OBJECT is a Spectrum-class object, which can then be passed to OA table routines.

```
OA_OBJECT OaReadSpectrumFromQube (ODLTREE qube_node,
                                  int     line,
                                  int     sample)
```

qube_node	in	A pointer to a node of an ODL tree describing a Qube type object in a file.
line	in	The line number of the spectrum.
sample	in	The sample number of the spectrum.

Chapter 4. Utility Routines

The following routines are part of the OA Library and provide services that can be used by applications.

4.1 Memory Management Routines

These routines exist so that users can replace the default calls to *malloc*, *realloc* and *free* inside of them with their own custom memory management functions, if desired.

4.1.1 OaMalloc

This routine and *OaRealloc* are used by OAL and L3 for all dynamic memory allocation.

```
char *OaMalloc (long bytes)
```

bytes	in	The number of bytes of memory to allocate.

A pointer to memory is returned, or NULL if no more memory is available.

4.1.2 OaRealloc

This routine adjusts the size of a memory block originally allocated by *OaMalloc* or *OaRealloc.*

```
char *OaRealloc( char *old_ptr,
                 long old_size,
                 long bytes)
```

old_ptr	in	A pointer to dynamic memory originally allocated by *OaMalloc* or *OaRealloc*.
old_size	in	The size of the dynamic memory block pointed to by old_ptr.
bytes	in	The size to reallocate to.

A pointer to memory is returned, or NULL if no more memory is available. Note that the *old_size* argument is not used by the default version of this routine as supplied in the OAL release, but it may be useful to users who replace *OaRealloc* with their own custom memory reallocation routine.

4.1.3 OaFree

This routine frees memory allocated by *OaMalloc* or *OaRealloc*.

```
void OaFree (PTR ptr)
```

ptr	in	A pointer to object data in dynamic memory, originally allocated by *OaMalloc* or *OaRealloc*.

4.2 Error Reporting Routines

These routines can be modified to suit a particular user interface or to filter error messages based on the value of *oa_errno*.

4.2.1 Error Reporting Mechanism

The OA Library indicates errors by function return values, text messages, and by setting a global error code variable to an appropriate error code.

- **Function Return Values:**
  If an OA routine that returns a pointer as its function value encounters an error, then it sets the global variable *oa_errno* to an appropriate error code, issues an error message with *OaReportError*, and returns NULL. If no error is encountered, then a valid pointer is returned and *oa_errno* is not set (it retains its last value).

  If an OA routine that returns an integer as its function value encounters an error, then it sets the global variable *oa_errno* to an appropriate error code, issues an error message with*OaReportError*, and returns a non-zero value. If no error is encountered, then the integer value zero is returned and *oa_errno* is not set.

- **Text Error Messages:**
  Error messages are reported to the application using the *OaReportError* routine. By default, *OaReportError* writes text error messages to the standard error output. These messages can also be routed to a file by calling *OaRouteErrorMessages*. This may be desirable if the application doesn't make stderr visible to the user, or doesn't allow writing to stderr. These two routines can be easily modified to suit a particular user interface.

- **Error Codes:**
  Global variable *oa_errno* is set before every call to *OaReportError*, allowing the code inside *OaReportError* to selectively filter messages depending on the value of *oa_errno*. When multiple calls to *OaReportError* are made by different layers of the OA Libary in response to an error, only the first caller sets *oa_errno*. For example, a routine where an error first occurs will set *oa_errno*, call *OaReportError* and return; its caller may then issue another error message, without changing *oa_errno*. In this fashion, *oa_errno* is propogated up through the layers of the OA Library to the application. A list of *oa_errno* values is in Appendix A. The values are divided into categories, each with its own range of *oa_errno* values, so that applications can easily filter error messages.

- **Numeric Error Reporting:**
  The structure layer has counters which keep track of numeric errors occurring in binary-to-binary data translations. See page 76.

4.2.2 OaReportError

This routine writes an error message to the standard error output by default. By calling *OaRouteErrorMessages* this can be changed to output to a file. All OA routines set *oa_errno* and call *OaReportError* when they detect an error. The routine can be easily modified to fit any particular user interface; see "Customizing the Error Reporting Routine" on page 9 for details.

```
int OaReportError (char *error_message)
```

error_message	in	Pointer to a string containing the error message to write.

The function always returns zero.

4.2.3 OaRouteErrorMessages

This routine sets the destination for messages written by *OaReportError*. The destination can be a file the caller has opened, a file the caller specifies by name, or the standard error output (the default).

```
int OaRouteErrorMessages (char *message_fname,
                          FILE *message_fptr)
```

message_fname	in	Pointer to a string containing a file name suitable for fopen().
message_fptr	in	An pointer to an open file descriptor.

If *message_fptr* is non-NULL, then the output of *OaReportError* is directed to it. If *message_fname* is non-NULL, then the file is opened and the output of *OaReportError* directed to it. If both are NULL, then the output of *OaReportError* is directed to the standard error output The function always returns zero.

4.3 Keyword Value Conversion Routines

The values of attributes (keywords) of an object are stored in an ODL tree in ASCII format. For applications (and OA routines) to manipulate the attribute values, they must often be translated to and from binary format. This set of routines converts attribute values from ASCII to binary and from binary to ASCII.

4.3.1 OaKwdValuetoLong

This routine finds a specified keyword in an ODL tree node, gets its value and converts it to a long integer.

```
int OaKwdValuetoLong (char     *kwd_name,
                      ODLTREE odltreenode,
                      long     *value)
```

kwd_name	in	String specifying the name of the keyword.
odltreenode	in	The ODL tree node in which to look for the keyword.
value	out	Pointer to a long integer for the result to be stored in.

If successful, the function returns zero and places the keyword value, converted to a long integer, in **value*. If unable to convert the keyword value it returns one and reports an error; if the keyword name can't be found in the ODL tree node, it returns one and doesn't report any error. The statements,

```
ODLTREE table_node;
long rows;

if (OaKwdValuetoLong( "ROWS", table_node, &rows) == 0)
{
...
}
```

gets the value of the ROWS keyword from the ODL tree node *table_node,* and enters the *if* block if it got it OK.

4.3.2 OaLongtoKwdValue

This routine adds a keyword with the given name and value to an ODL tree node. It converts the long integer to ASCII and uses this as the keyword value. If there is already a keyword in the node with the same name, it replaces its value with the new value. Otherwise it creates a new keyword and pastes it as the last keyword in the ODL tree node.

```
int OaLongtoKwdValue (char     *kwd_name,
                      ODLTREE odltreenode,
                      long     value)
```

kwd_name	in	String specifying the name of the keyword.

odltreenode	in/out	The ODL tree node in which to put the keyword.
value	in	A long integer which after binary-to-ASCII conversion becomes the keyword value.

The function returns zero if all the input arguments were valid; otherwise it returns one. The statement,

```
ODLTREE image_node;
long checksum = 1319509;

OaLongtoKwdValue( "CHECKSUM", image_node, checksum);
```

adds keyword CHECKSUM with value 1319509 to *image_node*.

4.3.3 OaKwdValuetoStr

This routine finds a specified keyword in an ODL tree node, and sets the *value* argument to point directly into the tree at the keyword's value.

```
int OaKwdValuetoStr (char    *kwd_name,
                     ODLTREE odltreenode,
                     char    **value)
```

kwd_name	in	String specifying the name of the keyword.
odltreenode	in	The ODL tree node in which to look for the keyword.
value	out	The char pointer which gets set to point to the keyword value.

If successful, the function returns zero and sets value to point directly into the tree at the keyword value. If the keyword name can't be found in the ODL tree node, it returns one and doesn't report any error. The statements,

```
ODLTREE series_node;
char *str = NULL;

OaKwdValuetoStr( "SAMPLING_PARAMETER_NAME", series_node, &str);
if (str != NULL)
{
...
}
```

points str to the keyword value of keyword *SAMPLING_PARAMETER_NAME*.

4.3.4 OaStrtoKwdValue

This routine adds a keyword to the ODL tree node with the given name and value. If there is already a keyword in the node with the same name, it replaces its value with the new value. Otherwise it creates a new keyword and pastes it as the last keyword in the ODL tree node. Internally, the keyword value is copied using *OaMalloc* before putting it into the keyword structure.

```
int OaStrtoKwdValue (char    *kwd_name,
                     ODLTREE odltreenode,
                     char    *str)
```

kwd_name	in	String specifying the name of the keyword.
odltreenode	in/out	The ODL tree node in which to put the keyword.
str	in	String specifying the keyword value.

The function returns zero if all the input arguments were valid; otherwise it returns one. The statement,

```
ODLTREE series_node;

OaStrtoKwdValue( "SAMPLING_PARAMETER_UNIT", series_node, "SECOND");
```

adds keyword *SAMPLING_PARAMETER_UNIT* with value *SECOND* to the ODL tree node *series_node*.

4.3.5 OaSequencetoLongArray

This routine finds a specified keyword in an ODL tree node, checks that its value is a sequence of integers, then converts all the sequence values from ASCII to binary and stores them in an array.

```
int OaSequencetoLongArray (char    *kwd_name,
                           ODLTREE odltreenode,
                           long    **array,
                           int     *sequence_items)
```

kwd_name	in	String specifying the name of the keyword.
odltreenode	in	The ODL tree node in which to look for the keyword.
array	in/out	Pointer which on exit points to an array of longs.
sequence_items	out	Number of items in array.

If successful, the function returns zero. If the keyword name can't be found in the ODL tree node, it returns one and doesn't report any error. If the keyword value isn't a sequence of ASCII integers, or if there is an error parsing the sequence or in the binary-to-ASCII conversions, it returns one and reports an error. This function allocates space for the array, which the user should free. The statements,

```
ODLTREE qube_node;
long *core_items;
int n_core_items=0, i;
OaSequencetoLongArray( "CORE_ITEMS", qube_node, &core_items,
                       &n_core_items);
if (n_core_items > 0)
{
  for (i=0; i<n_core_items; i++)
  {
```

```
    printf( "%ld ", core_items[i]);
  }
  OaFree( (char *) core_items);
}
```

gets the *CORE_ITEMS* keyword value, a sequence, into the *core_items* variable, prints out the values and frees the array.

4.3.6 OaSequencetoStrArray

This routine finds a specified keyword in an ODL tree node, checks that its value is a sequence, then copies each sequence value to its own string, and stores all the resulting string pointers in an array.

```
int OaSequencetoStrArray (char    *kwd_name,
                          ODLTREE odltreenode,
                          char    ***array_ptr,
                          int     *sequence_items)
```

kwd_name	in	String specifying the name of the keyword.
odltreenode	in	The ODL tree node in which to look for the keyword.
array_ptr	in/out	Pointer which on exit points to an array of string pointers.
sequence_items	out	Number of items in the output array.

If successful, the function returns zero. If the keyword name can't be found in the ODL tree node, it returns one and doesn't report any error. If the keyword value isn't a sequence, or if there is an error parsing the sequence, then it returns one and reports an error. Both the individual strings and the array of pointers to them are allocated individually and the user should free them. The statements,

```
ODLTREE qube_node;
char **core_items;
int n_core_items=0, i;
OaSequencetoStrArray( "CORE_ITEMS", qube_node, &core_items,
                      &n_core_items);
if (n_core_items > 0)
{
  for (i=0; i<n_core_items; i++)
  {
    printf( "%s ", core_items[i]);
    OaFree( (char *) core_items[i]);
  }
  OaFree( (char *) core_items);
}
```

gets the *CORE_ITEMS* keyword value, a sequence, into the *core_items* variable, prints out the values and frees each individual string, as well as the array of strings.

4.4 Object Access Helper Routines

These routines are widely used inside OA's object layer routines, and may also be useful to end-users.

4.4.1 OaGetObjectClass

This routine translates the ODL tree node's class name into an integer defined by the enumeration type *oa_object_class_enum* .

```
int OaGetObjectClass (ODLTREE odltreenode)
```

odltreenode	in	Pointer to a node of an ODL tree.

A *oa_object_class_enum* enum value is returned as the function value. The statements,

```
ODLTREE odltreenode;
int object_class;

object_class = OaGetObjectClass( odltreenode);
```

gets the object class of *odltreenode* into the *object_class* variable.

4.4.2 OaObjectClasstoStr

This routine translates an integer defined by the enumeration type *oa_object_class_enum* into a string.

```
char *OaObjectClasstoStr (int oa_object_class)
```

oa_object_class	in	An *oa_object_class_enum* value, for example, OA_CONTAINER .

The string returned as the function value is in upper case, and looks just like the *oa_object_class_enum* enum value after the "OA_", for example, "CONTAINER". If the input object class doesn't match a known object class, the string "OA_UNKNOWN_CLASS" is returned. The returned string points into a static array, and shouldn't be modified.

4.4.3 OaGetObjectInterchangeFormat

This routine finds the interchange format of a top-level object (Image, Table etc). It gets the value of the INTERCHANGE_FORMAT keyword from the input node, and translates it into an integer defined by the enumeration type *oa_interchange_format_enum*.

```
int OaGetObjectInterchangeFormat (ODLTREE TLO_object_node)
```

TLO_object_node	in	Pointer to a node of an ODL tree.

An *oa_interchange_format_enum* enum value is returned as the function value: OA_BINARY_INTERCHANGE_FORMAT, OA_ASCII_INTERCHANGE_FORMAT or OA_UNKNOWN_INTERCHANGE_FORMAT. If the INTERCHANGE_FORMAT keyword doesn't exist in the input node, it returns OA_BINARY_INTERCHANGE_FORMAT, the default per the PDS Standards Document. If the object class is HISTORY, it returns OA_ASCII_INTERCHANGE_FORMAT.

4.4.4 OaStrtoPDSDataType

This routine translates a DATA_TYPE keyword value string into an integer defined by the enumeration type *oa_PDS_data_types_enum* .

```
int OaStrtoPDSDataType (char *str,
                        int  interchange_format)
```

str	in	Pointer to a string containing a PDS data type, (usually points to the DATA_TYPE keyword value in an ODL tree node).
interchange_format	in	An *oa_interchange_format_enum* value: OA_ASCII_INTERCHANGE_FORMAT or OA_BINARY_INTERCHANGE_FORMAT.

A *oa_PDS_data_types_enum* value is returned as the function value, for example, if *str* = "INTEGER" and *interchange_format* = OA_ASCII_INTERCHANGE_FORMAT, then the function returns OA_ASCII_INTEGER.

4.4.5 OaPDSDataTypetoStr

This routine translates an integer defined by the enumeration type *oa_PDS_data_types_enum* into a string.

```
char *OaPDSDataTypetoStr (int PDS_data_type)
```

PDS_data_type	in	A *oa_PDS_data_types_enum* value, for example, OA_VAX_INTEGER .

The string returned as the function value is in upper case, and is the de-aliased version of the data type. For example, if PDS_data_type is OA_VAX_INTEGER, then the output is "LSB_INTEGER". If the input data type doesn't match a known data type, the string "UNK" is returned. The returned string points into a static array, and shouldn't be modified.

4.4.6 OaGetImageKeywords

This routine gets and translates the keyword values of useful Image keywords.

```
int OaGetImageKeywords (ODLTREE  image_node,
                        long      *lines,
                        long      *line_samples,
                        long      *sample_bits,
                        char     **sample_type_str,
                        long      *bands,
                        int       *band_storage_type,
                        long      *line_prefix_bytes,
                        long      *line_suffix_bytes,
                        int       *encoding_type)
```

image_node	in	Pointer to a node of an ODL tree of class Image.
lines	out	The value of the LINES keyword, converted to a long.
line_samples	out	The value of the LINE_SAMPLES keyword, converted to a long.
sample_bits	out	The value of the SAMPLE_BITS keyword, converted to a long.
sample_type_str	out	Points directly into the tree at the SAMPLE_TYPE keyword value, so should not be freed.
bands		The value of the BANDS keyword, converted to a long. If the keyword doesn't exist, *bands is set to 1.
band_storage_type	out	The value of the BAND_STORAGE_TYPE keyword, converted to a *oa_band_storage_types_enum* value.
line_prefix_bytes	out	The value of the LINE_PREFIX_BYTES keyword, converted to a long. If the keyword doesn't exist, *line_prefix_bytes is set to 0.
line_suffix_bytes	out	The value of the LINE_SUFFIX_BYTES keyword, converted to a long. If the keyword doesn't exist, *line_suffix_bytes is set to 0.
encoding_type	out	The value of the ENCODING_TYPE keyword, converted to a *oa_compression_types_enum* value.

If successful, the function returns 0. If the input node wasn't of class Image, or if any of the required keywords weren't present, then the function returns 1 and the output parameters may not all be set. The statements,

```
ODLTREE image_node;
long lines, line_samples, sample_bits, bands;
long line_prefix_bytes, line_suffix_bytes;
int result, encoding_type, band_storage_type;
char *sample_type_str;

result = OaGetImageKeywords( image_node, &lines, &line_samples,
                             &sample_bits, &sample_type_str, &bands,
```

```
                                      &band_storage_type, &line_prefix_bytes,
                                      &line_suffix_bytes, &encoding_type);
if (result == 0)
{
...
}
```

gets the image keywords for *image_node*.

4.4.7 OaGetTableKeywords

This routine gets and translates the keyword values of useful Table keywords.

```
int OaGetTableKeywords (ODLTREE  table_node,
                        long     *rows,
                        long     *row_bytes,
                        long     *row_prefix_bytes,
                        long     *row_suffix_bytes,
                        int      *interchange_format,
                        int      *table_storage_type)
```

table_node	in	Pointer to a node of an ODL tree with a Table-like class.
rows	out	The value of the ROWS keyword, converted to a long.
row_bytes	out	The value of the ROW_BYTES keyword, converted to a long.
row_prefix_bytes	out	The value of the ROW_PREFIX_BYTES keyword, converted to a long. If the keyword isn't present, then *row_prefix_bytes is set to 0.
row_suffix_bytes	out	The value of the ROW_SUFFIX_BYTES keyword, converted to a long. If the keyword isn't present, then *row_suffix_bytes is set to 0.
interchange_format	out	The value of the INTERCHANGE_FORMAT keyword, converted to a *oa_interchange_format_enum* value. If the keyword isn't present, then the value is set to OA_BINARY_INTERCHANGE_FORMAT.
table_storage_type	out	The value of the TABLE_STORAGE_TYPE keyword, converted to a *oa_table_storage_type_enum* value. If the keyword isn't present, then the value is set to OA_ROW_MAJOR.

If successful, the function returns 0. If the input node didn't have a Table-like class, or if any of the required keywords weren't present, then the function returns 1 and the output parameters may not all be set.

4.4.8 OaGetQubeKeywords

This routine gets and translates the keyword values of useful QUBE keywords.

```
int OaGetQubeKeywords (ODLTREE  qube_node,
                       long     **core_items,
                       char     ***axis_names,
                       long     **suffix_items,
                       long     *core_item_bytes,
                       char    **core_item_type)
```

qube_node	in	Pointer to a node of an ODL tree of class QUBE.
core_items	out	Address of a pointer to a long, which on return points to a 3-element array of longs allocated by the routine. The caller should free the array when finished with it.
axis_names	out	Points to a 3-element array of strings allocated by the routine. The caller should free each string and the array itself when finished with it.
suffix_items	out	Points to a 3-element array of longs allocated by the routine. The caller should free the array when finished with it.
core_item_bytes	out	Address of a long; on return the long contains the value of the CORE_ITEM_BYTES keyword.
core_item_type	out	Points directly into the ODL tree at the CORE_ITEM_TYPE keyword value, so it should not be freed.

The QUBE must be a standard ISIS QUBE with 3-item sequences for CORE_ITEMS, AXIS_NAME and SUFFIX_ITEMS. If successful, the function returns 0 and sets all the output parameters appropriately; otherwise it returns 1. The statements,

```
ODLTREE qube_node;
long *core_items, *suffix_items, core_item_bytes;
char **axis_names, *core_item_type
int result, i;

result = OaGetQubeKeywords( qube_node, &core_items, &axis_names,
                            &suffix_items, &core_item_bytes,
                            &core_item_type);
if (result == 0)
{
  ...
  for (i=0; i<3; i++)
    OaFree( (char *) axis_names[i]);
  OaFree( (char *) axis_names);
  OaFree( (char *) core_items);
  OaFree( (char *) suffix_items);
}
```

gets qube keywords for *qube_node*, and frees them after they've been used.

4.4.9 OaGetFileKeywords

This routine searches upwards in the ODL tree associated with a file to find various file and object attributes, which are returned as output parameters. It returns the first occurrence of each attribute as it searches upwards, starting at the input node.

```
int OaGetFileKeywords (ODLTREE odltreenode,
                       char    **label_filename,
                       char    **data_filename,
                       int     *record_type,
                       long    *record_bytes,
                       long    *file_records,
                       long    *file_offset,
                       int     *object_interchange_format)
```

odltreenode	in	Pointer to a node of an ODL tree associated with a file. This node is normally a "top-level" object node, i.e. a Table, Image etc., the search for file keywords starts here and works up.
label_filename	out	A pointer to the name of the file that contains the label; this includes the directory path, if one was specified when the label was read in by *OaReadLabelFile*. The caller should free this when finished with it.
data_filename	out	A pointer to the name of the file that contains the object data. The caller should free this when finished with it. If the data file is in the same directory as the label file, then the directory path of the label is included, if one was specified when the label was read in by *OaReadLabelFile*.
record_type	out	Indicator of the type of records in the file.
record_bytes	out	The length of records, in bytes. For variable length records, this is the maximum length.
file_records		The number of records in the file.
file_offset	out	For stream and fixed length records, this is the byte offset of the object in the file. For variable length records, this is the record offset of the object. Both start at zero (the start of the file).
object_interchange_ format	out	Indicator of whether the object is in binary or ASCII format. The choices are defined by the enumeration type *oa_interchange_format_enum*.

The routine opens the data filename given in the label, with the path of the label file prepended to the name, to verify the data file exists; if unsuccessful, the routine tries variations on the name (lower-case, appended ";1") and looks for it in the current directory. If successful, the routine sets *data_filename* to the pathname it was able to open. If the routine determines that the data file is inside a Zip compressed file, then it spawns Unzip to decompress it, and sets *data_filename* to the unzipped file name.

4.4.10 OaCopyTree

This routine makes a copy of the ODL tree below and including the input node, optionally stripping off keywords, comments and/or SDT nodes. (SDT nodes are never present in the user's ODL trees, but often are present in ODL trees internal to OA routines.)

```
ODLTREE OaCopyTree (ODLTREE input_node,
                    int     options)
```

input_node	in	Pointer to a node of an ODL tree.
options	in	A flag indicating whether keywords, comments and/or SDT nodes will be stripped out. The defined values OA_STRIP_KEYWORDS, OA_STRIP_COMMENTS and OA_STRIP_SDT_NODES can be OR'd together in any combination. If options is zero, then nothing is stripped out.

A pointer to the copy of the ODL tree is returned as the function value. The statements,

```
OA_OBJECT oa_object;
ODLTREE odltree;

odltree = OaCopyTree( oa_object->odltree, 0);
```

copies the ODL tree in an *OA_OBJECT*.

4.4.11 ODL Tree Navigation Macros

The following macros are provided in *oal.h* to facilitate moving around in ODL trees: *LeftmostChild*, *RightmostChild*, *LeftSibling*, *RightSibling*, *Parent*.. Each macro takes an ODLTREE type as input and returns an ODLTREE or NULL. The caller should never supply a NULL pointer to these macros, as there is no error checking.

Chapter 5. IDL Interface

IDL (Interactive Data Language) is a widely used scientific data analysis package. Most OAL object layer routines, utility routines and L3 routines are callable from IDL. An IDL program can read PDS Objects into memory, translate them into IDL variables, then manipulate them using IDL. IDL variables can be translated into OA objects then manipulated using OAL routines. Applications which use the OA Library can be written entirely in IDL. The *examples* subdirectory of the release and Appendix D.2 of this User's Guide contains example programs written in IDL which use the OA Library.

5.1 Overview

- **Calling sequences:**
  For every C object layer routine, utility routine and L3 routine, there is an IDL function (called a "wrapper") with the same name as the C routine, which takes the closest IDL equivalents of the C arguments. Macros used in OAL such as *LeftmostChild* are also available as IDL functions. For routines whose C argument is a pointer, the IDL argument should be the IDL equivalent of what the C argument points to. For example, when a C routine's argument is a pointer to an integer, the corresponding IDL function argument should be an IDL integer. When an OA routine's C argument is an integer or a long (or a pointer to an integer or long), an IDL integer or long can be passed; the wrappers convert the input argument to the type required by the C function so users don't have to worry about whether to pass an IDL integer or long.

- **Pointers:**
  Pointers are returned to IDL from OAL and L3 functions as IDL 'double' variables. These pointers are inscrutable within IDL; they can be stored and compared to NULL (represented by 0.0), but only become useful when passed back into OAL and L3 functions, or into *OaIDLGetObjectData*.

- **Converting OA Objects into IDL variables:**
  The object data of an in-memory OA object can be gotton into an IDL variable by calling the *OaIDLGetObjectData* function.
  An OA Image object is returned as a 2-dimensional array variable.
  An OA Histogram object is returned as a 1-dimensional array variable.
  An OA Table-like object is returned as an IDL array of structures. The number of array elements equals the number of rows in the Table, thus each structure in the array contains one row of the Table. Each structure tag name is the same as the corresponding NAME keyword in the ODL tree's column. Tables with any level of nested Containers are supported; each Container becomes a nested structure.

- **OAL and IDL memory spaces:**
  OAL stores object in memory allocated by *OaMalloc*. When object data is converted to an IDL variable, IDL code within the interface allocates space for the IDL variable in IDL, and the data is copied from the OAL memory space to

the IDL variable. Once OAL object data has been converted to an IDL variable, the object can be deleted (memory freed) by a call to *OaDeleteObject* without effecting the IDL variable.

- **How IDL variables are converted into OA Objects:**
  IDL variables of certain types can be converted into an OA object by calling the *OaIDLVariabletoOaObject* function.

- **OA_OBJECT and ODLTREE structures:**
  The contents of OA_OBJECT and ODLTREE structures can be gotton into IDL variables with *OaIDLGetOaObjectStruct* and *OaIDLGetODLTreeNodeStruct* These functions return IDL structures which contain IDL versions of the contents of the C structure. Although individual ODL tree node pointers can be stored in IDL, the actual linked tree structure is still in OAL's memory space, and not directly accessible in IDL. This is not a limitation, because all L3 tree manipulation functions can be called from IDL; the tree being manipulated is the ODL tree in OAL's memory space, not an IDL variable.

- **Enumerated types:**
  Enumerated types commonly used in calling OAL and L3 routines are available in IDL via the *!OA* and *!ODL* system variables. For example, where in C code you use *ODL_RECURSIVE_DOWN*, in IDL you use *!ODL.RECURSIVE_DOWN*. See the end of *OAL_IDL_interface.pro* for a complete list.

- **Prevent screen output from OAL and L3:**
  Screen output from OAL and L3 should be inhibited when IDL is running, because on certain platforms it will crash IDL. Always specify an error message file when calling *OaParseLabelFile*, and call *OaRouteErrorMessages* at the start of your IDL application to redirect messages issued internally by *OaReportError* (or modify *OaReportError* in *rprt_err*.c to write somewhere else besides the default stderr).

- **Supported Platforms:**
  This interface is portable to any platform on which the IDL *CALL_EXTERNAL* function is available. The IDL interface has been tested on these platforms: Sun SPARCstation, SGI, Dec Alpha/OSF, PowerMac with CodeWarrier.

- **Installation:**
  A Makefile for Unix platforms is provided in the release to create the shareable library containing the OA library, L3 and *OAL_IDL_interface.c*.
  The variable *OAL_SHARED_LIB_NAME* in the file *OAL_IDL_interface.pro* must be set to the path name of the shareable library file where the compiled OAL code is located; edit *OAL_IDL_interface.pro* and set it for your system.
  Once the library is created, compile the IDL part of the interface code and define the *!OAL* and *!ODL* system variables with:

```
IDL> .run OAL_IDL_interface.pro
```

5.2 IDL Interface Routines

5.2.1 OaIDLReadObject

This IDL procedure reads a PDS object from a file into an IDL variable; it is provided as a useful user interface, and as an example of using OAL from IDL.

```
pro OaIDLReadObject( IDL_VARIABLE, OA_OBJECT,
                     FILE_NAME=FILE_NAME,
                     OBJECT_NAME=OBJECT_NAME)
```

IDL_VARIABLE	out	The IDL variable in which to return the object data.
OA_OBJECT	out	Optional parameter; if present, the OA_OBJECT pointer is returned in it, which can then be passed to other OAL functions. If not present, the OA_OBJECT is deleted before returning.
FILE_NAME	in	Optional keyword giving the name or path to the label file. If not present, the procedure prompts the user for the label file name.
OBJECT_NAME	in	Optional keyword giving the name of the object to be read in. If not present, the procedure lists the object names it found in the label, and prompts the user to enter the name of the object to read.

In this interactive example, *OaIDLReadObject* prompts the user for the label file name and the object name to read. Since the user didn't specify a directory path, *OaIDLReadObject* looks for the label file in the current working directory. It then reads in the *Image* object, and returns the object data in the *VAR* variable. The user then calls help to find out about *VAR*.

```
IDL> OaIDLReadObject, VAR
Enter PDS label spec: BROWSE.LBL
Objects are: IMAGE_HEADER IMAGE
Enter object name: IMAGE
IDL> help,VAR
VAR       BYTE = Array(1024, 512)
IDL>
```

This non-interactive example accomplishes the same thing:

```
IDL> OaIDLReadObject, VAR, FILE_NAME='BROWSE.LBL', OBJECT_NAME='IMAGE'
IDL> help,VAR
VAR       BYTE = Array(1024, 512)
IDL>
```

In this example, *OaIDLReadObject* reads a Table object from a GRSFE CD-ROM, and returns an array of 360 structures, corresponding to the 360 rows in the Table:

```
IDL> OaIDLReadObject, VAR
Enter PDS label spec: /cdrom/gr_0001/daedalus/DAEDWAVE.LBL
Objects are: TABLE_HEADER TABLE
Enter object name: TABLE
IDL> help,VAR
VAR    STRUCT   = -> <Anonymous> Array(360)
IDL> help,/struct,VAR
** Structure <2120b8>, 5 tags, length=20, refs=1:
      CHANNEL_NUMBER    LONG        0
      WAVELENGTH        LONG        0
      SEGMENT_NUMBER    LONG        1
      GAIN_OFFSET       LONG        4
      DECONVOLUTION_COEFFICIENTSLONG      0
IDL>
```

5.2.2 OaIDLGetObjectData

This IDL function gets the object data of an in-memory OA object into an IDL variable.

```
function OaIDLGetObjectData( OA_OBJECT)
```

OA_OBJECT	in	An IDL double representing a pointer to an OA_OBJECT in OAL's memory space, as returned by *OaReadObject*, for example.

The function returns an IDL variable; if unsuccessful, it returns a scalar integer IDL variable with a value of zero. If successful, it returns one of the following, depending on the input object class:

- Image**:**
  The function returns a two-dimensional IDL array.
- Histogram**:**
  The function returns a one-dimensional IDL array.
- Table, Series, Spectrum, Palette, Collection:
  The function returns an IDL array of structures. The number of array elements equals the number of rows in the Table, thus each structure in the array contains one row of the Table. Each structure's tag name is the same as the NAME keyword in the corresponding ODL tree's column. Tables with any level of nested Containers are supported; each Container becomes a nested structure.
- ARRAY:
  If the ARRAY doesn't have a Collection under it, the function returns an IDL array with the same dimensions as the OAL array.
  If the array does have a Collection under it, the function returns an IDL array of structures, just like for a Table (see above).

In this example, the IDL procedure *DisplayImage* reads a label file and checks if there is an Image object in it. If so, it reads in the image object, converts it to an IDL variable and displays it.

```
pro DisplayImage, LABEL_FILENAME

ODLTREE = OaParseLabelFile( LABEL_FILENAME, 'PARSER_ERRORS.TXT',
                            !ODL.ODL_EXPAND_STRUCTURE, 1);
if ODLTREE eq 0.0 then return
IMAGE_NODE = OdlFindObjDesc( ODLTREE, '*IMAGE', '', '', 0, 0)
if IMAGE_NODE eq 0.0 then return
OA_OBJECT = OaReadObject( IMAGE_NODE)
if OA_OBJECT eq 0.0 then return
IMAGE = OaIDLGetObjectData( OA_OBJECT)
if n_elements( IMAGE) ne 1 then tvscl, IMAGE
return
end
```

5.2.3 OaIDLVariabletoOaObject

This IDL function converts an IDL variable to an OA object.

```
function OaIDLVariabletoOaObject( IDL_VARIABLE, CLASS_NAME)
```

IDL_VARIABLE	in	An IDL variable whose type is consistent with CLASS_NAME (see below).
CLASS_NAME	in	An IDL string variable giving the OA object class to convert IDL_VARIABLE to.

The function returns an IDL double, which is a pointer to the OA Object in OAL's memory space, and can be passed to OAL routines. If unsuccessful, it returns 0.0. *CLASS_NAME* is currently restricted to the following:

- 'IMAGE'**:**
  IDL_VARIABLE must be a two-dimensional array of any numeric type except complex.
- 'HISTOGRAM'**:**
  IDL_VARIABLE must be a one-dimensional array of any numeric type.
- 'TABLE', 'SERIES', 'SPECTRUM', 'PALETTE', 'COLLECTION':
  IDL_VARIABLE must be a one-dimensional array of structures. The number of array elements becomes the number of rows in the Table, so each row in the Table will contain the data from one structure in the array. The structure's tag names are used as the NAME keywords in the OA_OBJECT's ODL tree columns. Nested structures are not currently supported.

5.2.4 OaIDLGetOaObjectStruct

This IDL function gets the useful fields of an OA_OBJECT structure in OAL’s memory space into an IDL structure.

```
function OaIDLGetOaObjectStruct( OA_OBJECT)
```

OA_OBJECT	in	An IDL double returned from a previous OAL function, such as *OaReadObject*.; points to an OA_OBJECT in OAL's memory space.

The function returns this IDL structure:

```
{oa_object_struct, ODLTREE  : double(0.0), $
                   DATA_PTR : double(0.0), $
                   SIZE     : long(0)}
```

This code fragment prints out all the keyword names in the root node of an OA_OBJECT's ODL tree.

```
S = OaIDLGetOaObjectStruct( OA_OBJECT)
KWD = OdlGetFirstKwd( S.ODLTREE)
while KWD ne 0 do begin
  print, OdlGetKwdName( KWD)
  KWD = OdlGetNextKwd( KWD)
endwhile
```

5.2.5 OaIDLGetODLTreeNodeStruct

This IDL function gets the useful fields of an ODLTREE structure into an IDL structure.

```
function OaIDLGetODLTreeNodeStruct( ODLTREE)
```

ODLTREE	in	An IDL double returned from a previous OAL or ODL function; points to an ODLTREE in OAL's memory space.

The function returns this IDL structure:

```
{ODLTREE_node_struct, CLASS_NAME    : '', $
                      PRE_COMMENT   : '', $
                      LINE_COMMENT  : '', $
                      POST_COMMENT  : '', $
                      END_COMMENT   : '', $
                      FILE_NAME     : '', $
                      CHILD_COUNT   : long(0),     $
                      PARENT        : double(0.0), $
                      LEFT_SIBLING  : double(0.0), $
                      RIGHT_SIBLING : double(0.0), $
                      FIRST_CHILD   : double(0.0), $
                      LAST_CHILD    : double(0.0), $
                      FIRST_KEYWORD : double(0.0), $
                      LAST_KEYWORD  : double(0.0)}
```

This routine is rarely used, since L3 routines and macros are available in IDL to get at most of the fields of an ODLTREE. Instead of calling *OaIDLGetODLTreeNodeStruct*, call *OdlGetObjDescClassName*, *OdlGetObjDescChildCount*, *Parent*, *LeftSibling*, *RightSibling*, *LeftmostChild*, *RightmostChild*, *OdlGetFirstKwd*, or *OdlGetNextKwd*.

5.2.6 OaIDLGetProfile

This IDL procedure gets the fields of OAL's global *Oa_profile* variable into an IDL structure.

```
pro OaIDLGetProfile, OA_PROFILE
```

OA_PROFILE	out	An IDL structure variable with fields listed below.

```
{profile_struct, DST_FORMAT_FOR_ASCII_SRC  : long(0), $
                 DST_FORMAT_FOR_BINARY_SRC : long(0), $
                 DST_ALIGNMENT_TYPE        : long(0), $
                 DATA_TRANSLATION_PROFILE  : long(0), $
                 CHECK_ASCII_WRITES        : long(0)}
```

5.2.7 OaIDLSetProfile

This IDL procedure sets the fields of OAL's global *Oa_profile* variable from the input IDL structure.

```
pro OaIDLSetProfile, OA_PROFILE
```

OA_PROFILE	in	An IDL structure variable with fields listed below.

```
{profile_struct, DST_FORMAT_FOR_ASCII_SRC  : long(0), $
                 DST_FORMAT_FOR_BINARY_SRC : long(0), $
                 DST_ALIGNMENT_TYPE        : long(0), $
                 DATA_TRANSLATION_PROFILE  : long(0), $
                 CHECK_ASCII_WRITES        : long(0)}
```

Chapter 6. Fortran Interface

Most object layer routines, utility routines and L3 routines are callable from FORTRAN via wrapper routines written in C, on VAX/VMS and all Unix platforms. Macros used in OAL such as *LeftmostChild* are also provided as functions in FORTRAN. The code for these wrappers is in the files *OAL_fortran_interface.c* and *L3_fortran_interface.c* in the *source* subdirectory of the release. The two FORTRAN include files, *OAL_FORTRAN.INC* and *L3_FORTRAN.INC*, provide parameters to replace the C #defines and enumerated types, and FORTRAN function declarations for the wrappers. A documentation header preceeding each wrapper routine describes the FORTRAN calling sequence for the routine. The *examples* subdirectory of the release contains example programs in FORTRAN.

- Each object layer routine, utility routine and L3 routine has a wrapper written in C which is directly callable from FORTRAN. Each wrapper routine is simply a short piece of C code which handles the different FORTRAN parameter passing mechanism, converts strings, and calls the corresponding C routine. Most of the wrappers are called as FORTRAN functions; a few are called as subroutines.

- There are also several functions to facilitate access to OAL's global *Oa_profile* variable, and to OA_OBJECT data structures.

- ODLTREE node pointers are passed to and from FORTRAN using INTEGER*4 on all platforms except the Dec/Alpha, where INTEGER*8 are used. Users running on a Dec/Alpha must edit *OAL_FORTRAN.INC* and *L3_FORTRAN.INC*, and change the return values of the functions from integer*4 to integer*8.

- There are no common blocks or other internal storage of the pointers. FORTRAN programs layered on top of the OA library and L3 must keep track of pointers exactly the way C programs do.

- FORTRAN strings can be passed to and from the wrapper routines; conversions between FORTRAN and C strings are done internally inside the wrapper routines.

- The '%val' construct is not needed when calling these wrapper routines.

Part II — Advanced User's Guide

Chapter 7. Stream Layer

Stream layer functions are called from object layer methods to read from and write to files containing the data for one or more PDS data objects. The stream layer performs many of the same functions that are performed by the standard C language input/output routines, but it does so in a way that eliminates the differences that can arise when reading PDS data files formatted for one platform (e.g., VAX/VMS) on another type of platform (e.g., Sun/Unix). The stream layer has no knowledge of — or concern for — data objects: to the stream layer the contents of a file are simply a stream of bytes, potentially arranged into a series of records. A single call to a stream layer routine can read or write a portion of an object, a complete object, or even multiple objects.

The stream layer can handle fixed length and variable length records, as well as stream files which have embedded record delimiters (for example, a stream of ASCII text with carriage return or line feed markers delimiting each line). See Sections 14.1 to 14.3 of the *PDS Standards Reference* for further information on these formats and their use. A variable-length record has a 16-bit record control word (RCW) at the beginning of each record giving the length of the record in bytes. In the stream layer the RCW is interpreted as a VAX LSB two-byte integer. Note that the ISO-9660 CD-ROM standard allows the RCW to be an integer value in MSB order, but in practice this situation probably never occurs, and it violates PDS standards, so it is not handled by the stream layer. If the value of the RCW is odd, an extra byte with value zero is expected at the end of the record to make the record length even.

Different code is compiled into stream layer routines to handle I/O depending upon the platform on which the software is running. For example, file I/O on a variable-length record file is transparent to the software under VMS, because the VMS file I/O system specifically handles this format; but variable-length record I/O is not transparent on other systems and has to be handled explicitly by stream layer code.

The stream layer is tolerant of a limited set of discrepancies which can arise when a data file is transferred from one operating environment to another (e.g., from VAX to Unix) resulting in the data file no longer having the same record format specified in its label. Handling all of the pathological possibilities that can arise during data file transfer is, however, not possible. Instructions on how to properly transfer data files across platforms — and what to do if you have an improperly transferred file — is included in Appendix C - Transferring PDS Files Between Computers.

The stream layer is a stand-alone package which can be compiled separately from the rest of the OA library. The source files are *stream_l.c*, *stream_l.h*, *oamalloc.c*, *oamalloc.h* and *rprt_err.c*.

The data structures and routines specific to the stream layer are described below. The compilable definitions for these structures and routines are found in the include file named *stream_l.h*.

7.1 Stream Layer Data Structures

The stream layer uses a *stream descriptor* to track the status of each active input or output data stream. A stream descriptor is created through a call to *OalOpenStream*, deallocated by *OalCloseStream*, and used by the other stream layer routines. It is neither necessary nor desirable for the user to modify the contents of a stream descriptor directly: all initialization and updating is done through the stream layer routines. An exception is if the user wants to do I/O through a pipe or socket.

```
struct OaStreamStruct {
  char *filename;
  int  record_type;
  int  VMS_record_type;
  long record_bytes;
  FILE *fp;
  char *buf;
  long buf_siz;
  long current_position;
  int  flags;
};
```

filename	Name of the file open for reading or writing.
record_type	Indicator of the type of record. Values are a subset of *oa_record_type_enum* enumeration type values: OA_FIXED_LENGTH OA_VARIABLE_LENGTH OA_STREAM OA_UNDEFINED_RECORD_TYPE
VMS_record_type	Indicator of the type of RMS record on VMS systems only; not used on other platforms. Values are a subset of *oa_record_type_enum* enumeration type values: OA_FIXED_LENGTH OA_VARIABLE_LENGTH OA_STREAM OA_UNDEFINED_RECORD_TYPE
record_bytes	For fixed length records, this is the length of each record. For variable length records, this is the length of the longest record in the file. It is ignored for stream files.
fp	Pointer to the file control structure for the input/output file.
buf	Pointer to an internal buffer for input and output.
buf_siz	Length of the input/output buffer in bytes.

current_position	For stream, fixed-length and undefined record formats, this is the byte offset at which the next read or write will begin (starting from 0, the start of the file). For variable length records, this is the next record number (starting from 0).
flags	Indicator for various stream characteristics. Values are defined by the *oa_stream_flags_enum* enumeration type: OA_IS_SEEKABLE OA_MISSING_LAST_RECORD_BYTE These are bits, and one or more can be set at a time. See the *OalSeek* function for a description of the OA_IS_SEEKABLE flag. OA_MISSING_LAST_RECORD_BYTE is set internally by *OalOpenStream* when it detects a condition caused by improper file transfer.

7.2 Stream Layer Routines

This section describes the routines that make up the stream layer. The names of all these routines begin with the three letters *Oal* to indicate that they are part of the inner workings of the OA Library.

7.2.1 OalOpenStream

This routine opens a file, allocates space for a stream descriptor for the file, and initializes the descriptor.

```
struct OaStreamStruct *OalOpenStream( char  *filename,
                                      int   record_type
                                      long  record_bytes,
                                      long  file_records,
                                      char  *read_write_mode)
```

filename	in	Name of the file to be opened for reading or writing. The file name string should be suitable for use by the *fopen()* C library routine.
record_type	in	Record type indicator. See the description in Section 7.1 above. When *read_write_mode* is "r", if *record_type* is OA_VARIABLE_LENGTH and we're not running under the VMS file system, *OalOpenStream* goes through various contortions to see if the file is truly a variable-length record file. If *record_type* is OA_UNKNOWN_RECORD_TYPE, *OalOpenStream* assumes the input file starts with a PDS label, and tries to determine the record type.

record_bytes	in	Length of each record. See the description in Section 7.1 above.
file_records	in	This is an optional input which helps *OalOpenStream* verify files which were transferred over a network from another machine. If unknown, should be set to 0.
read_write_mode	in	Read/write indicator. See the description in Section 7.1 above.

The routine opens the file with the name specified by the *filename* parameter and stores a pointer to the file's control block into the stream descriptor. The routine then fills in the rest of the stream descriptor, calculating the value for *buf_siz*, allocating space for the buffer pointed to by *buf*, and setting the value of *current_position* to zero.

The function returns a pointer to the created stream descriptor. If an error occurs, the return value is set to NULL.

7.2.2 OalReadStream

This routine reads data from a file.

```
int OalReadStream (struct OaStreamStruct  *stream_id,
                   long                    bytes_to_read,
                   char                  **buf_ptr,
                   long                    file_offset,
                   long                   *bytes_read)
```

stream_id	in/out	Pointer to the stream descriptor for the file that is to be read.
bytes_to_read	in	The number of bytes to read. If the value supplied for this parameter is >0, the specified number of bytes are read and placed in the output buffer. If the parameter value is 0, the routines reads as much data as it pleases (usually the number of bytes specified by *buf_siz*).
buf_ptr	in/out	Pointer to a buffer to hold the output. If the pointer this points to is NULL on input, then the data will be read into the internal buffer pointed to within the stream descriptor and a pointer to this buffer will be returned upon exit.
file_offset	in	For stream, undefined, and fixed length records, this is the byte position to which to start reading. For variable length records, this is the record at which to start reading. The count starts at zero.
bytes_read	out	Number of bytes actually read.

If the *file_offset* parameter has a value of -1, the read starts at the current position; otherwise the *OalSeek* function is called to position the file to the position indicated by the value.

This routine removes the RCWs at the beginning of variable-length records. It also eliminates the pad byte at the end of odd-length variable length records. The routine does string-oriented *fgets*() calls for stream files and byte-oriented *fread*() calls otherwise.

If at least one byte is read, a value of zero is returned. If the stream descriptor's file pointer was already at end-of-file when *OalReadStream* was called, 1 is returned. If there was any other error, -1 is returned with an error message.

7.2.3 OalWriteStream

This routine writes data to a file.

```
int OalWriteStream (struct OaStreamStruct *stream_id,
                    long                   bytes_to_write,
                    char                  *buf,
                    long                  *bytes_written)
```

stream_id	in/out	Pointer to the stream descriptor for the file that is to be written.
bytes_to_write	in	The number of bytes to write.
buf	in	Pointer to a buffer that holds the data to be written.
bytes_written	out	Actual number of bytes written.

- **VARIABLE_LENGTH records:**
  Each call to *OalWriteStream* causes a new record, with bytes_to_write as the LSB integer byte count prefix, to be written to the file; an extra byte is added to make the byte count even, if necessary (this extra byte is NOT included in the byte count prefix). On VMS this is done transparently. On other systems it is done explicitly by *OalWriteStream*.

- **FIXED_LENGTH records:**
  If *bytes_to_write* is less than RECORD_BYTES, then *OalWriteStream* writes additional bytes to pad the record out to RECORD_BYTES; if *bytes_to_write* is greater than RECORD_BYTES, then *OalWriteStream* writes multiple records, each RECORD_BYTES in size, and pads the last one out to RECORD_BYTES if necessary

- **STREAM or UNDEFINED records:**
  OalWriteStream writes *bytes_to_write* bytes with no modifications. The PDS recommended <CR><LF> line terminators should already be present in the buffer. The object layer routine *OaAddLineTerminatorstoTable* can be used to add these.

If the write is successful, a value of zero is returned.

7.2.4 OalSeek

This routine acts like the similar C library function *fseek*(), except that it also works for variable-length record files. For variable length records, the file is positioned to the first byte of the RCW associated with the specified record. When the stream descriptor's flags field has the OA_IS_SEEKABLE bit set, *OalSeek* may use *fseek*(); this is the default set by *OalOpenStream*. When it is not set, *OalSeek* will use *fread*() instead; this is useful when reading from a non-seekable device such as a pipe or socket. *OalSeek* can only be used to position in streams opened for read.

```
int OalSeek (struct OaStreamStruct *stream_id,
             long                  file_offset)
```

stream_id	in/out	Pointer to the stream descriptor for the file that is to be read.
file_offset	in	For stream and fixed length records, this is the byte position to which to position the file. For variable length records, this is the record at which to position. The count starts at zero.

If the seek is successful, a value of zero is returned. If the stream descriptor's file pointer was already at end-of-file when *OalSeek* was called, 1 is returned. If there was any other error, -1 is returned, and an error message is issued with *OaReportError*.

7.2.5 OalCloseStream

This routine closes a file and deallocates the stream descriptor associated with the file.

```
int OalCloseStream (struct OaStreamStruct *stream_id)
```

stream_id	in/out	Pointer to the stream descriptor for the file that is to be closed.

A value of zero is returned.

7.2.6 OalNewStreamDescriptor

This routine allocates a stream descriptor and initializes it to all zeroes.

```
struct OaStreamStruct *OalNewStreamDescriptor ()
```

A pointer to the stream descriptor is returned.

Chapter 8. Structure Layer

This chapter describes the structure layer of the OA Library. Where the stream layer discussed above treats the data associated with PDS data objects as a stream of bytes, the structure layer treats these data as a sequence of *atomic* data types: integer and real numbers, ASCII characters, etc. The bit patterns for representing atomic data can vary from one computer platform to another. For example, VAX floating point numbers differ in several significant ways from the floating point numbers used on a Sun workstation. Since PDS data can be encoded using the atomic data formats for any one of a number of platforms, it is often necessary to translate binary data read from a file into the atomic data formats appropriate to a user's computer. The structure layer performs this function. The structure layer can also translate data from ASCII to binary format or from binary to ASCII, and it can be used to skip over unwanted data in a file.

The key to the functioning of the structure layer is a data structure called a Stream Decomposition Tree (SDT). An SDT is an ODL tree that is augmented with some additional information to facilitate the transfer and translation of data. For the OA Library, each data object or collection of data objects that resides in a file or in memory has an ODL tree associated with it. The ODL tree is a static description of the data insofar as it specifies the attributes for the data object or objects at a point in time. When data are being transferred — either from file into memory or from one memory location to another — there are two points in time that matter: the instance before the transfer, when there must be an ODL tree describing the original, or *source*, data; and immediately after the transfer when there must be an ODL tree that describes the *destination* data. Whenever a transfer takes place, the structure layer creates a destination ODL tree from the source tree. The source and destination trees describing any given PDS object may differ for the following reasons:

- If the data are stored in the file in ASCII format, it may be desirable to convert the data to binary data at the destination so that the data object can be manipulated directly.

- If the data are stored in a binary file in a format different than what the host computer can handle, the atomic data types might need to be converted to the host's native data types.

- Some platforms prefer or require binary numeric data types to be aligned on word or double-word boundaries. If this is the case, the SDT specifies the proper alignment for the data at the destination.

- Only a portion of a source data object may be selected for transfer. For example, image and table objects can have prefixes and suffixes, but if the prefix or suffix aren't needed, they can be eliminated from the destination data to save space and simplify access.

To begin the process of transferring — and potentially translating data — from source to destination, an object layer routine calls the structure layer routine *OalCreateSDT* (and possibly *OalCompressSDT* as well) to create the SDT from the source ODL tree. To transfer data from a file to memory, buffers of data read from the source file using stream layer routines are provided to the *OalProcessSDT* routine which uses the SDT to translate the data as needed. At the end of the data transfer/translation process, the object layer routine calls *OalSDTtoODLTree* to clean up, eliminating the SDT information used only for data transfer and translation, and leaving an ODL tree that properly describes the destination data. SDTs are usually only used for reading in data from a file and doing in-memory conversions. Object layer routines that write an object to a file typically should not use an SDT, because it is better to first convert the object to the proper destination format in memory and then call the stream layer routines to write the object to a file.

The mechanisms for determining the size and format of a source PDS data object vary from object to object. For example, the size of an image is determined by a combination of several type-specific attributes (LINES, LINE_SAMPLES and SAMPLE_BITS). The *OalCreateSDT* routine and some of the routines that it calls contain code to convert the object-specific size and format information into object-independent parameters that are encoded in the SDT and used by the structure layer routines that actually transfer and translate data. The *OalSDTtoODLTree* routine then translates these object-independent parameters back into the proper attributes for each type of PDS data object. This means that if new types of objects are added to the PDS repertoire, some structure layer routines may need to be modified to accommodate the new objects.

Binary-to-binary translations are carried out using a mechanism called a Binary Representation, or *binrep*. A binrep is an abstract description of a numeric binary data type. There is a binrep for each of the numeric atomic data types for each platform supported by the OA Library. The structure layer uses the binreps for the source and the binreps for the destination to guide the translation process. This allows binary translations to be done by a single routine (most other schemes require from 2N to N^2 conversion routines for each atomic data type, where N is the number of platforms supported). A new platform can be supported by adding the binreps for its atomic data types, and updating the data types profile structure; no additional code is required.

The format of the destination data in a data transfer/translation operation is typically determined by a *host profile* data that is specific to the platform on which the software is running. This profile indicates the format of the atomic data types used by the host computer. The proper host profile is selected using a constant (which must be defined at compile time) which identifies the platform on which the software is running. By default the host profile specifies that the destination data are to be in binary format, even when the source data are encoded in ASCII. The host profile also defaults to aligning binary data properly for the platform. The interchange format and alignment of the destination data can be overridden by the user. Over-riding the interchange format can be used to leave ASCII data transferred from file in ASCII format in memory; over-riding the alignment might be desirable if saving memory space is particularly important.

8.1 Structure Layer Data Structures

The data structures used by the structure layer are created and maintained within the structure layer routines themselves. Users who operate upon PDS data objects with OA routines need know nothing about these structures, and writers of OA routines usually don't need to deal directly with them. The data structures are defined in the files *oal.h* and *binrep.h*.

8.1.1 Stream Decomposition Tree Node

This is the data structure that is attached to the nodes of an ODL tree to convert the tree into an SDT. The SDT node data structure is attached to the ODL tree node by putting a pointer to the SDT node in the *appl1* field of the ODL tree node.

```
typedef struct Stream_Decomposition_Tree_Node {
  long         total_repetitions;
  long         current_reps;
  OA_ATOM_INFO src;
  OA_ATOM_INFO dst;
  char         conversion_type;
  long         buf_bytes;
  char        *buf;
} SDT_node;

typedef SDT_node *SDTNODE;
```

total_repetitions	The number of times to loop through all of this node's children.
current_reps	Running count of repetitions currently done during stream processing.
src	The data structure that describes the source data. See the *oa_atom_info* data structure below.
dst	Same as above, but for the destination data.
conversion_type	Type of conversion to perform. Allowed values, as defined by the *oa_conversion_types_enum* enumeration type are: OA_NOT_APPLICABLE OA_BINREP OA_ASCII_TO_BINARY OA_BINARY_TO_ASCII OA_MEMCPY
buf_bytes	The number of bytes currently in the transfer buffer.
buf	Small internal buffer used to hold a partial atom of data when the atom is broken across a stream buffer.

The following data structure is used twice in each SDT node to describe the source and destination for a data transfer.

```
typedef struct oa_atom_info {
  long              start_offset;
  long              bytes_processed;
  long              size;
  PTR               ptr;
  char              PDS_data_type;
  struct binrep_desc *binrep_descrip;
  char              format_spec[15];
  char              alignment_req;
} OA_ATOM_INFO;
```

start_offset	The offset in bytes of the start of the first repetition of a data atom from its parent.
bytes_processed	Number of bytes of the atom which have been processed during the current repetition.
size	Size of a single repetition of the atom in bytes.
ptr	Pointer to the start of the current repetition of the SDT node's source or destination data. PTR is a typedef equivalent to char * on most platforms and char huge * on the IBM PC under MS-DOS.
binrep_descrip	Pointer to a data structure containing a binrep to guide binary-to-binary data conversion.
format_spec	The format specification used to direct binary-to-ASCII conversions.
alignment_req	The memory boundary on which the atomic data must be aligned, in bytes. Zero means not applicable. Other allowed values are 1 (byte aligned), 2, 4 and 8.

PDS_data_type	Type of atomic data value. The values are defined by the enumeration type *oa_PDS_data_types_enum* . There is a enumerated value for each of the data types and aliases from the PDS Standards Reference, Table 3.2. The values for aliased data types are set to the same value as the corresponding standard data type. OA_ASCII_REAL OA_ASCII_INTEGER OA_ASCII_COMPLEX OA_BIT_STRING OA_BOOLEAN OA_CHARACTER OA_COMPLEX OA_DATE OA_FLOAT OA_IBM_COMPLEX OA_IBM_INTEGER OA_IBM_REAL OA_IBM_UNSIGNED_INTEGER OA_IEEE_COMPLEX OA_IEEE_REAL OA_INTEGER OA_LSB_BIT_STRING OA_LSB_INTEGER OA_LSB_UNSIGNED_INTEGER OA_MAC_COMPLEX OA_MAC_INTEGER OA_MAC_REAL OA_MAC_UNSIGNED_INTEGER OA_MSB_BIT_STRING OA_MSB_INTEGER OA_MSB_UNSIGNED_INTEGER OA_PC_COMPLEX OA_PC_INTEGER OA_PC_REAL OA_PC_UNSIGNED_INTEGER OA_REAL OA_SUN_COMPLEX OA_SUN_INTEGER OA_SUN_REAL OA_SUN_UNSIGNED_INTEGER OA_TIME OA_UNSIGNED_INTEGER OA_VAX_BIT_STRING OA_VAX_COMPLEX OA_VAX_DOUBLE OA_VAX_INTEGER OA_VAX_REAL OA_VAX_UNSIGNED_INTEGER OA_VAXG_COMPLEX OA_VAXG_REAL OA_UNKNOWN_DATA_TYPE

8.1.2 Host Profile

The profile specifies how data are to be translated and aligned.

```
struct oa_profile {
  char dst_format_for_ASCII_src;
  char dst_format_for_binary_src;
  char dst_alignment_type;
  char data_translation_profile;
  char check_ASCII_writes;
};
```

dst_format_for_ASCII_src	Indicator of the interchange format for the destination data when the source data is ASCII. The choices are defined by the enumeration type *oa_interchange_format_enum*: OA_ASCII_INTERCHANGE_FORMAT OA_BINARY_INTERCHANGE_FORMAT
dst_format_for_binary_src	Indicator of the interchange format for the destination data when the source data is binary. The choices are defined by the enumeration type *oa_interchange_format_enum*, as above.
dst_alignment_type	Indicator of the alignment desired for the destination data. The choices are defined by the enumeration type *oa_alignment_type_enum*: OA_NOALIGN OA_ALIGN_EVEN OA_ALIGN_RISC
data_translation_profile	Indicator of the platform for which the destination data are to be formatted. The choices are defined by the enumeration type *oa_data_type_profiles_enum*: OA_ALPHA_OSF OA_ALPHA_VMS OA_IBM_PC OA_MAC_IEEE OA_SGI OA_SUN3 OA_SUN4 OA_ULTRIX OA_VAX
check_ASCII_writes	If TRUE, *OaWriteObject* checks that ASCII objects consist only of alpha-numeric characters and allowed control characters.

The value of *data_translation_profile* is used as an index into the data types profile, a variable *Oa_type_conversion_info*, which is an array of data structures giving the proper translations for each atomic data type on the specified platform. This includes information for binary-to-binary translations as well as the default field widths and formats for binary-to-ASCII translations. Enum values are the same for platforms with identical data types, for example, OA_ALPHA_OSF, OA_ULTRIX and OA_IBM_PC.

The writer of application code can change these values — although the need to do so is probably only when converting to or from ASCII — by directly modifying the global variable *Oa_profile*. Changes should be made prior to building the SDT (or calling an OA routine), and set back to the defaults afterwards. For example, if on a Sun 4, the default host profile is:

```
Oa_profile.dst_format_for_ASCII_src  = OA_BINARY_INTERCHANGE_FORMAT
Oa_profile.dst_format_for_binary_src = OA_BINARY_INTERCHANGE_FORMAT
Oa_profile.dst_alignment_type        = OA_ALIGN_RISC
Oa_profile.data_translation_profile  = OA_SUN4
Oa_profile.check_ASCII_writes        = TRUE
```

If a user wanted to convert a binary object to VAX data types prior to writing it to a file, she would change the profile as follows, then call *OaConvertObject*.

```
Oa_profile.dst_alignment_type        = OA_NOALIGN
Oa_profile.data_translation_profile  = OA_VAX
VAX_object = OaConvertObject( native_object);
```

The user can also change the individual data translation specifications in the data types profile *Oa_type_conversion_info*, but this is not recommended, except possibly to change the default field width or format in which an ASCII value is to be written during a binary-to-ASCII conversion.

8.2 Structure Layer Routines

The chief structure layer routines — those routines that are called directly from within OA routines — are fully specified below. Routines that are invoked by other structure layer routines — and thus hidden from the sight of developers of OA routines — are discussed only briefly. The names of all these routines begin with the three letters *Oal* to indicate that they are part of the inner workings of the OA Library.

8.2.1 OalCreateSDT

This routine creates an SDT from a source ODL tree.

```
ODLTREE OalCreateSDT (ODLTREE TLO_node,
                      int     src_interchange_fmt)
```

TLO_node	in/out	A pointer to a top-level object node in an ODL tree, for example, a Table or Image.

src_interchange_fmt	in	Indicator of the interchange format of the source data. The choices are defined by the enumeration type *oa_interchange_format_enum*: OA_ASCII_INTERCHANGE_FORMAT OA_BINARY_INTERCHANGE_FORMAT

This routine is called by OA routines to set up an SDT. Upon return, each node of the ODL tree that was passed as the first argument has an SDT node attached to it, with all the initial SDT parameters set.

For Image and Table type objects (as well as the object types derived from Table), the tree structure may be modified to accommodate prefixes and suffixes. When a table with prefix bytes is to be read in, a node representing the prefix table is added before the first column of the table node. Since prefixes are always stripped out, this node has *dst.size*=0. For an image with prefix/suffix bytes, an additional node is added below the image node. The image node specifies a loop through all the lines, and the new node below it specifies a loop through all the samples in a line. Nodes representing the prefix and suffix tables are added before and after the new node. Since the prefix and/or suffix bytes are always thrown away, these nodes have *dst.size*=0.

8.2.2 OalCompressSDT

This function improves performance by combining SDT nodes which have memory-to-memory transfer (using *memcpy*) as their conversion function. Several short transfers may be combined into one longer one, and the number of nodes in the SDT, as well as the overhead in structure layer processing, reduced.

```
ODLTREE OalCompressSDT (ODLTREE sdt)
```

sdt	in	A pointer to the root node of the SDT that is to be compressed.

The function copies the tree before compressing it. It returns the root node of the compressed SDT. The input SDT is unchanged.

8.2.3 OalInitializeSDT

This function finds the first data node to be used as the input for the first call to *OalProcessSDT*, and stores the *data_ptr* pointer in that node, as well as in the root node.

```
ODLTREE OalInitializeSDT (ODLTREE sdt,
                          PTR     data_ptr)
```

sdt	in/out	A pointer to the root node of the SDT.

data_ptr	in	A pointer into memory where *OalProcessSDT* will start placing the destination data. PTR is a typedef equivalent to char * on most platforms and char huge * on the IBM PC under MS-DOS.

8.2.4 OalProcessSDT

This routine controls the actual transfer and translation of data. It traverses the SDT, calling the transfer/translation routines at each SDT node, until all the data in the buffer has been processed.

```
int OalProcessSDT  (PTR     source,
                    long    source_bytes,
                    ODLTREE *current_node)
```

source	in	A pointer to the source data. PTR is a typedef equivalent to char * on most platforms and char huge * on the IBM PC under MS-DOS.
source_bytes	in	Number of bytes in the source data.
current_node	in/out	Pointer to a pointer to the current data node in an SDT.

When reading from files, the source is an I/O buffer full of data and the destination(s) are in memory. For in-memory conversions, both the source and destination are in memory. When writing to files the source is in memory and the destination is an I/O buffer. This latter option might, for example, be used to convert a row of a binary table to ASCII, which *OalWriteStream* could then write to a file.

OalProcessSDT always processes the number of bytes specified for the source unless it reaches the end of the SDT while doing so. Care should be taken that the output buffer doesn't overflow. This is usually done by allocating the output buffer to be the entire size of the object being processed by *OalProcessSDT*.

Upon exit the current_node parameter points to the ODL tree node where the processing of the next piece of the source data begins. This may be the same as the previous current node or it may be a different node.

8.2.5 OalSDTtoODLTree

This routine strips off the SDT nodes associated with the ODL tree nodes and updates the ODL tree nodes' keywords to match the destination data instead of the source data described by the original ODL tree nodes.

```
int OalSDTtoODLTree (ODLTREE sdt
                     int     dst_interchange_format)
```

sdt	in/out	A pointer to the root node of an SDT.

dst_interchange_format	in	Indicator of the interchange format of the destination data. The choices are defined by the enumeration type *oa_interchange_format_enum*: OA_ASCII_INTERCHANGE_FORMAT OA_BINARY_INTERCHANGE_FORMAT

8.2.6 OalFreeSDT

This routine frees the ODL tree whose root is passed in, including freeing the SDT nodes which are attached to the ODL tree nodes.

```
int OalFreeSDT (ODLTREE  sdt)
```

sdt	in/out	A pointer to the root node of an SDT.

The function always returns zero.

8.2.7 OalReportBinrepErrors

This routine calls *OaReportError* with *oa_errno*=904 if any of the binrep error counters are non-zero: integer truncation errors, negative-to-unsigned integer conversions, exponent underflows and overflows, and precision losses. The error message reports the number of errors of each type which have occurred since the counters were last set to zeros by *OalResetBinrepErrors*.

```
int OalReportBinrepErrors (char *proc_name)
```

proc_name	in	If non-NULL, then the error message will be prepended by proc_name to give the user an indication of where the errors occurred.

8.2.8 OalResetBinrepErrors

This routine sets all the binrep error counters to zeros. It can be called by object layer routines before a conversion; after the conversion, a call to *OalReportBinrepErrors* shows if any errors occurred during the conversion.

```
int OalResetBinrepErrors ()
```

8.2.9 Low-Level Structure Layer Routines

The following routines are called directly or indirectly by the top-level structure layer routines discussed above. They are briefly described here, but since they are not called by the developers of object layer routines, their calling sequences are not detailed.

8.2.9.1 OalPostOrderTraverse

This routine does a post-order traversal of an SDT tree, and at each node it calls a function specified as an input parameter. If the called function returns a non-zero value, *OalPostOrderTraverse* aborts the traversal and returns the value. The root node is the last node to be visited. This routine is called by *OalCreateSDT* and *OalSDTtoODLTree*.

8.2.9.2 OalBuildSDTNode

This routine initializes all the parameters of a single SDT node. It identifies two types of nodes: data nodes, which describe an atomic data type and it's conversion to another atomic data type, and repetition nodes, which indicate how many passes through the repetition node's children to make. For data nodes, the offset, size and data type of both the source and destination data are stored in the SDT node attached to the ODL tree node. This is later used by *OalConvert* to convert the data atom of source data into destination data. For repetitions nodes, the repetitions count used by *OalPositionToNextDataNode* is initialized. This function is called by *OalCreateSDT* from within *OalPostOrderTraverse*.

8.2.9.3 OalDetermineConvertParameters

This routine determines the SDT destination parameters for data nodes: destination data type, size, alignment, conversion type and binrep description. If the conversion_type is BINARY_TO_ASCII, then it also determines the format specification to use in conversion, and stores it in the SDT node's *dst.format_spec* field. This function is called by *OalBuildSDTNode*.

8.2.9.4 OalGetTypeConversionFromProfile

This routine gets the profile record for a particular data type conversion from the host profile. The profile record specifies a binary and an ASCII type to convert to. This routine is called by *OalDetermineConversionParameters.*

8.2.9.5 OalFindBinrepDescrip

This routine gets the proper binrep for a given binary data type from an array of binreps. This routine is called by *OalDetermineConversionParameters*.

8.2.9.6 OalPositionToNextDataNode

This function positions to the next SDT node during transfer/translation process, and updates the source and destination data pointers in the SDT node. These point to the current memory locations in the source and destination data. A modified post-order tree traversal order is used; a node with children has a repetition count associated with it, which is decremented each time a complete loop through the children nodes is finished. This function is called by *OalProcessSDT*.

8.2.9.7 OalConvert

This routine performs the data conversion specified in an SDT node. The conversions are ASCII-to-binary, binary-to-ASCII, binary-to-binary and straight copy. This function is called by *OalProcessSDT*.

8.2.9.8 OalGetNativeCTypeInfo

This routine gets the PDS enumerated data type, size and binrep description which map to the input C type string on the platform being run on. For example, if the input C type string is "int" and OAL is running on a Sun, the routine returns MSB_INTEGER, 4, and a pointer to the MSB_INT4 binrep structure. This routine is called during initialization by *OalConvert.*

8.2.9.9 OalBinrepConvert

This routine does a single binary-to-binary conversion. It converts a binary numeric type to a different binary numeric type and stores the result. The number of bytes in the source and destination and a specification of their types are given in the source and destination binrep description structures. This routine is called by *OalConvert*.

8.2.9.10 OalAdjustKwdstoMatchSDT

This routine updates an ODL tree node's keywords to match its SDT node. It is called after SDT processing to make the ODL tree's keyword values, which describe the source data, now describe the converted destination data. This function is called by *OalSDTtoODLTree*.

Chapter 9. Developing New Object Access Routines

9.1 Understanding Object Access Routines

The code of an OA read routine tends to follow a predictable pattern (which means that new OA routines can be created quickly from existing routines). Other types of routines in the object layer which manipulate in-memory data, as opposed to reading the data from a file, use many of the same structures and logic, so much of this discussion is also applicable to them.

Figure 2 below shows a schematic of the routine *OaReadTable,* which reads table objects from a file. Like all OA routines for reading PDS data objects from a file, the input to this routine is a pointer to the node of an ODL tree that describes the table (1). *OaReadTable* first calls the object layer utility routine *OaGetFileKeywords* to get from the ODL tree information about the file that contains the table (2). It then calls the object layer utility routine *OaCheckODLTree* to make sure that the ODL tree is organized properly and can be handled by the OA Library software (3).

OaReadTable then calls the structure layer routine *OalCreateSDT* to turn the ODL tree for the table object into an SDT that will guide the process of transferring the table to memory (4). *OalCreateSDT* checks to determine the data formats for the source data in the file and determines from the host profile the proper data formats for the table in memory. After the SDT is built, the structure layer routine *OalCompressSDT* can be called to simplify the SDT if possible, thus speeding the data transfer process (5). As the final step in setting up for data transfer, the stream layer routine*OalOpenStream* is called to initialize the input stream from the file (6).

OaReadTable now begins the actual data transfer process by making repeated calls to *OalReadStream* to read buffers of data and *OalProcessSDT* to translate the table data as needed and place it into the proper location in memory (7 and 8).

Upon completion of the data transfer, the structure layer routine *OalSDTtoODLTree* is called to get rid of the SDT nodes and to leave an ODL tree that describes the resultant data table in memory (9). An object descriptor is then built to point to both the table object data and the table's associated ODL tree. The object descriptor is returned as the function value of *OaReadTable* (10).

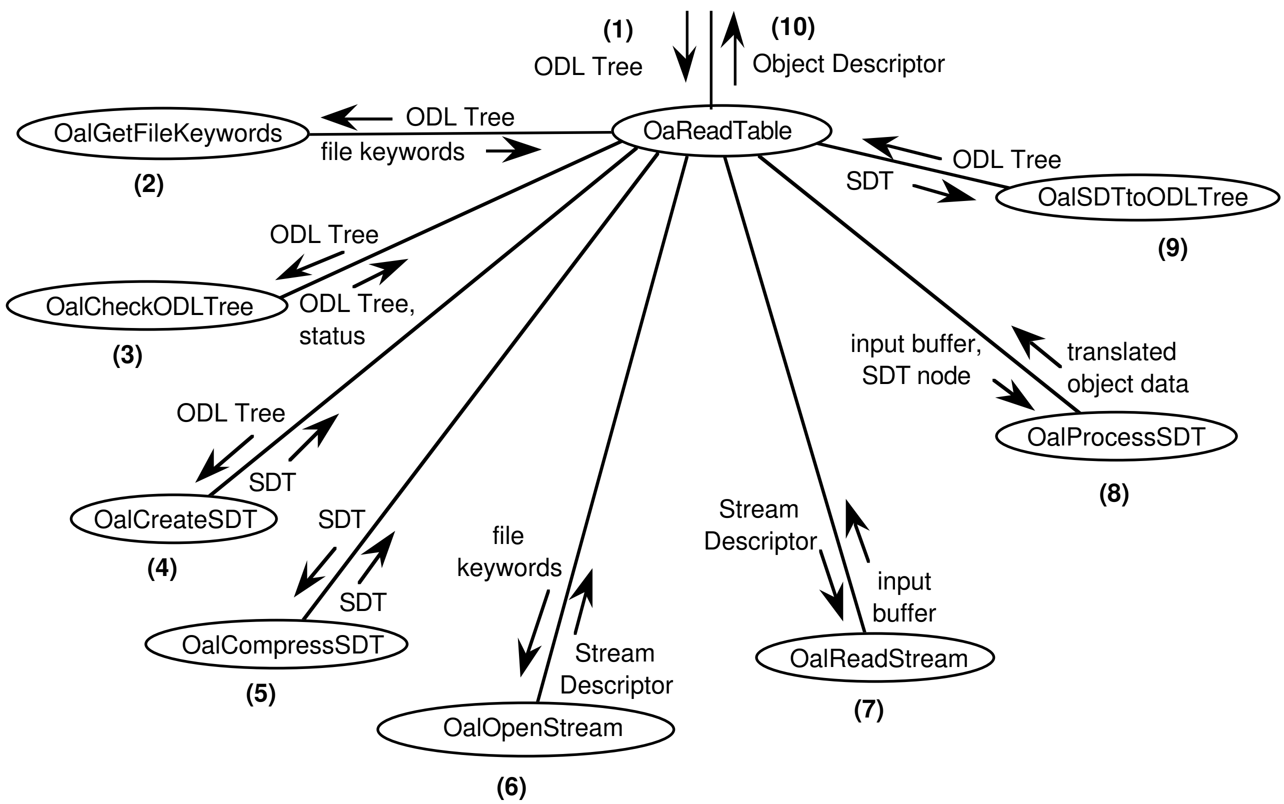

Figure 2 — Process Flow for the Object Layer Routine *OaReadTable*

Additional steps which apply to all OA routines:

- Check all input parameters for validity.

- Always make a copy of the input ODL tree before changing it or calling *OalCreateSDT*, otherwise the caller's tree will be changed.

- Initialize the SDT with a pointer which points to the start if the destination memory, usually space you've allocated with *OaMalloc*.

- In the first call to *OalReadStream*, specify the *file_offset* to start reading at; this is the value returned by *OaGetFileKeywords*, or one you've calculated yourself.

- After calling *OalSDTtoODLTree*, be sure to delete or modify the values of keywords which were copied from the input tree but are no longer applicable to the output tree. E.g. is CHECKSUM still valid?

Other OA routines manipulate objects in memory, instead of reading an object from a file. These routines generally don't need the stream layer, but may still use an SDT to transfer, filter and/or convert data.

9.2 Techniques For Creating the SDT

The SDT data structure is the key to describing the locations and sizes of the source data atoms (in a file or in memory), getting your data converted to the right format, and filtering out unwanted data. Once you've created an SDT, the structure layer routine *OalProcessSDT* processes the input stream, handling all the positioning, conversions and filtering you've specified.

If you are implementing an OA routine for an existing object type, then you can use *OalCreateSDT* to to most of the work for you. Even if you're implementing a routine for a new object type *OalCreateSDT* doesn't know about, you can still use it - you should avoid creating an SDT from scratch.

If you're writing an OA routine to handle an object type the structure layer doesn't know about, consider transforming your ODL tree or creating a new one which looks like one of the supported object types - the same data can be represented by many different PDS objects. For a simple example of this, look at the code for *OaReadSpectrumFromQube* in *obj_l2.c* - this routine reads a Spectrum from a Qube without the structure layer knowing anything about Qubes or Qube-specific keywords. It does this by transforming the Qube tree into a Spectrum-type tree, whose single column node represents each core pixel in the Qube. The rest of the data in the Qube is not explicitly described, but is encompassed by the ROW_BYTES and ROWS keywords in the Spectrum node. The call to *OalCreateSDT* detects implicit SPARES before and after the column, and adds explicit SPARE nodes, and sets up their SDT nodes to throw out this data.
In this case the ODL tree created to input into the SDT has the same class and structure as the output ODL tree, but you could make a different ODL tree to describe the output data than the one you used to create the SDT with.

9.3 Requirements for New Object Access Routines

As new routines are added to the OA Library, certain conventions should be followed to make the new routines easy to use for others.

In general, routines that read data objects from file should do only that: they shouldn't manipulate the data, except possibly to reformat the data for easier access in memory. An example of this kind of reformatting is the decompression done by *OaReadImage*, which decompresses a compressed image during the read operation.

An OA routine that manipulates data objects should take as its input arguments one or more object descriptors for objects that are already in memory. The routine may also have zero, one or more arguments that are of a standard C data type. The routine will return either an object descriptor or a data value of a standard C type. An example of such a manipulation routine might be an OA routine that takes in an Image object and a Histogram object and that returns an Image object that has been histogram-equalized to enhance its contrast. In general any PDS data objects that are input to an OA routine should be left undisturbed by the routine.

The routines that form the OA Library's object layer meet the following general requirements. User-supplied OA routines are strongly encouraged to also adhere to these requirements.
These requirements assume that the language of implementation is C.

[9.3.1] Each OA routine shall have a name that conforms to the PDS OA software naming convention:

- An OA routine name begins with the letters *Oa*, with the first letter only capitalized.
- After the beginning letters, the name includes a word or words that specify the operation that is to be performed. The first letter only of each word is capitalized. Examples are: *Read* and *Convert.*
- At the end of the name is a word or words that identifies the type of object — or part of an object — that is operated upon by the routine. The first letter only of each of these words is capitalized. For example, *Image*, *CompressedImage*, and *CompressedImageLine.*
- All of the parts of the name are provided without underscores or other intervening punctuation. For example, *OaReadImage*.

[9.3.2] Any OA routine that can access more than one type of PDS data object shall signify this by setting the type name in the routine to be *Object*. For example, *OaCopyObject*.

[9.3.3] An OA routine for a specific type of object (for example Image objects) shall be capable of handling all objects of that type, although the operation performed by the routine may not be appropriate for some data objects (for example, a routine for decompressing an image must be able to accept an uncompressed image but won't do anything with it).

[9.3.4] An OA routine that reads an object, or part of an object, from a PDS-labeled file shall be designated by having the word *Read* in its name. For example, a routine to read a Histogram from file would be named *OaReadHistogram.*

- The parameter list to an OA routine that reads an object from file shall include as input an ODL tree for the file, with a pointer to the node of the ODL tree that identifies the object to be read from the file.
- The OA routine shall use OA stream layer routines to open and read from the file whenever feasible. An example of an exception to this is the Clementine JPEG decompression software - *OaReadImage* uses the stream layer to open and position the file pointer to the start of the compressed data, but then passes the file pointer to custom decompression software. This software then takes over reading the file using it's own I/O.
- If an OA routine reads all of an object it shall call the appropriate stream layer routine to close the stream to the file before returning.

[9.3.5] An OA routine that writes a PDS data object to a file shall have the word *Write* in its name.

[9.3.6] An OA routine that converts a non-PDS data object into a PDS data object shall have the word *Import* in its name. For example, a routine to take a standard C array and make it a PDS Image object would have the name *OaImportImage*.

[9.3.7] An OA routine that converts a PDS data object into a non-PDS data object shall have the word *Export* in its name. For example, a routine to take a PDS image object and make it into a standard C array would have the name *OaExportImage*.

[9.3.8] An OA routine shall have a fixed number of parameters. Each parameter may be either a pointer to a node of an ODL tree, a pointer to a PDS data object or a value that belongs to a standard C language data type (or a pointer to such a value).

[9.3.9] All PDS data objects shall be passed to OA routines in a consistent manner using an *object descriptor*.

[9.3.10] If an OA routine reads from or writes to a file, then an object descriptor for the file shall be the first parameter of the OA routine.

[9.3.11] If an existing data object, or set of data objects, of the type specified in the routine name are to be manipulated by an OA routine, then those data objects shall be the first parameters in the calling sequence, with the exception of cases where the objects are being read from or written to files (see Requirement 9.3.6 above).

[9.3.12] An OA routine shall return a single value that is either a pointer to an object descriptor or a value belonging to a standard C data type (or a pointer to such a value).

[9.3.13] An OA routine that creates a PDS data object shall create all of the required keywords (attributes) of the data object, although the keyword need not have values initially assigned. The user may need to supply keyword values either through other OA routines or using L3 routines.

[9.3.14] An OA routine that manipulates and changes a PDS data object must also update any of the keywords that are affected by the change, and delete any keywords which have been rendered invalid by the change. For example, a routine which decompresses an image should remove the ENCODING_TYPE keyword, or set it to "N/A".

[9.3.15] An OA routine that copies a data object shall copy all of its keyword values, along with the data associated with the object.

[9.3.16] A routine that deletes a data object shall delete both the data associated with the object, all of the attributes of the data object (the ODL tree), and the associated object descriptor.

[9.3.17] An OA routine may have all or only part of a data object in memory at a time. It shall be possible for a user to determine which portion of an object is in memory at any given time (for example, which lines of a large image are currently in memory). This should be done by setting keywords in the object's ODL tree.

[9.3.18] OA routines shall adhere to the standard scheme for reporting processing status and errors that is described in the *User's Guide*.

[9.3.19] OA routines shall adhere to a standard scheme for routing error messages to users as described in the *User's Guide*. The user shall be able to select the destination of error messages (i.e., to the screen, to a file, etc.).

Appendices

Appendix A. Error Codes

This appendix lists the error codes currently returned by OA routines in global variable *oa_errno*.

The error codes are arranged in categories to facilitate the future addition of new codes. Each category has a range of values allotted to it. The currently used values for each category are listed, along with descriptions of the errors which are similar in text to the message issued by *OaReportError*. Filtering of error messages inside *OaReportError* can be done by category (check if *oa_errno* is within a range of values) or by individual error code (check if *oa_errno* is equal to a certain value).

When new error messages are added to the OA Library, existing error codes should be used for errors similar to those already existing. Any new error codes should be added within the appropriate category.

Note that these codes have a dual purpose: filtering of messages, since *oa_errno* is set before every call to *OaReportError*, and testing the severity of erroneous function returns (which are almost always accompanied by a call to *OaReportError*).

A.1 Bad Input Arguments

This category uses *oa_errno* values in the range 501 - 509.

501

- Input ODL tree node, OA_OBJECT, OA_OBJECT's ODL tree or data pointer is NULL.
- Input string (char *) is NULL.

502

- Invalid input row number, start_line, stop_line, start_sample, stop_sample, read_write_mode, record_bytes, bytes_to_write.

A.2 Inconsistent Input Object

This category uses *oa_errno* values in the range 510 - 519.

510

- Invalid combination of is_in_memory, data_ptr and size in OA_OBJECT.

511

- LINES * LINE_SAMPLES * SAMPLE_BITS/8 not equal to input OA_OBJECT's size.
- (PREFIX_BYTES + ROW_BYTES + SUFFIX_BYTES) * ROWS not equal to input OA_OBJECT's size.

A.2 Illegal Operation

This category uses *oa_errno* values in the range 520 - 529.

520

- Attempt to delete the only sub-object in a table, attempt to delete the only row in a table.
- Binary INTERCHANGE_FORMAT objects not allowed in STREAM output files.
- Detected illegal control character in data for output to ASCII file.
- Data buffer for STREAM output didn't have \n terminator.
- Attempt to write object to output file which already has an object with the same name.
- Attempt to seek backwards when stream_id->flags OA_IS_SEEKABLE isn't set.
- Input tables must have same INTERCHANGE_FORMAT, ROW_BYTES, COLUMNS or ROWS.
- Improper RECORD_TYPE, ENCODING_TYPE, SAMPLE_BITS or TABLE_STORAGE_TYPE for operation.
- Image cannot have LINE_PREFIX_BYTES or LINE_SUFFIX_BYTES for operation.
- Table cannot be COLUMN_MAJOR for operation.
- Image is multi-banded (not supported yet).
- Can't perform operation with a variable-length record file.
- Can't handle a BIT_ELEMENT or BIT_COLUMN node directly under an ARRAY node.
- Can't convert a float of this size.
- HFD compressed images supported only in variable-length record files.
- Attempt to open file for write which has RECORD_TYPE of OA_UNKNOWN_RECORD_TYPE or OA_UNDEFINED_RECORD_TYPE.
- Attempt to open fixed-length file for write with odd record length.
- Attempt to read an object with unknown class.

A.3 Bad Input ODL Tree, Keywords or Class for Operation

This category uses *oa_errno* values in the range 530 - 549.

530

- Input ODL tree node or input OA_OBJECT's ODL tree has improper object class for operation.
- Input node's parent must be a Table, the root of the ODL tree or the child of the root of the ODL tree.
- Input Table node has no sub-objects.
- Input Table must have Container as only sub-object.
- Child_count field not the same as COLUMNS keyword value.
- File_name field in ODL tree node is NULL.

- Collection must be nested inside an ARRAY or Collection.
- COLUMN_MAJOR table with nested Container or ROW_PREFIX_BYTES or ROW_SUFFIX_BYTES not allowed.
- Couldn't find top-level object.
- Couldn't find pointer to top-level object.
- Given object class doesn't have required parent and/or child node(s) or class.
- Couldn't find encoding histogram for HFD compressed image.

531
- Required keyword missing.
- Invalid or unknown keyword value for INTERCHANGE_FORMAT, TABLE_STORAGE_TYPE, FILE_TYPE, RECORD_TYPE, DATA_TYPE, COMPRESSION_TYPE.
- Keyword required for operation but not found.

532
- Miscellaneous label errors: two sequences which should have the same number of items don't, unknown PDS version, PDS_VERSION_ID doesn't match label contents, number of sequence items in AXIS_ITEMS not equal to AXIS keyword value.

533
- Illegal number of bytes for INTEGER or REAL type.

534
- Overlapping source data (structure layer). This is usually due to an error in the label.

535
- Invalid SDT contents (src.PDS_data_type, conversion_type field, Oa_profile field(s).

A.4 Inconsistency Between File and File Keywords

This category uses *oa_errno* values in the range 550 - 559.

550
- Actual record type doesn't match RECORD_TYPE keyword value.
- Actual file size doesn't equal FILE_RECORDS * RECORD_BYTES.

A.5 Syntax Errors in ODL Keyword Values

This category uses *oa_errno* values in the range 600 - 609.

600
- Error converting keyword value to long.
- Error converting keyword long to keyword value.
- Error converting sequence value to long.
- Error parsing ODL sequence.

601

- Error parsing FORMAT keyword value.

A.6 Low-level I/O Errors

This category uses *oa_errno* values in the range 700 - 709.

700

- fread, read, write, or rewind error.
- fopen error opening object file, label file or tmp file for output.
- fstat on HISTORY tmp file returned error.
- fread failed to read HISTORY tmp file.
- Invalid current variable-length record size read.

A.7 Run-time Errors

This category uses *oa_errno* values in the range 710 - 719.

710

- Image position and file position are out-of-sync.
- First byte in Previous Pixel compressed image wasn't 255.
- SDT source size not equal to HFD decompressed line buffer size.

A.8 Memory Errors

This category uses *oa_errno* values in the range 720 - 729.

720

- *OaMalloc* or *OaRealloc* failed.

A.9 Internal Errors

This category uses *oa_errno* values in the range 730 - 739.

730

- Couldn't find binrep_q_code.
- Oa_type_conversion_info array index out-of-range.
- Oa_profile.dst_alignment_type has unknown value.

A.10 Warnings

This category uses *oa_errno* values in the range 900 - 949.

900

- Implicit SPARE detected.

901

- File size not equal to FILE_RECORDS * RECORD_BYTES for fixed-length file.

902

- Read 0 bytes (*OaReadStream*).

903

- PDS version number not PDS3.

904

- *OalReportBinrepErrors* sets *oa_errno* to 904 when it calls *OaReportError*.

A.11 Informational

This category uses *oa_errno* values in the range 950 - 999.

950

- *OaReportFileAttributes*.
- Opened file after upper-casing file name or after appending ";1".
- VMS record type/length message.
- Converting ODL tree to version 3.
- Debug messages.

Appendix B. How OAL Deals with Spares

An explanation of SPARES and usage guidelines for them are found in the PDS Standards Document from A-97 to A-100.

- OAL treats SPARES the same as prefixes and suffixes: they are removed when reading from a file or doing an in-memory transfer.
- OAL may add SPARES to binary tables for alignment purposes, just as compilers add alignment spares in structures.
- OAL treats implicit and explicit SPARES the same.
- *Binary-to-ASCII Conversions:* SPARES are always removed.
- *ASCII-to-Binary Conversions:* SPARES may be added to align the data in memory on platforms which require alignment, or they may be removed on platforms which don't require alignment.
- *Binary-to-Binary Conversions:* SPARES are added or removed as needed to align the data in memory.
- *ASCII-to-ASCII Conversions:* SPARES are always removed; this means double quote delimitors for CHARACTER columns, commas, and <CR><LF> line delimitors are always removed. Users should manipulate ASCII tables in memory with these removed, and add them only when preparing to write the data to an output file, if their presence is desired in the output file.

Appendix C. Transferring PDS Files Between Computers

When transfering files between different host systems, use ASCII (text) mode for files which contain only ASCII data (PDS label files, ASCII Tables, etc), and use binary (image) mode for files which contain binary data objects.

The following are some of the discrepancies which can arise when a PDS file is transferred from one operating environment to another (e.g. from VAX to Unix), resulting in the file no longer having the same record format specified in its label, or having otherwise altered contents.

- When transfering an ASCII file to a Mac in ASCII mode, the PDS recommended <CR><LF> (carriage-return and line-feed) characters at the end of each line may be converted to <CR><CR>. Similarily, when transfering a file from a Mac, <CR><LF> (or <CR><CR>) may be converted to <LF><LF>. The OA stream layer handles this situation.

- When transfering a PDS file from UNIX to VMS in ASCII mode, the <LF>s are stripped out and made part of the file's VMS record attributes, and an extra pad byte may be added to make each record an even number of bytes. For FIXED_LENGTH files, this may make the RECORD_BYTES keyword value invalid.

- A similar situation occurs when the ASCII file is transferred from VMS to UNIX in binary (image) mode (which is not recommended - use ASCII mode). The <LF>'s hidden in the VMS record attributes are not transferred, and the resulting actual 'record' length is one less than the RECORD_BYTES keyword indicates. The OA stream layer handles both these situations.

- Files transfered to a VAX with binary mode end up as fixed-length, 512-byte record files.

- Under VMS, a file with RECORD_TYPE keyword value of VARIABLE_LENGTH may be a real VMS variable-length record file, or it may be a fixed-length 512 byte record file which was transferred from another machine, with embedded 2-byte record length integers for each variable-length record. The OA stream layer detects and handles this situation.

Appendix D. Example Code

The same example program is given below in C, IDL and Fortran. This source code is also provided in the release in the files *t_images.c*, *t_images.pro*, and *t_images.f*. The program reads an Image and a Histogram from a PDS data file (from a Voyager CD-ROM), calculates the histogram of the Image, then compares the calculated histogram to the histogram read from the data file - the two should be identical. Another example program provided in the release (*t_tables.c*, *t_tables.pro*, *t_tables.f*) demonstrates various Table functions.

D.1 C Example

This is one of the C examples supplied with the OA library distribution.

```
/* This program reads the IMAGE and IMAGE_HISTOGRAM objects from the
   attached label file V1.LBL.  It then calculates the histogram of the
   image, and compares it with the IMAGE_HISTOGRAM read from the file -
   the two should be identical.  The file V1.LBL must be present in the
   default directory the program is run from.
*/

#include "oal.h"
#include <string.h>
#include <stdio.h>

main() {

#ifdef IBM_PC
typedef long int4;
#else
typedef int int4;
#endif

OA_OBJECT oa_object, image_object1, image_object2, histogram_object;
OA_OBJECT image_handle;
ODLTREE root_node, odltreenode, image_node, histogram_node;
KEYWORD *kwdptr;
char *label_filename, *errfilespec, fmt[80];
PTR image_values;
int4 histogram_values[256];
int4 *image_histogram_values;
int i, j, lines_per_call, err, lines;
long line, sample;
unsigned char u;

/* Initialize histogram buffer to 0's.  */
for (i=0; i<256; i++)
  histogram_values[i] = 0;

/* Read in the label from V1.LBL.  This is an attached label with
   variable-length records, containing a HFD compressed Voyager image of
   miranda, an engineering table and two histograms.  OaParseLabelFile
```

```
   uses OA's Stream Layer to read the variable-length records, and L3's
   OdlParseLabelString to create the ODL tree.  */

label_filename = "V1.LBL";
root_node =  OaParseLabelFile( label_filename, NULL,
                               ODL_EXPAND_STRUCTURE,
                               TRUE);
if (root_node == NULL) {
  OaReportError( "Error from OaParseLabelFile!");
  return(0);
}

/* Find the IMAGE node in the ODL tree.  */

if ((image_node = OdlFindObjDesc( root_node, "IMAGE", NULL, NULL, 0,
                                  ODL_RECURSIVE_DOWN)) == NULL) {
  OaReportError( "Couldn't find IMAGE node!");
  return(0);
}

/* Calculate the histogram from the image; use OaReadPartialImage to
   read in the image piece by piece, lines_per_call lines at a time.
   This uses less memory than OaReadImage (or OaReadObject), because
   only a small portion of the image is in memory at a time.  */

lines_per_call = 200;

if ((image_handle = OaOpenImage( image_node)) == NULL) {
  OaReportError( "Error from OaOpenImage!");
  return(0);
}

sprintf( error_string,
         "Calculating histogram; %d calls to OaReadPartialImage...",
         800/lines_per_call);
oa_errno = 950;  /* Error code for Informational messages. */
OaReportError( error_string);

for (i=0; i<(800/lines_per_call); i++) {
    image_object1 = OaReadPartialImage(
                      image_handle,
                      (long) (i*lines_per_call+1),
                      (long) (i*lines_per_call+lines_per_call),
                      (long) 1, (long) 800));
    if (image_object1 == NULL) {
    OaReportError( "Error from OaReadPartialImage!");
    return(0);
  }
  image_values = image_object1->data_ptr;

  for (line=0; line<lines_per_call; line++) {
    for (sample=0; sample<800; sample++) {
      u = (unsigned char) image_values[ line*800 + sample];
      histogram_values[u] += 1;
    }
  }
  OaDeleteObject( image_object1);
}
```

```
OaCloseImage( image_handle);


/* Find the IMAGE_HISTOGRAM node in the ODL tree.  */

histogram_node = OdlFindObjDesc( root_node, "IMAGE_HISTOGRAM", NULL,
                                 NULL, 0, ODL_RECURSIVE_DOWN);
if (histogram_node == NULL) {
  OaReportError( "Couldn't find IMAGE_HISTOGRAM node!");
  return(0);
}

/* Read the IMAGE_HISTOGRAM into memory.  */

histogram_object = OaReadHistogram( histogram_node);
if (histogram_object == NULL) {
  OaReportError( "Error from OaReadHistogram!");
  return(0);
}

/* Strip the HISTOGRAM object of everything but the object data.  */

image_histogram_values = (int4 *) OaExportObject( histogram_object);

/* Print out a few values to see if it worked.  */

oa_errno = 950;  /* Error code for Informational messages. */
OaReportError(
  "First 5 values should be:  529   182   139   284   315");
#ifdef IBM_PC
strcpy( fmt, "First 5 values are:       %ld  %ld  %ld  %ld  %ld\n");
#else
strcpy( fmt, "First 5 values are:        %d   %d   %d   %d   %d\n");
#endif
sprintf( error_string, fmt, image_histogram_values[0],
         image_histogram_values[1],  image_histogram_values[2],
         image_histogram_values[3],  image_histogram_values[4]);
OaReportError( error_string);

/* Compare the histogram we calculated from the image with the histogram
   read from the file.  */

for (i=0; i<256; i++)
  if (histogram_values[i] != image_histogram_values[i]) {
    sprintf( error_string,
             "Error: histograms do not match; i = %d\n", i);
    OaReportError( error_string);
    return(0);
  }
OaReportError( "histograms match");

OaFree( (PTR) image_histogram_values);

return(0);
}
```

D.2 IDL Example

This is one of the IDL examples supplied with the OA library distribution.

```
;*********************************************************************
; This procedure reads the IMAGE and IMAGE_HISTOGRAM objects from the
; attached label file V1.LBL, calculates the histogram of the image,
; and compares it with the IMAGE_HISTOGRAM read from the file - the two
; should be identical.
; The file V1.LBL must be present in directory specified by LABEL_DIR
; (edit first section of code below to match your directory).
;
; Platforms and compilers this has been tested on are:
; 1) SGI/Irix
; 2) Sun SPARCstation/Solaris
; 3) DecAlpha/OSF
; 4) PowerMac, Metrowerks CodeWarrier
;
;*********************************************************************

pro t_images, IMAGE, HIST

case 1 of
  (!VERSION.arch eq 'mipseb'): begin
    LABEL_DIR = '/miranda2/monk/oal/lbls/'   ;SGI
  end
  (!VERSION.arch eq 'sparc'): begin
    LABEL_DIR = '/alcmene2/monk/oal/lbls/'   ;Sun SPARCstation
  end
  (!VERSION.arch eq 'alpha' and !VERSION.os eq 'OSF'): begin
    LABEL_DIR = '/usr/users/monks/oal/lbls/' ;Dec Alpha/OSF
  end
  else: begin
    LABEL_DIR = 'MacT06-2:OAL:'              ;Macintosh
  end
endcase

Z = OaRouteErrorMessages( LABEL_DIR + 'OAL_ERRORS.TXT', 0)

ROOT_NODE = OaParseLabelFile( LABEL_DIR + 'V1.LBL', $
                              LABEL_DIR + 'ODL_ERRORS.TXT', 1, 0)
if ROOT_NODE eq 0 then begin
  print,'OaParseLabelFile returned NULL!'
  return
endif

IMAGE_NODE = OdlFindObjDesc( ROOT_NODE, 'IMAGE', '', '', 0, 0)
if IMAGE_NODE eq 0 then begin
  print,'OdlFindObjDesc could not find IMAGE node!'
  return
endif

IMAGE_OBJECT = OaReadObject( IMAGE_NODE)
if IMAGE_OBJECT eq 0 then begin
```

```
  print,'OaReadObject returned NULL!'
  return
endif

IMAGE = OaIDLGetObjectData( IMAGE_OBJECT)

HISTOGRAM_NODE = OdlFindObjDesc( ROOT_NODE, 'IMAGE_HISTOGRAM', '', '', $
                                 0, 0)
if HISTOGRAM_NODE eq 0 then begin
  print,'OdlFindObjDesc could not find IMAGE_HISTOGRAM node!'
  return
endif

HISTOGRAM_OBJECT = OaReadObject( HISTOGRAM_NODE)
if HISTOGRAM_OBJECT eq 0 then begin
  print,'OaReadObject returned NULL!'
  return
endif

HIST = OaIDLGetObjectData( HISTOGRAM_OBJECT)

print,'First 5 values should be:   529   182   139   284   315'
print,'First 5 value are:       ' + string( format='(5(I6))',HIST(0:4))

W = where( HIST ne histogram( IMAGE), COUNT)
if COUNT gt 0 then begin
  print,'Error: histograms do not match; ', COUNT, ' differences.'
endif else begin
  print,'histograms match'
endelse
return
end
```

D.3 Fortran Example

This is one of the Fortran examples supplied with the OA library distribution.

```
C This program reads the IMAGE and IMAGE_HISTOGRAM object from the
C attached label file V1.LBL, calculates the histogram of the image,
C and compares it with the IMAGE_HISTOGRAM read from the file - the two
C should be identical.
C The file V1.LBL must be present in the default directory the program
C is run from.
C
C Platforms this has been tested on are:
C   1) SGI/Irix
C   2) Sun Sparc/Solaris
C   3) VAX/VMS
C   4) Dec 3100/Ultrix
C   5) Dec Alpha/OpenVMS
C

        implicit none

        INCLUDE 'OAL_FORTRAN.INC'
        INCLUDE 'L3_FORTRAN.INC'

C These variables must be the same size as C long's or pointers.
C On a Dec Alpha they should be integer*8;  on all other platforms they
C should be integer*4.

        integer*4          ODLTREE, SIZE
        integer*4          ROOT_NODE, ODLTREENODE, IMAGE_NODE
        integer*4          HISTOGRAM_NODE
        integer*4          IMAGE_OBJECT1, HISTOGRAM_OBJECT
        integer*4          IMAGE_OBJECT2, IMAGE_HANDLE
        integer*4          IMAGE_HISTOGRAM_PTR, IMAGE_PTR
        integer*4          START_LINE, STOP_LINE
        integer*4          START_SAMPLE, STOP_SAMPLE

C These variables must be the same size as C int's.

        integer*4          OBJECT_POSITION, ERR
        integer*4          HISTOGRAM(256)

C These variables must be the same size as C short's.

        integer*2          EXPAND, NOMSGS, SEARCH_SCOPE

        byte               IS_IN_MEMORY
        character*80       LABEL_FILENAME, ERRFILESPEC
        character*128      STR
        character*50       KEYWORD_NAME, KEYWORD_VALUE, OBJECT_CLASS


C Local function declarations:
```

```
      integer*4  COMPARE_HISTOGRAMS


C Read in the label from V1.LBL.  This is an attatched label with
C variable-length records, containing a HFD compressed Voyager image
C of miranda, an engineering table and two histograms.

      type *,'Reading V1.LBL'
      LABEL_FILENAME = 'V1.LBL'
      ERRFILESPEC = ' '
      EXPAND = 0
      NOMSGS = 0
      ROOT_NODE = OaFortParseLabelFile( LABEL_FILENAME, ERRFILESPEC,
     1                                  EXPAND, NOMSGS)
      if (ROOT_NODE .eq. NULL) then
        call OaFortReportError( 'Error: OaFortParseLabelFile ' //
     1                          'returned 0!')
        go to 999
      end if

C Print out the class names of all the root node's children.

      ODLTREENODE = OdlFortLeftmostChild( ROOT_NODE)
      call OaFortReportError( 'Children of root node are:')
      do while (ODLTREENODE .ne. NULL)
        ERR = OdlFortGetObjDescClassName( ODLTREENODE, OBJECT_CLASS)
        call OaFortReportError( 'OBJECT_CLASS = ' // OBJECT_CLASS)
        ODLTREENODE = OdlFortRightSibling( ODLTREENODE)
      end do
      type *,' '

C Get the IMAGE node.

      OBJECT_CLASS = 'IMAGE'
      KEYWORD_NAME = ' '
      KEYWORD_VALUE = ' '
      OBJECT_POSITION = 0
      SEARCH_SCOPE = ODL_RECURSIVE_DOWN
      IMAGE_NODE = OdlFortFindObjDesc( ROOT_NODE, OBJECT_CLASS,
     1                                 KEYWORD_NAME, KEYWORD_VALUE,
     2                                 OBJECT_POSITION, SEARCH_SCOPE)
      if (IMAGE_NODE .eq. NULL) then
        call OaFortReportError( 'Error: OdlFortFindObjDesc ' //
     1                          'could not find IMAGE node!')
        go to 999
      end if

C Read the IMAGE into memory two different ways.

      type *,'Reading in IMAGE with OaFortReadImage...'
      IMAGE_OBJECT1 = OaFortReadImage( IMAGE_NODE)
      if (IMAGE_OBJECT1 .eq. NULL) then
        call OaFortReportError( 'Error: OdlFortReadImage failed!')
        go to 999
      end if

      type *,'Reading in IMAGE with OaFortReadPartialImage...'
      IMAGE_HANDLE = OaFortOpenImage( IMAGE_NODE)
```

```
        if (IMAGE_HANDLE .eq. NULL) then
          call OaFortReportError( 'Error: OaFortOpenImage ' //
        1                         'failed!')
          go to 999
        end if
        START_LINE   = 1
        STOP_LINE    = 800
        START_SAMPLE = 1
        STOP_SAMPLE  = 800
        IMAGE_OBJECT2 = OaFortReadPartialImage( IMAGE_HANDLE,
        1                                       START_LINE,
        2                                       STOP_LINE,
        3                                       START_SAMPLE,
        4                                       STOP_SAMPLE)
        if (IMAGE_OBJECT2 .eq. NULL) then
          call OaFortReportError( 'Error: OdlFortReadPartialImage ' //
        1                         'failed!')
          go to 999
        end if
        ERR = OaFortCloseImage( IMAGE_HANDLE)

C Get the IMAGE_HISTOGRAM node.

        OBJECT_CLASS = 'IMAGE_HISTOGRAM'
        HISTOGRAM_NODE = OdlFortFindObjDesc( ROOT_NODE, OBJECT_CLASS,
        1                                KEYWORD_NAME, KEYWORD_VALUE,
        2                                OBJECT_POSITION, SEARCH_SCOPE)
        if (HISTOGRAM_NODE .eq. NULL) then
          call OaFortReportError( 'Error: OdlFortFindObjDesc ' //
        1                         'could not find IMAGE_HISTOGRAM ' //
        2                         'node!')
          go to 999
        end if

C Read the IMAGE_HISTOGRAM into memory.

        type *,'Reading in HISTOGRAM'
        HISTOGRAM_OBJECT = OaFortReadHistogram( HISTOGRAM_NODE)
        if (HISTOGRAM_OBJECT .eq. NULL) then
          call OaFortReportError( 'Error: OdlFortReadHistogram ' //
        1                         ' failed!')
          go to 999
        end if

C Strip the HISTOGRAM object of everything but the data.  The returned
C value is a pointer to the data.

        IMAGE_HISTOGRAM_PTR = OaFortExportObject( HISTOGRAM_OBJECT)

C Print out a few values to see if it worked.

        call OaFortReportError( 'First 5 values should be:       ' //
        1                       '529   182   139   284   315')
        STR = 'First 5 histogram values are: '
        call PRINT_HISTOGRAM_VALUES( %val( IMAGE_HISTOGRAM_PTR))

C Calculate the histogram from the image, and put the result in
HISTOGRAM.
```

```
C Instead of calling OaFortExportObject to get the data_ptr, get values
C from the Oa_object structure, just to show a different way of doing
C it.  Then pass the data_ptr into CALCULATE_HISTOGRAM.

        call OaFortGetObjectInfo( IMAGE_OBJECT1, ODLTREE, IMAGE_PTR,
        1                         SIZE, IS_IN_MEMORY)
        call CALCULATE_HISTOGRAM( %val(IMAGE_PTR), HISTOGRAM)

C Compare the calculated histogram with the histogram read from the
C file.

        ERR = COMPARE_HISTOGRAMS( %val(IMAGE_HISTOGRAM_PTR), HISTOGRAM)
        if (ERR .ne. 0) then
          go to 999 !Error message already printed
        end if

C Free all the objects.  HISTOGRAM_OBJECT was already freed by export
C call.

        call OaFortDeleteObject( IMAGE_OBJECT1)
        call OaFortDeleteObject( IMAGE_OBJECT2)

        type *,'All tests worked!'
999     end


        subroutine PRINT_HISTOGRAM_VALUES( IMAGE_HISTOGRAM)
        integer*4 IMAGE_HISTOGRAM(*)

        character*32  TMP_STR
        character*128 STR
        integer*4     I

        do I = 1,5
          write( TMP_STR, '(I5)') IMAGE_HISTOGRAM(I)
          STR = STR(1:index( STR, '        ')) // TMP_STR
        end do
        STR = 'First 5 values are:            ' // STR
        call OaFortReportError( STR)
        end


        subroutine CALCULATE_HISTOGRAM( IMAGE, HISTOGRAM)
        byte      IMAGE(800,800)
        integer*4 HISTOGRAM(256)

C Define an equivalence (overlay) so that can access the MSB and LSB
C bytes of a 2-byte integer independently.  By setting the 2-byte
C integer to 1, then testing which byte is 1, you can find whether the
C platform you're running on uses MSB integers or LSB integers, and thus
C which byte of a 2-byte integer is the MSB byte.
C When calculating the histogram, each 1-byte image pixel is copied into
C a 2-byte integer, then the integer's MSB byte set to 0.  This is then
C used to index the HISTOGRAM array.  Indexing the array with a byte
C directly from the image doesn't work because pixel values between 128
C and 255 give negative numbers, and an error when trying to index.

        byte              PIXEL_BYTE_MASK(2)
```

```
      integer*2         PIXEL, I, J
      equivalence( PIXEL, PIXEL_BYTE_MASK)
      byte              MSB_BYTE

      PIXEL = 1
      if (PIXEL_BYTE_MASK(1) .eq. 1) then
        MSB_BYTE = 2
      else
        MSB_BYTE = 1
      end if

C Initialize output HISTOGRAM.

      do I = 1,256
        HISTOGRAM(I) = 0
      end do

      do I = 1,800
        do J = 1,800
          PIXEL = IMAGE(I,J)
          PIXEL_BYTE_MASK( MSB_BYTE) = 0
          HISTOGRAM( PIXEL+1) = HISTOGRAM( PIXEL+1) + 1
        end do
      end do
      end


      integer*4 function COMPARE_HISTOGRAMS( HISTOGRAM1, HISTOGRAM2)
      integer*4 HISTOGRAM1(256)
      integer*4 HISTOGRAM2(256)

      integer*4 I

      do I = 1,256
        if (HISTOGRAM1(I) .ne. HISTOGRAM2(I)) then
          type *,'Error: histograms do not match; I = ',I
          COMPARE_HISTOGRAMS = 1
          return
        end if
      end do
      type *,'histograms match'
      COMPARE_HISTOGRAMS = 0
      return
      end
```

Index

—P—

—Q—

—R—

—S—

—T—

—U—