U. S. Department of the Interior
U. S. Geological Survey

ODDF: a File I/O Subroutine Package Implementing NASA PDS Data Description and USGS Map Projections, Version 1.6

paper edition

by

Michael W. Webring

Open File Report OF 98-765

Contents

Introduction

The audience for this report is the scientific programmer engaged in algorithm development that leads to the routine processing of earth science data. The goal of the Object Description Data File (ODDF) system is to allow many programmers with a variety of programming goals to concentrate on the computational problem and leave the details of data file structure to the system. Data files in a common format then form the link among separate programs that comprise a processing system. This report will focus on the usage of the system for data I/O and the descriptions in the resulting files.

ODDF is an implementation of the NASA Planetary Data System (PDS) (Martin, et al, 1988) and creates data files that are self-describing. The plaintext header preceding the data is human readable without special software and contains enough information on file structure and data attributes that, to a great extent, the data may be extracted and interpreted without the ODDF system or external documentation. Data descriptions are in the form of keyword-equals-value attributes, comments, and text; arbitrary codes are kept to a minimum. An important feature of PDS is that the structure of the data object is customizable for fast access or simple coding to fit a variety of processing environments. In addition there is enough flexibility in the data description to define data objects that do not fit the current definitions. Point data, for instance, may have a variety of structures that may not fit into either table or series form. The PDS object descriptions (metadata) may be updated with no changes to the basic file structure.

The basic data types implemented in this release are points, lines and 2-dimensional grids; each may be registered to a given coordinate system or a cartographic map projection. Point or station data consists of a short identifier, cartographic position and multiple measured or derived quantities. Lines are aggregates of points defined by the identifier. Grids are a single measured quantity evenly sampled over an area. Point and line data are stored in a data object similar to a table while grids are stored in a format similar to raster scans.

Besides the basic data I/O, the system contains map projection calculation, interactive prompting and journaling of user responses, and filename manipulation. Journaling can be an aid to documentation of a processing stream and may eventually evolve into a history record embedded in the data file. The primary filename manipulation is the addition of version numbers so users do not accidently overwrite data or have program flow interrupted because of duplicate filenames in a given directory.

The full description and inheritance of the mapping coordinate system is one of the major features of ODDF. Each map projection has a set computational parameters, some like the definition of the ellipsoid shape and size, are common to all projections but many others like the distance to a perspective viewpoint are unique to a given projection. The number of these parameters and the need for flexibility for future expansion was the driving force behind the adoption of the PDS map projection data object with its verbose plaintext description. The advent of the Global Positioning System (GPS) with the accompanying change of both horizontal and vertical datums illustrates the upgrade potential of the description. Essentially, new subroutines were added to the projection sub-system of ODDF and the information stored in new attributes without change to the existing attributes. The inheritance portion of the system required no changes.

Whenever necessary, the ODDF system may be queried for the values of data objects and the answers can remain local to the calling routine, which facilitates modular design of application programs. ODDF itself is written in a well defined subset of Fortran-77 and in an object oriented style that is robust and upgradable. Wherever possible the system provides explicit responses for ambiguous syntax in Fortran such as error branching during keyboard interaction. The PDS data file documents itself so very simple hardwired programs can extract the data when necessary. The self-documentation allows data recovery from any stage of processing so that files on archival media are useful in a wide variety of circumstances.

ODDF is in the form of semi-independent subsystems that communicate through subroutine arguments while internally the subsystem routines communicate through shared data objects. So, for instance, the prompting,

filename manipulation and map projection subsystems all reference a character string subsystem but not each other. Point and grid data I/O subsystems do not reference each other but do reference the PDS object description language (ODL) routines which in turn references the character subsystem. The opening and closing of files is handled by the filename manipulation system and communication with the user's screen and keyboard is handled by two routines in the prompting system . Data I/O is limited to several routines in each of the ODL, point and grid systems. The design goal was to isolate the whole system from those aspects of Fortran that vary from vender to vender and to allow as much independence as possible between the different subsystems.

The ODDF system has grown to a substantial size, some 350 subroutines, many of which deal with the construction and parsing of the plaintext descriptions. Much of the size of the ODDF system stems from the fact that whatever is simple for humans, like reading text, is difficult for programs. All of the subroutines compile on every Fortran compiler available for testing, but the number of subroutines will cause problems on machines of limited capacity. It is hoped the power of hardware and operating systems will continue to improve so that PC's, for instance, will be able to run this system in the future. When the ODDF system cannot be brought online, the programs using it must be provided with some alternative I/O, the typical case might be when only one of the programs in a software package is desired. Since an application program connects to the system at only a few places, a simple ODDF emulator is feasible. An emulator is being tested, but is not available with this release.

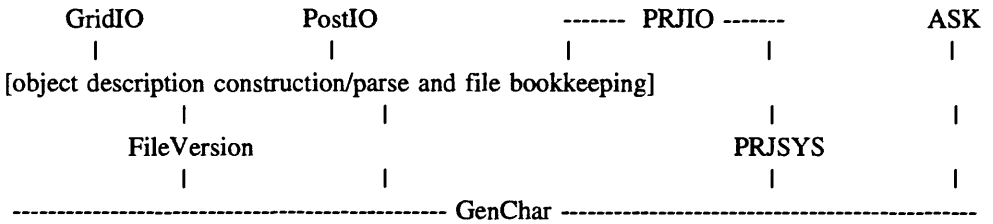
Source code availability

Source code and installation instructions for the ODDF system are available from the Internet at host musette.cr.usgs.gov, directory /pub/oddf.

Components of ODDF

ODDF is divided into modules or functionalities. Several, like GridIO, PostIO and PRJIO are higher level and call lower level modules like FileVersion and GenChar. Each of the modules below with the exception of the construction/parse layer have detailed user guides later in this report.

The relation among the modules can be diagrammed as follows, where a higher module may call or reference module at a lower level. The vertical lines indicate some of the relationship between functions.



GenChar, is a general character manipulation package that contains functions for parse, concatenation, search, replace and the like. The routines are primarily designed for the parse and construction of plaintext descriptions in PDS/ODDF file headers, however, they are also useful as an extension to Fortran-77 character handling.

FileVersion is a package that resolves operating environment inconsistences and opens/closes files. It also provides a file version capability and other filename manipulations. FileVersion is called by GridIO, PostIO and the bookkeeping layer; and may be called by the user's application to open files with custom formats.

The layer labeled "[object description construction/parse and file bookkeeping]" contains much of the keyword recognition and higher level character manipulation as well as file buffers, pointers and the like. The layer may be considered as a hidden or undocumented layer.

GridIO handles the basic operations of reading and writing grid data.
PostIO handles the basic operations of reading and writing point and line data.
PRJIO is the interface between the object description layer and PRJSYS.
PRJSYS is the map projection computation package.
ASK handles interaction with the user.

The components in the above diagram with the exception of PRJIO and PRJSYS normally reside in a single library on UNIX systems. PRJIO and PRJSYS are normally contained in two separate libraries for space saving reasons because most application programs do not reference or manipulate map projections once a data set is generated and therefore these modules are used less often. The above library organization is somewhat variable and modules like ASK, FileVersion or GenChar can be compiled separately if desired, provided the vertical dependencies are kept in mind.

Linking libraries on UNIX systems

The main ODDF system, map projection calculation and the interface between them are usually contained in separate archives. This organization allows ODDF as a unit to be smaller and enforces a separation in functionality.

The libraries referred to in this document are:

lib_prjio.a	PRJIO system, contains POaaaa routines.
lib_prjsys.a	PRJSYS system, contains MPaaaa routines, the IACTPJ interactive setup routine and the GCTP subroutines.
lib_oddf.a	contains ODDF. The major subsystems include: ASK, GridIO, PostIO, FileVersion and GenChar which have their own documentation.

On UNIX systems, these libraries should be linked in this order where lib_prjio.a is the most dependent and lib_oddf.a is the most independent.

Basic system design and use criteria

The ODDF system designed for transport to all machines with a Fortran-77 compiler, but selected mil-spec non-ANSI constructs do permeate the system. For example ordinary statements are preceded with a tab rather than 6 spaces and the lowercase alphabet is used since automatic translation of these constructs is simple. The six character subroutine and variable name limit, however, is strictly adhered to.

The system is written with a subset of robust Fortran constructions and translation to C using an automatic code generator is a possibility planned for version 2. The translation to an OOP language like C++ is not planned for the immediate future.

Object Oriented Programming (OOP) is the basic design philosophy used to develop ODDF. While this is a new approach tuned more to the C programming language than Fortran, it is really just a set of programming rules that lead to stable and upgradable systems. Fortran cannot enforce all the public vs. private structures designed into an OOP language, and therefore programmers need to be aware of some pitfalls.

One of the basic OOP constructs is the data object and in the Fortran version of ODDF, common areas are used to contain data objects. To simplify the modification and verification of these data objects, the Fortran "include" statement is used to embed a master common area definition into subroutines. Each of the high-level modules diagrammed above have a separate include file. It is **imperative** that an application not use the same common area name. The include files and named common areas are listed below.

The system also does not distinguish between public and private functions or subroutines and, for instance, the entire plaintext parse and construct level diagrammed above as a hidden level is accessible to application programs. It is not recommended procedure to use system or private routines in application programs. The routines described in the accompanying user guides (eg. GridIO) are the points at which application programs interact with ODDF. These routines are frozen in external functionality so that ODDF system upgrades will not affect an application. Any ODDF routine not described in the application level user guides must be considered as subject to change without notice.

Fortran named common areas

The ODDF system uses the following common areas, tabulated by the include files for each module. Each of the include files contain corruption detect fields but they are checked only during certain operations. The detect fields are designed to detect data that overrun array boundaries (every programmer knows a way to do this) and not to enforce a public/private usage. In any case, application programs must not declare or include any of the following named common areas. Upgrades to the ODDF system will be accomplished by adding common areas in sequence to the appropriate include file.

module	include file	named common areas
GridIO	grdio.cmn	iogd1 to iogd7,
PostIO	expio.cmn	ioxp1 to ioxp14,
PRJIO	posys.cmn	mopj1 to mopj9,
PRJSYS	mpsys.cmn	mps01 to mps12,
ASK	ask.cmn	ioak1 to ioak1,
FileVersion	versfile.cmn	iofv1 to iofv5
hidden	odl.cmn	iood1 to iood22,
hidden	pdsio.cmn	iopf1 to iopf4,
hidden	history.cmn	iohs1 to iohs1,
hidden	table.cmn	iotb1 to iotb2.

Explanation of typography

The following typographic conventions are used to indicate different textual interpretations in this document. Fortran code and PDS labelling examples will have slight variations from the general forms given below.

Words in capital letters indicate subroutine names, program variable names, acronyms or important items. Bold

typeface indicates section titles or important items. Italics indicate example Fortran code fragments.

A character string enclosed in double quotes in text sections (eg. "^") is the actual character string whereas single quotes indicate a character string that is a template or example that may vary somewhat. For instance, many routines have a 'read' mode in the argument list and also be described in the text. In this case the 'read' either indicates the conceptual operation or the spelling can vary. Example Fortran code uses the single quote to denote the actual contents of a character variable (see below), but since these examples are italicized or are complete listings this inconsistency should not be a problem.

Underlines indicate excerpts from the user's screen, for instance if a program prints "Enter zstart" to the user's screen then it is underlined in this text. Angle brackets, < >, indicate a key on the keyboard; for instance, the control-d entry from a keyboard will be denoted as <ctl><d>. Angle brackets appearing in a PDS label are delimiters for units character strings, example "false_easting = 0.0 <kilometers>".

The underscore "_" is sometimes used to link words (typically nouns) into a conceptual unit. The PDS keywords introduce this usage where the words should not be separately parsed as they would be when delimited by spaces, eg. RECORD_BYTES. This document will occasionally use the underscore to refer to computing or system environment entities (eg. current_record, line_id) where multiple consecutive nouns may lead to confusion. In prose sections, the hyphen will be used in the normal sense.

Explanation of numeric notation

Scientific notation will be given as 10^38 (10 to the 38th power) or as 0.9999e38 with the mantissa 0.9999 preceding the exponent 38.

The caret "^" is also used in the PDS labels to indicate a pointer keyword, eg. "^express_series = 5".

Variable types are specified for each argument to a subroutine. The calling program must pass a variable of the same type to the subroutine or risk a program crash or (perhaps worse) inconsistent behavior. Five variable types are used:

'char*(*)'	indicates a variable length character string,
char*8	indicates a character string 8 bytes in length,
'integer'	indicates a 4 byte integer,
'real*4'	indicates a 4 byte single precision floating point variable and
'double'	indicates an 8 byte double precision variable (aka. real*8).

The Fortran default integers are I to N inclusive. In the usage guides that follow, integer variables will begin with I to N to correspond with typical Fortran code. Variables beginning with "N" will normally refer to 'number of', eg. NCHAN. Character strings and floating point variables will use variable names that do not start with I to N. Most example code will not include typing statements for variables to avoid unnecessary complexity.

Supplied vs. returned subroutine arguments

Each variable in the following user guides has a supplied, returned or S/R designation. If a variable is supplied, the routine will not try to change it and therefore either a variable may be used or a constant may be embedded in the calling statement. For example, the first argument to the ASKF4 routine is supplied by the calling program and therefore either:

*character prompt*80*
call askf4(prompt, zs, kboard) **or**

call askf4('Enter zstart', zs, kboard) will work.

If the routine is returning an updated value, there must be a variable in the location. The second argument to the ASKF4 routine is returned to the calling program and therefore:

call askf4('Enter zstart', zs, kboard) will work, whereas
call askf4('Enter zstart', 10.0, kboard) with "10.0" as the second argument will crash the program because the routine cannot update the argument.

Example subroutine documentation

The example subroutine documentation below shows many of the typographic usages from the previous section. Both arguments to the GCLOSE routine are supplied and none are returned.

begin example:

PROGRAM USAGE: GCLOSE

Close a file unit that was opened with GOPEN. This routine **must** be called to allow ODDF to adjust the contents of the plaintext headers after a new grid has been written.

programming example

call gclose(iunit, stat)

explanation

name	type	supplied/returned	description
IUNIT	integer	S	File unit number.
STAT	char*(*)	S	Closing status either 'keep', 'delete' or ' '. Blank is an implied 'keep'. Delete is used for scratch files that are deleted when closed.

end example

Error state variable

The basic ordering of parameters in all subroutine calls follows this pattern (there are no function calls in the Fortran version of ODDF):

call name(opr, file_unit_number, arg, arg, ..., error).

Any of the entries may be absent, but when present are in this relative order. OPR is either 'read' or 'write', meaning read from the ODDF system or write to the system. Actual I/O to the data file does not necessarily happen when a routine is called with a read or write. FILE_UNIT_NUMBER is the number referring to a data file. ARG,ARG... may be considered as more or less related elements of a data object. The ERROR state parameter equals either zero or not-zero, where zero is 'ok' and the calling routine must supply the interpretation of not-zero or 'not-ok'.

The ODDF system cannot intentionally halt an application. The various levels of ODDF communicate status via the error parameter and take action based on the current state. If an unrecoverable error exists, the routines will eventually communicate this error state to the application program which then has the option of halting or alerting the user.

The OOP general rule for writing is to thoroughly check a data object before writing, therefore do not check during a read. Similarly, application programs should consistently check the error state variable during writes. However, a file may be corrupted without the system's knowledge so application programs should occasionally check the status of 'read' routines.

As mentioned above, the error state parameter is either equal to zero or not-zero (ok, not-ok) where the calling routine determines the interpretation of not-ok. When the system encounters an error that has a repair path, it effects repairs silently. Errors that can be avoided by the application and unexpected inconsistencies are handled verbosely. The syntax for a verbose error message is **%%subroutine name: error description** (where underline indicates a message printed on the users screen).

Data I/O to a file is one case the system cannot repair - if a file is not open or if a disk drive fills up there is nothing the system can do. Data input from a file is expected to generate an occasional End_Of_File (EOF). The system handles this case silently and sets the error state parameter to not-zero. The data object the routine is preparing is invalid but there may not be an explicit set of no-data values put into the data object (ie. the routine may leave the returned data object unchanged).

There is no case when a file input routine will return an error parameter equal zero when the file is at EOF.

Summary of action for reading a file:

error = 0 and the file not at EOF. Data object ok and available via the argument list variables.

error = 0 and the file at EOF. Not supposed to occur.

error != 0 (not-zero) and the file at EOF. Data object contains invalid values (either explicit no-data values or values unchanged from the last call to the routine). The routine is silent.

error != 0 an the file not at EOF. Data object contains invalid values and the routine will generate an appropriate error message. The most likely way to get this case is to specify some parameter incorrectly (eg. attempt to read a grid row longer than the file contains).

Point and grid file header examples

The data files created by ODDF begin with a PDS type plaintext header. Although PDS allows the header to be detached in a separate file, keeping the header attached to the data means the two cannot become separated from each other as might be the case during routine data processing. Example headers for grid and point files follow. Appendix A contains example code to extract grid and point data using simple programs a person modifies to conform to the data after they read the header.

Example grid file header

Below is a grid header that was captured from a screen print of the binary direct access data file. The text consists of ASCII characters formatted with carriage-return/line-feed pairs contained in as many file records (ie. LABEL_RECORDS) as necessary. The keywords and their definitions were abstracted from the PDS version 1.0 manuals (Martin and others, 1988) available at the time ODDF version 1.1 was being written (circa 1991) with some update from PDS version 3.0 (JPL, 1992). The keywords attempt to conform to the ISIS Qube (Martin and others, 1988 and JPL, 1992) but the ODDF data files have not been tested with the ISIS description reader.

Explanation

The first line, ODDF = "version = 1.6.1, dictionary = 1.2.1", contains a system id and a text string to identify the system version and keyword dictionary used to create the file.

Comments are delimited by /* and */ similar to the C programming language syntax.

The "^" indicates the keyword is a pointer to either a record number later in the current file or a file name when the header and data object are in separate files (the filename form is not currently activated). Fortran is a record oriented language, but a short description in the comments shows how to compute the start byte of the grid core.

The PDS map_projection object description, as seen below, also includes the 'geographic' coordinate system of longitude/latitude. While not strictly a map projection, the geographic coordinate system functions in the same manner as any other 2-d coordinate system.

Further explanation is deferred to Appendix A where simple Fortran code uses the quantities in the header to read the data core.

Begin example ODDF grid header

```
ODDF = "version = 1.6.1, dictionary = 1.2.1"

record_type = fixed_length
record_bytes = 7200
file_records = 1501
file_state = clean
label_records = 1

/*  pointer to data object */
```

```
^qube = 2

/* A qube is a generalized grid. Each file record contains          */
/* one row of the grid. The first element in the first row is the grid  */
/* node that is located at the coordinates described by axis_start.     */
/* The storage mode is binary and the first byte of the grid is located */
/* at ( record_bytes * (^qube-1) ) + 1, when the value of ^qube is a    */
/* number; else start_byte is 1. The true core value is scaled with     */
/* core_base + core_multiplier * stored value. Invalid nodes are        */
/* flagged with values greater than or equal to core_null.              */

object = qube
  axes = 2
  axis_start = ( -95.99167, 25.00833 )
  axis_interval = (.16666668E-01, .16666668E-01 )
  core_items = ( 1800, 1500 )
  core_item_type = real
  core_item_bytes = 4
  core_name = "lon_range=(-96,-66) lat_range=(25,50) core_unit=meters"
  core_null = .99999997E+38
  core_base = 0.0
  core_multiplier = 1.0

  byte_order = LSB

  object = map_projection
    map_projection_desc = " "
    horizontal_datum = "NAD27"
    map_projection_type = "geographic"
    map_projection_unit = "degrees"
  end_object = map_projection

end_object = qube

end
```

End of example header

Example point and line file header

Below is a posting file header that was captured from a screen print. Where feasible, the keywords conform to the PDS table and series data objects. The structure of the posting file physical records predates ODDF and was difficult to describe with PDS version 1.0 keywords (Martin, et al,1988) and so a new data object termed the express_series was defined. PDS version 3.0 (JPL, 1992) recognizes buffered forms of binary tables but the descriptive keywords have not yet been incorporated into ODDF 1.6.

There are nested sets of object descriptions in the header; inside the express_series description is a table description consisting of column descriptions. The fourth column description has several attributes added to illustrate a complete description; the extra attributes are normally not written to the header unless they vary from the default values. The table description can be somewhat lengthy when the number of data channels gets into the 10's, but for all its verbosity is nonetheless readable by people. If the ODDF system is expanded to describe ASCII tables, the current keywords will remain the same.

Explanation

The attribute "false_easting = 0.0 <kilometer> " contains a kilometer units designation in angle brackets after the value. This is PDS standard usage and serves to make the units of values explicit and while the units designation is optional it serves to make the label more self-documenting. For instance, the attribute for the location of the data could be "^express_series = 3 <file_records>" making it more apparent to what the ^express_series pointer is referring.

Begin example ODDF posting file header

```
ODDF = "version = 1.6.1, dictionary = 1.2.1"

record_type = fixed_length
record_bytes = 2048
file_records = 366
file_state = clean
label_records = 2

/* pointer to data object */

^express_series = 3

object = express_series

/* The express_series is a buffered set of station logical records. The    */
/* express_series file record consists of a 3 integer header followed by   */
/* a 2-dimensional data array and optional pads. The header is             */
/* n_word_per_log_rec, n_log_rec, n_data+pad_words. The write mode         */
/* for the record is binary and all words are 4 bytes written in the       */
/* order given by byte_order. The station posting logical record           */
/* consists of an optional 8 byte id followed by binary 4 byte real data   */
/* channels. Profiles may be constructed from a sequence of records        */
/* that have the same id characters.                                       */
```

```
logical_record_type = posting
logical_record_bytes = 20
id_bytes = 8
profile_id_bytes = 4
profile_id_start_byte = 1

byte_order = MSB

object = map_projection
  map_projection_desc = " "
  horizontal_datum = "NAD83"
  ellipsoid = "WGS84"
  a_axis_radius = 6378.137 <kilometer>
  b_axis_radius = 6378.137 <kilometer>
  c_axis_radius = 6356.7523141 <kilometer>
  map_projection_type = "transverse mercator"
  map_projection_unit = "kilometers"
  reference_longitude = -105.0 <deg>
  reference_latitude = 40.0 <deg>
  false_easting = 0.0 <kilometer>
  false_northing = 0.0 <kilometer>
  center_scale_factor = .9996
end_object = map_projection

object = table
  interchange_format = binary
  name = "Test data"
  columns = 4
  object = column
    Name = "record id"
    start_byte = 1
    bytes = 8
  end_object = column
  object = column
    name = "X position"
    start_byte = 9
    bytes = 4
    unit = "kilometer"
  end_object = column
  object = column
    name = "Y position"
    start_byte = 13
    bytes = 4
    unit = "kilometer"
  end_object = column
```

```
    object = column
      name = "aircraft roll"
      data_type = real
      start_byte = 17
      bytes = 4
      unit = "degree"
      exclude = .9999999e38
      factor = 1.0
      base = 0.0
    end_object = column
  end_object = table
end_object = express_series

end
```

End of example header

ODDF Keyword Dictionary

Version 1.2, 98-10-29

The items in the dictionary are listed in the order they typically appear in the ODDF labels. The definition of terms start with the most general terms and progress to more specific ones. The introduction to lists gives context for the definitions. Most keywords and definitions follow PDS version 3 standards (JPL, 1992).

Definition of terms

label The plaintext data description that may either be in a separate file detached from the data object file or be attached to precede the data object in a complete data file. Labels contain only printing characters. Detached labels have not been activated with this release of ODDF.

attribute A statement in the label comprised of four elements: keyword, equal sign, value with optional units designation enclosed in angle brackets, and line terminator. Attributes describe the data and the data file. Example: record_bytes = 2048 <bytes>.

keyword A word comprised of alphabetic, numeral and/or underscore characters with no embedded blanks. The first letter is an alphabetic character. Keywords that are not PDS standard will be noted as such.

value The value may be a literal, a text string, a number or a parenthesized list of values. When the value is a number, then an optional units designation enclosed in angle brackets may follow the number.

literal values An attribute value that is an string of alphabetic, numeral and/or underscore characters with no embedded blanks and optionally may contain blanks or other printing characters when enclosed in apostrophes. Typically strings that are predefined may be interpreted as literal values.
Examples: record_type = fixed_length or ellipsoid = 'Clarke 1866'.

character values An attribute value that is a quoted string of characters. Character or text strings may consist of any printing characters including line terminators.
Example: name = "total magnetic field".

real values An attribute value that is a string of visible characters all of which are either numeric or the optional delimiters (ie. blanks, parentheses and commas) used to format an array of real numbers. Real numbers may be formatted in scientific or engineering notation as a mantissa multiplied by a power-of-ten exponent denoted with either an "e" or "d" for single or double precision mantissas.
Example: axis_start = (-120.5, 4.3567e03).

integer values An attribute value that is a string of visible characters all of which are either numeric or the optional delimiters used to format an array of integer numbers.
Example: axis_items = (325, 457).

logical record A grouping of characters and/or numbers, that is interpreted as a unit. Label attributes and posting records are examples.

posting record A specific type of point or station data record that consists of an optional eight character id and an array of real numbers. The array of real numbers (data channels) typically consists of an optional location and measured or derived quantities. The location when present is a coordinate n-tuple (typically a pair of numbers), and the measured quantities are application determined.

file record A grouping of bytes read or written to a file as a unit. Also known as a physical record.

Binary encoding

Several keywords and/or values describing the hardware type, byte ordering of binary numbers and encoding style of real numbers are necessary for automatic transport between systems, however, lacking these it should be assumed that:

1) a byte is 8 bits,
2) ASCII encoding is used to store one character per byte,
3) IEEE encoding is used to store real numbers in 4 bytes,
4) 2's complement encoding is used to store multi-byte signed integers in 2 or 4 bytes,
5) single byte integers are unsigned and,
6) the byte ordering of stored reals or integers is either most_significant_byte or least_significant_byte.

PDS defines 30 or 40 values that are associated with data type keywords and while many are aliases for the basic integer and real numbers, the list was far longer than needed by the initial implementation of ODDF. The basic data types used in ODDF are real and integer, this is modified by a byte count (eg. 2 and 4 byte integer, 4 and 8 byte real); and with the current release, a byte order for a number stored in a file (ie. most_significant_byte or least_significant_byte).

PDS file format keywords

The following keywords are present at the beginning of the label immediately after an optional statement that is the PDS archival string or the version numbers of the software that created the file. The id statement must be short enough that the record_type and record_bytes attributes are within the first 256 characters of the file.

keyword	value_type	description
record_type	literal	Record_type = fixed_length for unformatted direct access and record_type = stream for text. The qube and express_series data objects use fixed_length records.
record_bytes	integer	The number of bytes per record for fixed_length records.
file_records	integer	The number of records in fixed_length record files.
label_records	integer	The number of plaintext label records preceding the data object for fixed_length record files.
file_state	literal	The file state is "clean" if the label has been updated before the file is closed otherwise the value is "dirty". File_state is optional in PDS version 3.

PDS data object pointers

An ODDF file consists of a label, an optional history object, and a data object. ODDF defines two types of data objects: the qube and the express_series, where the qube contains grid data and the express_series contains either random point data or point data with line designation. The history record has not been activated with version 6 of ODDF.

keyword	value_type	description
^history	integer or character	The pointer to the history is either an integer indicating a file record or a quoted text string that gives a filename for the history. When the history is embedded in the data file, the preferred location is immediately after the label and before the data object.
^qube	integer or character	The pointer to the qube (grid) data object is either an integer indicating a file record or a quoted text string that gives a filename for the qube. This version of ODDF does not use the filename form. Example: ^qube = 2.
^express_series	integer or character	The pointer to the series (point) data object is either an integer indicating a file record or a quoted text string that gives a filename for the series. This version of ODDF does not use the filename form. The term "express_series" refers to the specific type of record blocking used for the posting file record. Not a PDS standard pointer.

PDS group delimiters

keyword	value_type	description
object	literal	First attribute in a set of data description attributes. Object and end_object group related attributes together into a unit. May be set equal to a literal string as an identifier. Example: object = qube.
end_object	literal	Last attribute of a data description group. Value is optional.

PDS text

keyword	value_type	description
text	character	General text may be included wherever appropriate in the PDS label to explain or expand the description of the data object. The value of the text attribute is a quoted character string that has embedded carriage-return/line-feed pairs to format the text into lines that contain no more than 80 characters. The embedded line terminators may be considered part of the text. Literal strings enclosed in apostrophes may be embedded in the text, but other quoted character strings are not used.

PDS QUBE keywords

The contents of a grid are measured or derived values sampled at regular intervals in two (or more) user selected dimensions. A grid value is defined in a **nodal or point** sense where the value stored in the grid file is valid at the X/Y coordinates of the node; between grid nodes a value is not defined.

A qube is a generalized form of grid data that includes prefix and suffix blocks that may contain qualifying information pertaining to the grid core (the measured or derived quantity of interest). ODDF currently implements the grid core information and not the side blocks. In addition, the current version of ODDF does not support multiple planes of 2-dimensional data and so the pertinent band ordering keywords are not given here.

The following keywords and map projection description are contained in an "object = qube", "end_object" attribute pair.

keyword	value_type	description
axes	integer	Number of axes in the core (the sampled grid). ODDF is currently limited to axes = 2, PDS allows up to 6.
axis_start	real array	Location of the first node in the grid. Axis_start is an array of length the value of axes. Example: axis_start = (-27.34, 67.44).
axis_interval	real array	Sampling interval along each axis. The array is the same length and the values in the same units as axis_start. When the map_projection

		object description is present, both axis_start and axis_interval are in the described units; otherwise the units are general. Example: axis_interval = (.25, .25).
core_items	integer array	Number of samples (grid nodes) in the core along each axis. Example: core_items = (345, 440).
core_item_type	literal	Data type of the core. ODDF allows 'integer' or 'real'. Example: core_item_type = real.
core_item_bytes	integer	Number of bytes in each core value. ODDF is currently limited to 4. Example: core_item_bytes = 4.
core_name	character	Title of the grid data up to 80 characters in length.
core_null	integer or real	Core values greater than or equal the interpreted value of this number are undefined (no-data). For floating point cores, (ie. 4 byte real) the null value is typically 0.99999999e38 where the "e38" indicates engineering notation where the mantissa is multiplied by ten to the 38^{th} power. Integer null values are close to the maximum for a given data type.
core_base	integer or real	Core values may be offset by a constant or a base value typically to reduce the number of significant figures in the stored values. The units of the base value are the **same** as the true value.
core_multiplier	integer or real	Core values may be scaled to a desired range by a multiplier. The relation of the true value to the value stored in the qube is: true_value = core_multiplier * stored_value + core_base.
byte_order	literal	The order in which the bytes of multi-byte numbers (eg. 2 and 4 byte integer, 4 and 8 byte real) are stored in a file. Byte_order can have a value of either MSB or LSB (ie. most_significant_byte or least_significant_byte) and indicates the byte ordering when more specific information in the core_item_type is not given. Not a PDS standard keyword.

ODDF posting record keywords

A posting record is a specific form of tabular (point) data that typically includes a station or line id, a geographic position and an application dependent number of measured or derived quantities.

These keywords, map projection description and table description are contained in an "object = express_series", "end_object" attribute pair.

keyword	**value_type**	**description**
logical_record_type	literal	Current value is "posting" which indicates a logical record that starts with an optional 8 character id and has one or more single precision real data channels. No other station or point type records have been

		defined with this release. Not a PDS standard keyword.
logical_record_bytes	integer	The length of the record in bytes. For ODDF posting records this is always a multiple of 4. Not a PDS standard keyword.
id_bytes	integer	The number of id bytes in a posting record is either 0 or 8. Not a PDS standard keyword.
profile_id_bytes	integer	If id_bytes equals 8 (ie. it exists) then 0 to 8 of these characters may comprise the line or profile id and the remaining characters may be application defined. Not a PDS standard keyword.
profile_id_start_byte	integer	If the profile_id exists (ie. profile_id_bytes is greater than 0), then the profile_id starts with this byte. ODDF currently limits this value to 1. Not a PDS standard keyword.
byte_order	literal	The order in which the bytes of multi-byte numbers (eg. 2 and 4 byte integer, 4 and 8 byte real) are stored in a file. Byte_order can have a value of either MSB or LSB (ie. most_significant_byte or least_significant_byte). In the case of the posting file record, the entire file record is written from a 4 byte integer buffer, meaning each group of 4 bytes in the record has either MSB or LSB ordering. Not a PDS standard keyword.

PDS table keywords

A series of data is normally very similar to a plaintext columnar table; ie. a limited number of items are repeatedly recorded. ODDF series are currently limited to the binary form, however, the PDS table and column keywords suffice for both table and series.

The following keywords and column object descriptions are contained in an "object = table", "end_object" attribute pair.

keyword	**value_type**	**description**
interchange_format	literal	Describes the encoding of the table data. ODDF is currently limited to binary tables. Example: interchange_format = binary.
name	character	Title of the data set. Example: name = "Aeromagnetic data"
columns	integer	Number of columns in the table. For ODDF posting type records the id when present is column 1 and the first data channel would then appear as column 2.
rows	integer	Number of rows in the table. For ODDF posting type files this would be the number of logical records and is considered to be optional.

PDS column keywords

A column description consists of the following keywords delimited with an "object = column", "end_object" attribute pair. An example column description would be:

```
object = column
  name = "magnetic field"
  start_byte = 9
  bytes = 4
  data_type = real
end_object
```

keyword	value_type	description
name	character	Title of the column up to 80 characters in length.
unit	character	The data units of the numeric values in this column. Length up to 40 characters. Example: unit = "nTesla".
data_type	literal	Data type for each column, either character or real for ODDF posting type records. Example: data_type = real.
start_byte	integer	The position of the starting byte for a column. For ODDF posting type records with an 8 character id, the first data channel would then have: start_byte = 9.
bytes	integer	The number of bytes in the column. For ODDF posting type records the id has bytes = 8, and for the data channels, bytes = 4.
format	character	The format gives the printing style of the column and desired number of significant figures. Fields are defined by Fortran style description (eg. format = "(e16.8)" for a 16 character field in engineering notation with 8 significant digits, example -0.12345678e-03). Not used for ODDF binary posting files.
exclude	real	Channel values greater than or equal to the interpreted value of this number are undefined (no-data). Typically 0.9999999e38 (10^38) is used for ODDF posting files.
base	real	The interpreted value of this number defines an offset added to the stored value to get the true value. Base is in the **same** units as the true value.
factor	real	The interpreted value of this number defines a multiplier used to scale the stored values to a desired range. The relation between the true value of the column and the stored value is: true_value = stored_value * factor + base.

Map projection keywords

The map projection (or more generally the mapping coordinate system) is a description that applies to the data object, typically either grid, line or point data near the Earth's surface. The units for values in the description are kilometers or decimal degrees where appropriate. The basic keywords can encompass a great number of descriptions with a minimum of embedded comments and hopefully no external documentation.

The version 1.2 dictionary adds datum and ellipsoid keywords to more completely define the coordinate system. The ellipsoid keyword is meant to aid the user, but the axis lengths have priority when computations are performed. See the map projection chapter for details.

The map_projection_type attribute is always present in the map_projection object description, all other attributes are optional depending on the specific coordinate system.

The following keywords and any text are contained in an "object = map_projection", "end_object" attribute pair.

keyword	value_type	description
map_projection_type	character	String that contains the map projection name (or more generally the name of the mapping coordinate system) used in the data object. Example: map_projection_type = "transverse mercator" or map_projection_type = "geographic".
map_projection_desc	character	Text string for the verbose description of the coordinate system. Quoted strings inside map_projection_desc are enclosed in apostrophes (single quotes).
reference_longitude	real	Longitude in degrees that intersects the east-west coordinate axis at zero map units.
reference_latitude	real	Latitude in degrees that intersects the south-north coordinate axis at zero map units. The intersection of the reference_longitude and reference_latitude can be defined as the mapping system origin; coordinate 0,0.
first_standard_parallel	real	The southern-most latitude in degrees that has a scale factor of 1 (true scale). Typically used to define the conic projections.
second_standard_parallel	real	The northern-most latitude in degrees that has a scale factor of 1 (true scale). Typically used to define the conic projections.
a_axis_radius	real	Length of the semi-major axis in kilometers of the reference ellipse of revolution (typically the equatorial radius for an oblate ellipsoid like the Earth). A_axis_radius is always present when a map projection is described and is optional for geographic coordinates.
b_axis_radius	real	Length of the intermediate axis in kilometers. B_axis_radius is redundant when used with an ellipsoid of revolution (all current Earth reference ellipsoids) and has a value equal to a_axis_radius.

c_axis_radius	real	Length of the semi-minor axis in kilometers of the reference ellipse of revolution. The three axis lengths may be set equal to each other when the projection is based on a sphere.

The following keywords are non-PDS standard.

map_projection_unit	character	String describing the system of units used in the data object. Example: map_projection_unit = "kilometers".
vertical_datum	character	String describing the vertical datum. Example: vertical_datum = "NAVD29". The vertical datum description is optional.
horizontal_datum	character	String describing the horizontal datum. Example: horizontal_datum = "NAD83". The horizontal datum description is optional.
ellipsoid	character	String enclosed in quotes describing the reference ellipsoid. Example: ellipsoid = "Clarke 1866" or ellipsoid = "WGS84". An ellipsoid name is optional and has secondary priority when the axis lengths are included in the description (ie. computations are done using axis lengths).
false_easting	real	Offset in kilometers added to the **true** X coordinate of a projection. Used with the universal transverse mercator (UTM) projection to cause all coordinates to be positive or alternately to reduce the significant figures in a stored coordinate. Either the posting record base or false_easting may be used, but not both in any one posting file.
false_northing	real	Offset in kilometers added to the **true** Y coordinate of a projection. Usage is similar to false_easting.
true_scale_latitude	real	Latitude in degrees that has a scale factor of 1 (true scale). Typically used to define projections that are not conic (eg. polar stereographic).
perspective_distance	real	The viewpoint distance from a reference plane in kilometers used to create a perspective projection.
center_scale_factor	real	Used to define the transverse mercator projections. Typically is less than 1 to balance the scaling errors over the width of the projected area (the western and eastern edges then have a scale factor greater than 1. The UTM projection, for instance, uses a center_scale_factor = .9996.

The center line of an oblique mercator projection is the projection of a great circle onto the Y axis of a mercator projection. The center line and coordinate origin may be defined either with a reference latitude and pair of longitude/latitude great circle points or with a reference longitude/latitude pair and an angle the great circle has in relation to north.

center_line_first_longitude	real	Used with the oblique transverse mercator projection, a component in degrees of the reference great circle.
center_line_first_latitude	real	Used with the oblique transverse mercator projection, a component in degrees of the reference great circle.
center_line_second_longitude	real	Used with the oblique transverse mercator projection, a component in degrees of the reference great circle.
center_line_second_latitude	real	Used with the oblique transverse mercator projection, a component in degrees of the reference great circle. When the center line is defined with a longitude/latitude pair, the reference latitude is required and the reference longitude is implicit.
center_line_azimuth	real	Used with the oblique transverse mercator projection, the angle in degrees east of north of the center line. The center line azimuth is given at the reference longitude/latitude and the center line longitude/latitude pairs are not used.

Subroutine Package GridIO

Read and Write Grid Data

Introduction

The ODDF structure for grids consists of direct access binary records with one record for each row of the grid. There are several plaintext header records and optionally several history records preceding the set of grid record. The header consists of ASCII characters and follows the NASA PDS (Planetary Data System) conventions for ISIS qube objects (Martin, et.al, 1988). The ISIS qube was designed for multi-planed spectral images where one image plane might contain a single band of light frequencies. There is provision for many more dimensions, numerous types of ordering of grid points, and coordinate rotations. This release of ODDF 1.6 is limited to a single plane (two dimensions) of either integer or floating point data with no side plane or band suffixes.

For our purposes, a grid is a measured or derived quantity sampled at regular intervals in two user selected dimensions, typically easting and northing for earth science data. The 2-dimensional coordinate system units may be general as in the case where a mathematical function is gridded, or for data registered to the Earth's surface, coordinates may be given in either a standard map projection or longitude/latitude (in this order). In all cases, X is the first coordinate defined and the X axis on maps is identified as a generally west-to-east line with Y coordinate equal zero. The Y coordinate is defined next and when the interval between rows is positive the rows are ordered from south to north. The grid start location is referenced to the origin of the X/Y coordinate system and when both X and Y sampling intervals are positive the grid start location appears to the lower-left of the display.

Gridded values are defined in a **nodal or point** sense where the value stored in the grid file is defined only at the X/Y coordinates of the node, and the grid is undefined between grid nodes. This definition is appropriate for many types of data and mathematical functions and with a priori assumptions, application programs may be extend this definition to either continuous signals like gravity field anomalies where smooth interpolation is appropriate or to step functions like a Geographic Information System (GIS) classification. The application program may use either interpretation, but in general a value in a grid is a point located at the X/Y coordinates implicit from its position in the grid file.

The maximum number of columns in a grid is currently limited to 16,000. The number of rows is unlimited except for consideration of the amount of space available in the disk file system.

Historical usage of the grid file and cautionary note

A grid file (circa early 1970's) was in use prior to the ODDF system and a number of programs were written over the years to use it. Some of the access styles built into GridIO reflect this prior usage and attempt to make the two functionally equivalent where possible. The early grid file was sequential binary, where the first record was a header similar to the one used with subroutine GH1F4 (see below), and subsequent records were rows of the grid. The sequential format of the grid limited interactive display and as the grids became larger access times went up dramatically. The fundamental difference between the early file and the ODDF grid is that the latter is direct access.

Over the years, the early grid had map projection and other information added to the header, but not all the possible information (eg. reference spheroid, standard parallels, etc) was allocated a place and indeed the binary encoded placement of an index or value at a given location would not allow any flexibility once the format was

established. With the addition of more information, in several stages, the edifice became hard for the application programmer to use.

The GridIO access presented below returns to the basic header and leaves open the possibility for specialty use through the addition of new access subroutines. The PDS/ISIS grid is designed to allow for the addition of new data descriptions, but of course, any new functionality (eg. coordinate system rotations, multiple data planes, etc) might not be compatible with existing programs. Careful system design should allow for the use of new information but be transparent to older implementations. The definition of the coordinate system provides an example; when the descriptions are updated to include a horizontal datum (eg. NAD83), application programs do not have to be recompiled because the information is contained in an unparsed text string that is inherited by subsequent files.

Undefined grid nodes

Grid nodes that do not have an assigned value (no data) are flagged with a large positive number. The **no-data** value for real*4 type grids is 10^38 or larger and 999,999,999 or larger for integer type grids. This value is encoded in the plaintext header using the PDS keyword CORE_NULL.

Direct access mode

An existing grid file may be opened with **mode='write'** and the grid rows read and/or rewritten in any order. If **mode='read'** is used, then the rows may be read in any order but the file is write-locked.

Functionality note

These routines are designed to emulate the historical implementations of the sequential grid and is similar to that released by Cordell and others, 1992 and Phillips, 1997. When the need arises for rotations, multiple data planes, 3 or more dimensions, and the like; new subroutines will be designed. All the routines defined in this document are frozen in terms of their look and feel to an application program.

Software dependencies

The entire ODDF system with exception of the map projection calculation and projection object I/O must be available.

SUBROUTINE USAGE GUIDE

The ODDF system is used for both GridIO and PostIO and the system as a whole must be initialized before any subsequent usage. To initialize ODDF, insert a call to PFINIT (Pds File INITialize). The syntax is:

```
character  progid*12
call pfinit( progid )
```

where the string PROGID describes the program about to execute. The call is made only once and its effect is to completely erase the contents of the ODDF system data objects, not only GridIO information but PostIO, ASK, FileVersion and several internal data objects. The best place for this call is in the application driver as one of the first executable statements.

These subroutines are valid for file units 1 to 99, however, units 1 to 4 are reserved for ODDF system use, and 5 and 6 are the traditional Fortran standard input and standard output (ie. terminal I/O); leaving units 7 to 99 for application use. As many grids may be open as needed by the application. The GOPEN routine attaches a disk file to a file unit number and sets up the record pointers. Subsequent I/O is directed to the disk file with the file unit number.

Some routines have a read or write operator at the beginning of the parameter list. The meaning of this read/write is constant throughout ODDF: 'read' is get information from the system, 'write' is put information into the system. The application should **never** read/write directly to an ODDf data file.

The error return parameter should be tested only for zero or not-zero. Zero is interpreted as 'ok' and not-zero as 'not-ok'. The calling routine determines the meaning of 'ok'.

Some of the subroutines listed have parameters which are either **supplied** or **returned** (S/R). In general, when the subroutine is 'writing' you supply a value and when 'reading' the variable is returned with an updated value. Be aware that a variable (as opposed to an explicit value) must be present in the argument list in a position where the system is going to return a value (the program cannot overwrite an explicit value and the program will crash).

The ordering of the arguments in the subroutine calls follows the ODDF pattern of:
read or write operator, file unit number, data object and error flag.

Subroutine list

gopen	Open a grid file.
gclose	Close the file.
gh1f4	I/O the grid header (type 1 grid, 2 axes, unrotated, real*4 core values.)
gh1i4	I/O a grid header for integer*4 core values.
growf4	I/O a real*4 grid row.
growi4	I/O an integer*4 grid row.

PROGRAM USAGE: GOPEN

Open a file unit for grid input/output. Unlike the PostIO drivers, the open statement may proceed the header statement reflecting historical usage. The filename supplied to GOPEN is processed through the FileVersion subroutine package (chapter 7) so that a new file is created with a filename that has an appended version number larger than other filenames by the same name in a directory (the version number is an appended colon and 2 digit number up to 20, eg. test.grd:05) and an old file by default is opened as the largest one available. Because the grid is direct access, a new file is not actually opened until the header routine (either GH1F4 or GH1I4) is called to establish the record length via the number of columns.

programming example

call gopen(iunit, filen, stat, mode, ierror)

explanation

name	type	supplied/returned	description
IUNIT	integer	S	File unit number in the range 1 to 99.
FILEN	char*(*)	S	Filename, must be less than or equal to 80 characters in length.
STAT	char*(*)	S	Either 'new' or 'old'. The first character will suffice.
MODE	char*(*)	S	I/O mode any of: 'read', 'write' or ' ' (blank). 'read' - readonly with a software imposed writelock. 'write' - indicates a file that might have been modified. The default for new files is 'write'. The use of this mode is encouraged when it is known in advance the file will be modified. ' ' - undefined mode; the calling program accepts some responsibility for ensuring proper access. For 'new' files the default mode is 'write'.
IERROR	integer	R	Returned error parameter equals zero if everything is ok. Not equal to zero is there is a problem. The routine is verbose incase of error.

PROGRAM USAGE: GCLOSE

Close a file unit that was opened with GOPEN. This routine **must** be called to allow ODDF to adjust the contents of the plaintext headers after a new grid has been written.

programming example

call gclose(iunit, stat)

explanation

name	type	supplied/returned	description
IUNIT	integer	S	File unit number.
STAT	char*(*)	S	Closing status either 'keep', 'delete' or ' '. Blank is an implied 'keep'. Delete is used for scratch files that are deleted when closed.

PROGRAM USAGE: GH1F4

I/O the basic grid header information. The name is short for Grid_Header_type1_Float_4, where 'float_4' indicates the core of the grid is comprised of single precision floating point (real*4) values. Use subroutine GROWF4 to I/O the grid rows. Map projection information is inherited from an input file and programming access is deferred to the map projection system. The call to GH1F4 may precede or follow the call to GOPEN.

programming example

call gh1f4(mode, iunit, title, ncol, nrow, xstart, deltax, ystart, deltay, ierror)

explanation

name	type	supplied/returned	description
MODE	char*(*)	S	Mode is either 'r' or 'w'.
IUNIT	integer	S	File unit number, range 1 to 99.
TITLE	char*(*)	S/R	Grid title. Not more than 80 characters in length.
NCOL	integer	S/R	Number of columns in the grid. Must be greater than zero and less than or equal to 16,000.
NROW	integer	S/R	Number of rows in the grid. Must be greater than zero.
XSTART	real*4	S/R	First column coordinate in relation to a user defined system like a map projection.
DELTAX	real*4	S/R	Spacing of columns, cannot equal zero.
YSTART	real*4	S/R	First row coordinate.
DELTAY	real*4	S/R	Spacing of rows, cannot equal zero.
IERROR	integer	R	Returned error parameter, zero is ok.

comments

XSTART, YSTART, DELTAX AND DELTAY are assumed to be in the same units and coordinate system.

The current release does not contain parameters to allow the grid columns and rows to be rotated with respect to the reference coordinate system.

The header routine should be called immediately adjacent to the grid file open statement so the interpretation of the plaintext may be stored for the file unit. Opening another grid file before calling the header routine for the first will result in incorrect-access error messages later.

PROGRAM USAGE: GH1I4

Similar to GH1F4 except that 4 byte integer values are stored in the core. The subroutine syntax is the same as GH1F4. Use subroutine GROWI4 to I/O the grid rows.

PROGRAM USAGE: GROWF4

Access a grid row using a single precision floating point (real*4) array. Old grids may be read and/or updated (rewritten) in a direct access fashion. New grids should be written sequentially starting with row one to avoid possible record reordering on some machines.

programming example

call growf4(mode, iunit, ithrow, row, ncol, ierror)

explanation

name	type	supplied/returned	description
MODE	char*(*)	S	Mode is either 'r'ead or 'w'rite.
IUNIT	integer	S	File unit number.
ITHROW	integer	S/R	Row to be accessed in range 0 to NROW. If equal zero the routine will perform the read or write to the next row (from the current row) and set ITHROW equal the new current value.
ROW	real*4	S/R	Array containing one row of the grid.
NCOL	integer	S	Length of ROW. NCOL must equal the NCOL from the call to GH1F4 or the routine returns an error.
IERROR	integer	R	Error parameter equals zero when row accessed properly.

PROGRAM USAGE: GROWI4

Same as GROWF4, except with 4 byte integer row array IROW.

programming example

call growi4(mode, iunit, ithrow, irow, ncol, ierror)

GridIO programming example

The complete and functional program below converts a file of grid values to a file of x,y,z coordinate triples. See the PostIO chapter for more information on point data I/O.

Several of the subroutine calls are followed by example tests of the error-state variable. The error test, throughout ODDF, is always whether an integer variable (eg. IERROR) is zero or not-equal to zero. For brevity in this program listing, only three tests are shown.

```
c Program grd2xyz.

      parameter   ( maxcol=5000, maxrow=maxcol )
      dimension   x(maxcol), y(maxrow), row(maxcol), xyz(3)
      character   title*80, id*8

c Call pfinit only once in a program.

      call pfinit( 'test_prog' )


c Open the grid and read the header into this application.  REMEMBER: If
c the grid contains a map projection object it will be inherited by the
c output post file.

      call gopen( 10, 'test.grd', 'old', 'read', ierror )

      if ( ierror .ne. 0 ) then
        print *, ' Error opening Input grid.'
        go to 999
      endif

      call gh1f4( 'read', 10,   title,   ncol, nrow,
     1            xs, dx, ys, dy,   ierror  )

      if ( ierror .ne. 0 ) then
        print *, ' Error reading grid header.'
        go to 999
      endif


c Open the output posting type file.  In this simple example the
c header routines are unnecessary, but the call to XIOPH1 explicitly
c stores the basic information to describe the posting record.
c Remember the posting description must be written to ODDF before the
c file is opened.

      nchan  = 3
      nidb   = 8
      npidb  = 0
      ispidb = 1
```

```
      call xioph1( 'write', 11, nchan, nidb, npidb, ispidb, ierror )

      call xopen( 11, 'test.pst', 'new', 'write', ierror )

      if ( ierror .ne. 0 ) then
        print *, ' Error opening output posting file.'
        go to 999
      endif

c  Setup the x and y coordinate arrays.

      do i = 1, ncol
        x(i) = xs + dx * float ( i - 1 )
      enddo
      do j = 1, nrow
        y(i) = ys + dy * float ( j - 1 )
      enddo


c  For this simple example ID is blank.  It could encode the grid row
c  so a profile based program could access by 'profile', in this case
c  the original rows (remember to change NPIDB).

      id = ' '

c  Cycle the grid rows.
c  Note: It's always bad practice to send a do-index into a subroutine,
c  this example sends a temporary variable 'jrow'.

      do  100  j = 1, nrow

        jrow = j
        call growf4( 'read', 10, jrow,   row, ncol,   ierror )

        do 50 i = 1, ncol

          xyz(1) = x(i)
          xyz(2) = y(j)
          xyz(3) = row(i)
          call xiopst( 'write', 11,  id,  xyz, nchan,  ierror )

50      continue
100   continue

      call gclose( 10, 'keep' )
      call xclose( 11, 'keep' )

999   stop
      end
```

Subroutine Package PostIO

Read and Write Point and Line Data

Introduction

The posting record is a station or single point logical record and consists of an 8 character (byte) id and an application determined number of single precision (4 byte real) data channels up to a maximum of 1026. The id is optional but when present is 8 bytes long. A line of stations (flightline or profile) can be constructed by assigning a given number of characters in the station id to be the line_id. A sequence of stations with the same line_id may then be processed as either line data or, since there is no change of data structure, as point data. In addition, an application program may interpret a given data channel to be a line attribute.

The user or the application program determines the interpretation of the data channels, although commonly, the location of the station is stored as an X/Y pair in channels one and two. The X/Y pair may be general coordinates, geographic longitude and latitude, or map projected coordinates. If the coordinate system is geographic or a map projection, then a plaintext description will appear in the file header. The interpretation of channels is aided by the PDS table description in the header. Each table column description contains title, units, precision, scaling and offset as well as position and field width in the record. The title of a data channel may be considered part of the interpretation and can convey information to a person unfamiliar with a particular data file.

The posting data file is direct access and unformatted (binary encoded) that has multiple records of plaintext description preceding the data records. The plaintext header contains ASCII text and formatting characters and is therefore printable on your screen without special software. The posting data is encoded in binary for rapid processing and therefore appears on your screen as random symbols. Multiple posting records are contained in each direct access file record so that a single read of the data file returns many posting records, which speeds processing.

Some of the data object description (metadata) can be inherited by new files without action by the application program. Currently, only information contained in the plaintext header can be inherited. Map projection information, for instance, is always inherited, even in the case of mixed data types such as when point data is converted to grid data. Other inherited information is the line id specification, providing the output is also posting type. Channel titles are not inherited and must be transferred by the application program.

Historical usage of the posting record

A posting file (circa 1978) was in use prior to the ODDF system and a number programs were written over the years to use it. Some of the access styles built into the PostIO package reflect this prior usage and attempts to make the two functionally equivalent where possible. The early posting file was sequential binary with one posting record per file record. There were no header or trailer records to give ancillary information. This posting record consisted of an 8 character id, X and Y position channels , and 6 data channels assigned to gravity anomaly data.

Later usage assigned the first 4 characters of the id to be the flightline id and the 6 data channels to specific aeromagnetic anomaly data. Next, the number and contents of all channels were made application defined, which made the data record more useful but more uncertain as to content. Later still, as the number of data points per file increased, multiple posting records were bundled together with several dimension values into a single direct access record. This last structure was termed the express file because it decreased access time dramatically. The routines below have names that begin with 'x' and many date from a late 1980's I/O package

by the author that evolved into ODDF.

Programming contributors to the evolution of the posting file (anonymously, in alphabetic order and apologies to any overlooked contributors/contributions) were Robert Bracken, Richard Godson, Robert Simpson and Michael Webring; all with the then Branch of Geophysics, USGS.

Recognition of legacy file types

The PostIO system does not recognize the historical sequential posting file, however external conversion programs are normally available and are relatively easy to write. The so-called express file is recognized for input (indeed the data object in an ODDF posting file is comprised of express records) but without the metadata in the header record, there is no map projection description and some assumptions as to posting record layout have to be made by the system.

Existent vs. nonexistent posting files

For the purposes of this document a 'new' file is one the system is about to create and an 'old' file is one the system has created **and** closed. See the FileVersion chapter for more information. Direct access of posting records is available only for old files.

No-data value

Posting records may have channels that contain 'no-data', such as when one continuously recording instrument has a power failure and other instruments continue to function. The channels that do not have a data value are flagged with a large positive number. The no-data value for 4-byte-real type values is 10^38 or larger. This value is encoded in the plaintext header table column description using the PDS attribute "EXCLUDE = 1.0e38".

Some of the routines described below have integer arguments (eg. NPIDB NumberOfProfileIDBytes) that may have a no-data value. The (4-byte or longword) integer no-data value is 999,999,999 or larger.

Point vs. line access

Individual posting records are read and written with the XIOPST routine described below and is the fundamental access routine. There is no limit to the number of stations in a file except for the physical storage available on a given machine. For historical reasons (functional compatibility with pre-ODDF programs using a sequential posting file), the XIOPST routine does not contain a record number argument and therefore is essentially a sequential access routine. The current-record, however, may be assigned for old files allowing direct access to any posting record.

A line of stations may be accessed with the XIOPRF routine with several caveats. The primary caution is that a line can be any number of stations greater than one and therefore long lines with thousands of points and hundreds of channels can easily exceed the storage capacity of most hardware. There is also a limit of 5000 lines per file because the catalog is stored internally as a data object. Several possible access modes are built

into the XIOPRF routine to handle extreme cases, but do not guarantee access to all the information that may be associated with a profile.

Point and line catalog files

When direct access of old files is desired, point and/or line catalog files are generated by the system. The catalog files (the filenames are prefixed with the data filename and end with "cat") serve as lookup tables that link stations or lines with specific file records. The catalog filenames do not contain a version number (see the FileVersion chapter) and therefore a given posting file has the choice of only one catalog. If the catalog is incorrect for the particular posting file being used, then it is automatically recreated at the time the parent file is opened. Both types of catalog may be deleted by the user when no longer needed.

The point catalog file is binary and not printable by the user. Each open posting file may have an open point catalog file, so several posting files can use point-wise direct access at the same time. The line catalog file is text and contains not only the line id and number of points in the line, but statistics on the X/Y coordinate extent of the line. The format of the line catalog file is detailed at the end of this chapter. Only one line catalog is held internally by the ODDF system, reflecting the early use of the catalog as an aid to the user rather than a general purpose processing tool.

Sequential vs. direct access

A new data file is created by sequentially writing either posting records or lines of posting records (in the usage guide these are routines XIOPST and XIOPRF). An old file is read sequentially by default with both routines.

Direct access of lines of data in old files is a matter of specifying a line id when using XIOPRF, while direct access of points is a two step process. Point data may be directly accessed by positioning the current record pointer with XIOPSN and then calling XIOPST. The recommended method to read or update potentially long lines of stations is to obtain the record locations via the line catalog, use XIOPSN once and then use calls to XIOPST to sequentially build the required channels into a profile. The line catalog information is available through the XCAT series of routines.

Direct access of existing files

An old posting file may be opened with **mode='write'** or **mode='update'** (see the XOPEN description) and the records read and/or rewritten in any order. Write mode would typically be used to sequentially write new data into given records (and is the default mode when creating a new file). Update mode causes a write-after-read to rewrite (update) a given record without a re-positioning step, and is convenient for corrections or the writing of values derived from the current record. If **mode='read'** is used, then any record may be read but the file is software write-locked.

Software dependencies

The entire ODDF system with the exception of the map projection calculation and projection object I/O must be available.

Disclaimer

The desired behavior of each routine is stated in the descriptions or comments. Any behavior not stated must be considered subject to change without notice. Undocumented behavior is usually an oversight by the author and future versions of the system may plug the hole or elevate the oversight to a feature. Changes in the **documented** behavior will hopefully be transparent or upward compatible to existing application programs.

General programming considerations

The ODDF system is used for both PostIO and GridIO and the system as a whole must be initialized before any subsequent usage. To initialize ODDF, insert a call to PFINIT (Pds File INITialize). The syntax is:

```
character  progid*12
call pfinit( progid )
```

where the string PROGID describes the program about to execute. The call is made only once and its effect is to completely erase the contents of the ODDF system data objects, not only Postio information but GridIO, ASK, FileVersion and several internal data objects. The best place for this call is in the application driver as one of the first executable statements.

These subroutines are valid for file unit numbers 1 to 99, although units 1 to 4 are reserved for ODDF system use and 5 and 6 are the traditional Fortran standard input and standard output (ie. terminal I/O). The XOPEN routine attaches a disk file to a file unit number and sets up the record buffers and pointers. Subsequent I/O is directed to the disk file with the file unit number.

Up to eight posting files can be open at once, each with a maximum record length of 64000 bytes. Currently ODDF creates file records that are 4096 bytes in length, but will automatically increase this length when the number of channels exceed 50.

Some routines have a read or write operator at the beginning of the parameter list. The meaning of this read/write is consistent throughout ODDF: 'Read' is get information from the system, 'Write' is put information into the system. The application should **never** read/write directly to an ODDF data file.

The error return parameter should be tested only for zero or not-zero. Zero is interpreted as 'ok' and not-zero as 'not-ok'. The calling routine determines the meaning of ok. For instance, during a sequential posting record read a not-ok error return usually means End-Of-File (which is actually ok in the sense that the input is complete).

Some of the subroutines listed have parameters that are either **supplied** or **returned** (S/R). In general, when the subroutine is 'writing' you supply a value and when 'reading' the variable is returned with an updated value. Be aware that a variable (as opposed to an explicit value) must be present in the argument list in a position where the system is going to return a value (the program cannot overwrite an explicit value and the program will crash).

The ordering of the arguments in the subroutine calls follows the ODDF pattern of:
read or write operator, file unit number, data object and error return.

Subroutine list

These routines are all named Xaaaaa. The 'x' indicates single point or single station type data. There are currently two major sub categories: XIOaaa and XCATaa for I/O operations and line catalog operations.

--file control--

xopen Open a file for posting record I/O.

xclose Close the file and update the plaintext header.

--header routines (data description of the post record)--

xioph1 I/O of the post logical record specifications.

xioph2 I/O of the data set title and several character strings describing each data channel.

--single record routines--

xiopst Posting record access to the next record.

xiopsn Reposition the file so the next XIOPST read/write does I/O to the specified logical record.

xiobak Backs up the current-record pointer by one record.

xiorew 'Rewind' the file so the next XIOPST read/write does I/O to the first posting record.

--profile (line) oriented routines--

xioprf Profile access where the id of post records indicates the logical grouping of records into lines.

xachan Assign channels to be accessed by XIOPRF.

xascan Assign start and stop scans (posting records) to be accessed by XIOPRF.

--profile catalog routines--

xcatb Profile catalog, brief.

xcatlr Profile catalog, logical-record. Return an array which points to the posting record number for the start of each line.

xcatxs Profile catalog with X/Y statistics for each line.

PROGRAM USAGE: XOPEN

Open a posting file for input/output. When the file is **old** then header information like map projection and posting record specification is parsed and stored as an object description. When the file being opened is **new** then the posting record specification and any existing map projection description information is written into the file header **at the time XOPEN is called**. Because the posting file is direct access and the header information can be extensive, the header routines XIOPH1 and optionally XIOPH2 need to be called before XOPEN so that file structure and the header record can be defined.

The filename supplied to XOPEN is processed through the FileVersion subroutine package (chapter 7) so that a new file is created with a filename that has an appended version number larger than other filenames by the same name in a directory (the version number is an appended colon and 2 digit number up to 20, eg. test.pst:05) and an old file by default is opened as the largest one available.

Up to eight posting files may be open at a time. The RECORD_BYTES keyword in the file header gives the physical or file record length. Currently **new** files are opened with RECORD_BYTES=4096.

programming example

call xopen(iunit, filen, stat, mode, ierror)

explanation

name	type	supplied/returned	description
IUNIT	integer	S	File unit number in range 1 to 99. Fortran usage limits this range to 7 to 99.
FILEN	char*(*)	S	Filename, less than or equal to 80 characters.
STAT	char*(*)	S	Open status must be either 'new' or 'old'. The first letter is recognized.
MODE	char*(*)	S	Any of (the first letter is recognized): 'Read' for old files. The file is write-locked and the program may be interrupted by the user without damage. 'Write'. This is the mode for a file being written or otherwise modified. This is the default mode for 'new' files. XCLOSE must be called if writing did occur. Interruption of a program using write mode can leave the file in an indeterminate state. 'Update'. Read, then optional rewrite of the same record. Interruption of a program using update mode can leave the file in an indeterminate state. XCLOSE must be called. ' ' blanks - not recommended. The application would have to do some bookkeeping operations the system as a whole

does automatically.

IERROR	integer	R	Returned error code equals zero when file opened correctly. The routine is verbose when the file is not opened correctly and IERROR is returned as not-zero.

PROGRAM USAGE: XCLOSE

Close a posting file and de-allocate ODDF pointers and buffer space. This routine flushes logical record buffers for write and update modes, updates some data descriptions in the file header, then closes the file unit.

This routine must be called if a file is **new**.

programming example

call xclose(iunit, stat)

explanation

name	type	supplied/returned	description
IUNIT	integer	S	File unit number.
STAT	char*(*)	S	Either blank, 'keep', or 'delete'. Blank is an implied keep. Use delete for scratch files.

Header routines

The header routines allow the application program to setup a posting record or to read the length and interpretation of a posting record. Use the XIOPH1 and XIOPH2 routines with **mode='write'** only once per new file. Do not call these routines with **mode='write'** for old files.

The order in which files are opened and header read or written is important:

1) When reading **old** files, open the old file with XOPEN then 'read' the header information from ODDF into your application.

2) When writing **new** files, 'write' the information to ODDF using XIOPH1 and XIOPH2 then open the file with XOPEN. In this manner the header information is known to ODDF before it opens the file and constructs the plaintext header. The header can be somewhat verbose so the bulk of the information must be known in advance if the system is to avoid the problem of extending the header after data records have been written.

PROGRAM USAGE: XIOPH1

Read or write information about the layout of the posting record. Note that when MODE equals 'write' then you supply the information listed as 'supplied/returned'. The reverse applies when MODE equals 'read'. XIOPH1 must be called before XOPEN when writing a new file so the posting record layout and length are known before the plaintext label is created.

programming example

call xioph1(mode, iunit, nchan, nidb, npidb, ispidb, ierror)

explanation

name	type	supplied/returned	description
MODE	char*(*)	S	Mode is 'read' or 'write', the first character is recognized.
IUNIT	integer	S	File unit number. Range 1 to 99.
NCHAN	integer	S/R	Number of real*4 data channels. Range 1 to 1026.
NIDB	integer	S/R	Number of id bytes (characters). Write: may be 0 **or** 8. Read: either 0 or 8. If the routine cannot determine the number of id bytes from the file, then 8 bytes is assigned if the logical record contains 4 or more longwords and assigns 0 for records less than or equal to 3 longwords. The number of channels is then adjusted accordingly to match the interpreted presence of the record id.
NPIDB	integer	S/R	Number of profile (line) id bytes. Write: may be 1 **to** 8 inclusive or no-data when NIDB equals 8 or; zero or no-data when NIDB equals 0. Read: same as for write. Zero means no profile id and no-data means no information. The system may assign the default NPIDB=4 if asked to do profile operations in the presence of a NPIDB equal to no-data.
ISPIDB	integer	S/R	Start byte of the profile id. Provided as an upgrade option, currently the only valid values are 1 or no-data.
IERROR	integer	R	Returned error code. The routine is verbose in case of an error.

PROGRAM USAGE: XIOPH2

Read and write information about the interpretation of the post record. The use of this routine is optional. None of the information is inherited and the application program must keep track of the 'interpretation' of the data channels. When writing, use this routine no more than once per new file.

For compatibility with more general record types (eg. ASCII tables) the entries are referred to as columns in the tabular form sense. For post records this means the id when present is column number 1, and then data channels are columns 2 through NCHAN+1. The PDS descriptions processed through this routine are limited to the ones that make sense for a binary table (ie. TYPE, FORMAT, START_BYTE etc. are done for you).

The relation between FACTOR and BASE is: true_value = stored_value * FACTOR + BASE (ie. BASE is in the same units as true_value). Note that routines XIOPST AND XIOPRF (documented below) read and write the 'stored_value' and the application program completes the interpretation of scaling via factor and base.

programming example

```
parameter       ( mxc=101 )
character       tbname*80, name*80,   unit*40,   null*40,   factor*40,   base*40
common /qaz1/   tbname,    name(mxc), unit(mxc), null(mxc), factor(mxc), base(mxc)
. . .
maxcol = mxc
call xioph2( mode, iunit,  tbname,  ncol, nrow,  maxcol,
&    name, unit, null, factor, base,  ierror )
```

explanation

name	type	supplied/returned	description
MODE	char*(*)	S	Either 'read' or 'write' (first character recognized).
IUNIT	integer	S	File unit number.
TBNAME	char*80	S/R	Title of the data set (table). Must be less than or equal 80 characters.
NCOL	integer	S/R	Number of columns in the table arrays. Read: NCOL = NCHAN + { 0 or 1 } depending on the presence of posting_id. In case of error, NCOL = no-data. Write: Either NCOL as defined for READ or no-data (>= 999,999,999) in which case NCOL will be derived from the XIOPH1 data object. Routine is verbose in case of error.
NROW	integer	S/R	Number of rows in the table. NROW is provided as an upgrade option. Current returned value is no-data.
MAXCOL	integer	S	Physical dimensioned size of the next five arrays: NAME, UNIT, NULL, FACTOR and BASE.

NAME	char*80	S/R	Array of column titles.
UNIT	char*40	S/R	Array of units descriptions, eg. "kilometers", "mGal". Blank for no information.
NULL	char*40	S/R	Array of null values. Blank for no information (defaults to "1.0e38" for real*4 channels).
FACTOR	char*40	S/R	Array of multiplicative factors. Blank for no information (defaults to "1").
BASE	char*40	S/R	Array of column base offsets in the units specified by UNIT. Blank for no information (defaults to "0"). The relation between FACTOR and BASE is: true_value = (stored_value * FACTOR) + BASE
IERROR	integer	R	Returned error parameter, zero is ok. An error on read may not be fatal to an application since it only indicates the lack of interpretative information.

comments

There exists the possibility of incorrect usage when combined with XIOPH1 (ie. NCOL is defined implicitly by the XIOPH1 parameters NCHAN and NIDB). The suggested order for application programs to call these two routines is XIOPH1 then XIOPH2.

PROGRAM USAGE: XIOPST

Read/write posting logical records. Profile and station data are considered to be logically equivalent by XIOPST (ie. it does not interpret the id). This routine sequentially accesses records by performing the I/O operation on the **next** record (in reference to the current position). This is the basic I/O routine and the one most likely used in an application.

The application must read or write the entire post record.

programming example

call xiopst(mode, iunit, id, chnval, nchan, ierror)

explanation

name	type	supplied/returned	description
MODE	char*(*)	S	Mode is either 'read' or 'write' to the file unit number IUNIT that has been opened with XOPEN.
IUNIT	integer	S	File unit number in range 1 to 99.
ID	char*8	S/R	Id of the posting logical record. The variable ID must be length 8, but whether or not it is actually used depends on the value of NIDB passed through XIOPH1.
CHNVAL	real*4	S/R	Array of single precision data.
NCHAN	integer	S/R	Length of CHNVAL. NCHAN should passed as a variable so the routine can return a corrected value in case of error. For instance passing NCHAN=0 to an existing file (during read) results in the routine returning the actual number of channels.
IERROR	integer	R	Error equals zero when the I/O operation was successful. Routine is silent at EOF during reads and sets IERROR not zero. The routine prints verbose messages if there is an inconsistent setup (eg. The program attempts to change the number of channels)

PROGRAM USAGE: XIOPSN

Resets the current record pointer for an **old** file so the next call to XIOPST will read or write the desired record. The routine also returns the current logical record number so a program can confirm the position of the last read or write operation. A catalog file is generated by this routine and the filename is the data filename with the last component changed to "ptcat" (eg. file test.dat:04 has a catalog file named test.ptcat without a version number). The catalog is in binary (ie. it's unprintable) and stays open as long as the parent file remains open. The catalog filename has no version number and therefore over-writes any preexisting catalog file of the same name.

programming example

call xiopsn(mode, iunit, logrec, ierror)

explanation

name	type	supplied/returned	description
MODE	char*(*)	S	Mode is either 'read' or 'write'.
IUNIT	integer	S	File unit number in the range 1 to 99.
LOGREC	integer	S/R	Posting (logical) record number. During MODE='write', assigns the logical record number so the **next** call to XIOPST reads or writes at LOGREC. During MODE='read', returns the logical record number of the presumed last I/O operation (see comments).
IERROR	integer	R	Returned error parameter is not equal zero in case either positioning or LOGREC is incorrect. The routine is verbose. If LOGREC is supplied as greater than 999,999,998; then it is assumed a record count is desired so the routine is silent but IERROR remains not-zero.

comments

The XIOPSN routine positions the file pointers for the presumed I/O operation to follow and since it does no data I/O, the posting data pointer is not modified until either XIOPST or XIOPRF is called. The result is that an XIOPSN 'write' will not show up in an XIOPSN 'read' until the record **has been accessed** (with XIOPST for instance). An XIOPSN read immediately after an XIOPSN write to record 100 would return a record pointer of 99 until XIOPST has been called to either read or write record 100.

In case the supplied LOGREC is less than or equal to zero, the routine resets LOGREC to 1. In the case where the supplied LOGREC exceeds the number of logical records in the file, LOGREC is reset to the maximum number of logical records in the file (this is an easy way the get the number of records in a file).

This routine does a search of the point catalog (which is stored in a binary data file) to find which file record contains LOGREC. The routine is therefore doing I/O to disk and should not be called to move forward or back

too many times if overall execution speed is important.

PROGRAM USAGE: XIOBAC

This routine is the logical equivalent to the Fortran backspace command and used in conjunction with a record oriented routine like XIOPST. Use XIOBAC only with **old** files.

programming example

call xiobac(iunit, ierror)

explanation

name	type	supplied/returned	description
IUNIT	integer	S	File unit number to be backspaced.
IERROR	integer	R	Returned error parameter.

comments

The following code fragment reads a posting record, moves the current record pointer back one record, and then re-writes the same posting record. Note that **update** mode (see XOPEN) is equivalent to the following fragment but without the XIOBAC statement.

call xiopst('r', 10, . . .
call xiobac(10, ierror)
call xiopst('w', 10, . . .

PROGRAM USAGE: XIOREW

This routine is the logical equivalent to the Fortran rewind command and is used in conjunction with a record oriented routine like XIOPST. The logical record pointer is set to the beginning of the file so the next XIOPST call accesses the first logical record. Use XIOREW only with **old** files.

programming example

call xiorew(iunit, ierror)

explanation

name	type	supplied/returned	description
IUNIT	integer	S	File unit number to be set at record one.
IERROR	integer	R	Returned error parameter.

PROGRAM USAGE: XIOPRF

Read and write lines (profiles) consisting of posting records. For **new** files, the routine sequentially writes the line data; no catalog is generated. For **old** files, the lines may be selected by LINEID in direct access mode. A line catalog is generated automatically for access of old files and the catalog data object may be read by the application program using the XCATaa routines.

The application program supplies storage space sufficient to hold the desired number of channels and points. The programmer needs to know the usage of logical vs. physical array dimensions and be aware of the limitations declared array space impose on general applications.

programming example

```
parameter   ( maxp=10000, mxch=100)
character   id*8
dimension   chnval(maxp,mxch)
. . .
maxpt   = maxp
maxchn = mxch
call xioprf( mode, iunit,   lineid,   npt, nchan,   chnval,   maxpt, maxchn,   nwnnw, ierror )
```

explanation

name	type	supplied/returned	description
MODE	char*(*)	S	Mode is either 'read' or 'write'.
IUNIT	integer	S	File unit number in range 1 to 99.
LINEID	char*8	S/R	Id of the current profile, declared length must be 8. The number of characters assigned to be the LINEID is the PROFILE_ID_BYTES keyword in the data file or the NPIDB argument to the XIOPH1 routine. The remaining characters in the posting record id are ignored. While reading, if LINEID is blank the routine will access the next profile and set LINEID to the current value .
NPT	integer	S/R	Number of points in the CHNVAL array.
NCHAN	integer	S/R	Number of channels in CHNVAL. NPT and NCHAN are the logical dimensions of array CHNVAL.
CHNVAL	real*4	S/R	Single precision array of data with physical dimensions CHNVAL(MAXPT,MAXCHN), where the first dimension varies fastest.

MAXPT	integer	S	First dimension of the CHNVAL physical array.
MAXCHN	integer	S	Second dimension of the CHNVAL physical array. The physical array size is the **space** available for the profile data. NPT and NCHAN indicate how much data exists in the two dimensions.
NWNNW	integer	S/R	Winnowing parameter. An NWNNW = 3 causes every third profile point to be stored in CHNVAL. NWNNW also serves as a CHNVAL array overflow indicator.
IERROR	integer	R	IERROR equals zero if the profile access is ok. During reads, the routine is generally silent and the return of an incomplete profile is indicated with the NWNNW argument.

comments

There are several behind the scenes functions related to this routine.

1) LINEID is passed as 8 characters but the number actually used to define a line or profile is set elsewhere in the system (see XIOPH1). The default number of characters which define a line are four and start at character one.

2) When reading a file sequentially one line at a time, the LINEID can be set to blanks before the XIOPRF call and the routine will return the next line in the file. When it does this, it puts the current line_id into LINEID.

3) This routine calls up the catalog for old files and uses this to access any profile. You must have a unique lineid for each profile. When writing a new file with this routine each call should be made with a different lineid or else the system will not re-access the file correctly. Check routines XCATXS/LR for access to the profile catalog.

4) The channels read into or written from array CHNVAL are set using routine XACHAN (assign_channel). If XACHAN is not called, the XIOPRF routine defaults to the total number of channels available.

5) The start/stop points (scan numbers) may be set with XASCAN (assign_scan) before calling XIOPRF to select the portion of interest. The start/stop scan numbers define a data window that is applied before the winnowing factor.

6) The winnowing parameter NWNNW is used for several functions during input; **this parameter is ignored during output**. Set this parameter to a desired valued before each call to XIOPRF since it is possibly updated with each access.

Definition of NWNNW
During writes, NWNNW is ignored. For reading, different values of NWNNW cause different access (a negative NWNNW indicates the system is making changes to the current profile):

NWNNW = 0. If the profile is too long to fit into the physical dimension MAXPT; the profile is **truncated** to fit and NWNNW is changed to the negative of the winnowing factor which would let the profile fit.

NWNNW > 0. Winnows at the desired sampling and if the profile does not fit truncates the profile and sets NWNNW as above.

NWNNW < 0 (automatic mode). Winnows at the positive value of NWNNW unless the resulting array is too long in which case the routine returns the whole profile winnowed the minimum amount that allows it to fit into CHNVAL. NWNNW is then set to the negative of the current winnowing factor.

Line utility routines

The XACHAN and XASCAN routines allow the application program to reduce the number of data values the XIOPRF CHNVAL array has to accommodate. This reduction is useful when perusing or displaying selected portions of a data file.

PROGRAM USAGE: XACHAN

Assign channels to be used during the XIOPRF read operation.

programming example

call xachan(mode, iunit, ichan, nchan, ierror)

explanation

name	type	supplied/returned	description
MODE	char*(*)	S	Mode is either 'write' or 'read'.
IUNIT	integer	S	File unit number associated with this array assignment. File must have been opened with XOPEN.
ICHAN	integer	S/R	Array of channel numbers that XIOPRF uses.
NCHAN	integer	S/R	Length of ICHAN.
IERROR	integer	R	Returned error parameter not equal to zero indicates an error.

PROGRAM USAGE: XASCAN

Assign the start and stop points (scan numbers) to be used during XIOPRF operations. Scans are the equivalent of post logical records and indicate the process of scanning all the instruments to produce a data point.

programming example

call xascan(mode, iunit, isscan, iescan, ierror)

explanation

name	type	supplied/returned	description
MODE	char*(*)	S	Mode is either 'write' or 'read'.
IUNIT	integer	S	File unit number associated with this assignment. File must have been opened with XOPEN.
ISSCAN	integer	S/R	Starting scan in the profile about to be read.
IESCAN	integer	S/R	Ending scan in the profile. The accessing routines take care of inconsistences for you (ie. negative or zero number of scans defaults to the whole profile, greater than current profile length returns the whole profile).
IERROR	integer	R	Not equal to zero indicates an error condition.

Line catalog routines

There are several variations of the catalog routines, the only difference being how much information is passed to the calling program.

Only one catalog at a time is stored by the system. Each time an application calls for a catalog for a different file unit (eg. you are reading profiles with XIOPRF from 2 different files) the system must read the ASCII catalog file. An example line catalog file is at the end of this chapter.

These routines generate catalogs as required and the user may delete the catalogs as desired.

PROGRAM USAGE: XCATB

This routine returns a brief catalog of line ids and the number of points in each line.

programming example

```
parameter       ( mxl=500 )
character       lineid*8
common /qaz1    lineid(mxl),
common /qaz2/   ndata(mxl)
. . .
maxlin = mxl
call xcatb( iunit, maxlin,  npidb, nline,  lineid, ndata,  ierror )
```

explanation

name	type	supplied/returned	description
IUNIT	integer	S	File unit number.
MAXLIN	integer	S	Physical dimension of LINEID and NDATA.
NPIDB	integer	R	Number of characters which define the profile id.
NLINE	integer	R	Number of lines in the file. NLINE is the number of entries in LINEID and NDATA.
LINEID	char*8	R	Array of profile ids. Physical length is MAXLIN.
NDATA	integer	R	Array of the number of points in each profile. Physical length is MAXLIN
IERROR	integer	R	Not equal to zero indicates an error condition.

PROGRAM USAGE: XCATLR

Similar to the brief catalog returned by XCATB with the addition of the posting (logical) record number of the first point in each profile.

programming example

call xcatlr(iunit, maxlin, npidb, nline, lineid, ndata, islogr, ierror)

explanation

IUNIT, MAXLIN, NPIDB, NLINE, LINEID, NDATA and IERROR are the same as for XCATB.

name	type	supplied/returned	description
ISLOGR	integer	R	Array of logical record numbers. Physical length of array is MAXLIN

comments

Profile based programs which must deal with a large number of data channels and/or a large number of points per profile can use the XCATLR catalog routine to obtain the starting posting record and the number of points (ISLOGR and NDATA) in each line. Then the XIOPSN and XIOPST routines can access the profile data one posting record at a time. In this fashion the arrays necessary can be very much smaller than if the I/O was done with XIOPRF.

PROGRAM USAGE: XCATXS

Similar to the brief catalog returned by XCATB with the addition of the X and Y coordinate arrays for each profile. Each profile has a minimum, maximum, average and standard deviation for both X and Y included in the line catalog to make display in cartographic form more efficient. X and Y are assumed to be in channels one and two.

programming example

```
parameter   ( mxl=500 )
character   lineid*8(mxl)
dimension   ndata(mxl),  xstat(4,mxl),   ystat(4,mxl)
. . .
maxlin = mxl
call xcatxs( iunit, maxlin ,  npidb, nline,  lineid, ndata, xstat, ystat,  ierror )
```

explanation

IUNIT, MAXLIN, NPIDB, NLINE, LINEID, NDATA and IERROR are the same as for XCATB.

name	type	supplied/returned	description
XSTAT	float	R	Array of profile statistics taken from the first channel of data in the posting record. The physical dimensions are: XSTAT(4,MAXLIN) and the 4 entries per each line are: minimum, maximum, average and standard deviation.
YSTAT	float	R	Array of statistics from the 2nd channel. Dimensions and contents the same as for XSTAT.

Posting I/O programming example

The program below is complete and functional. It reads formatted flight-line data and outputs a sequence of posting records.

It is preferable that the ID contains no embedded blanks and is left-justified to facilitate later retrieval by the user.

Several of the subroutine calls are followed by example tests of the error-state variable. The error test, throughout ODDF, is always whether an integer variable (eg. IERROR) is zero or not-equal to zero. For brevity in this program listing, only two tests are shown.

```
c-------------------------------------------------------
c program to convert outside data to ODDF.

      dimension z(20)
      character id*8

c  initialize ODDF, must precede any ODDF call.

      call pfinit( 'outside2post' )

c  Open formatted input file (see FileVersion for description).
c  Note the error test is only for zero or not-zero.

      call opnfmt( 10, 'fbn.asc', 'old', ierror )

      if ( ierror .ne. 0 ) then
        print *, ' Input file is not open.'
        stop
      endif

c  write profile id info into the system.

      nchan  = 20
      nidb   = 8
      npidb  = 6
      ispidb = 1

      call xioph1( 'write', 11, nchan, nidb, npidb, ispidb, ierror )

c  open output file.

      call xopen( 11, 'fbn.pst', 'new', 'write', ierror )

c  read data records, write to ODDF system.

      do 10 i = 1, 9999999

        read( 10, 123, end=20 ) id, z
123     format( a8, 20f12.4 )
```

```
      call xiopst( 'w', 11,  id,  z, nchan,  ierror )

      if ( ierror .ne. 0 ) then
        print *, ' Error writing posting record =', i
        go to 20
      endif

10    continue

c  close the output file, xclose must be called for 'new' files.

20    call xclose( 11, 'keep' )

      stop
      end
```

Line catalog file

Below is a line catalog file derived from a posting data file. The catalog is generated by the ODDF system as needed and resides in the user's current directory (not necessarily where the parent file is located).

example catalog file

```
%%028    %%
   1167      2     13      1     56      1
 -.91281166E+02 -.91279633E+02 -.91280365E+02  .51476288E-03
  .46612457E+02  .46676262E+02  .46644188E+02  .18293871E-01
%%027    %%
   4009     13     53     57     25   1168
 -.91271492E+02 -.91269669E+02 -.91270332E+02  .41081500E-03
  .45981190E+02  .46180656E+02  .46081032E+02  .57437863E-01
%%027B   %%
   2340     53     76     26     42   5177
 -.91270805E+02 -.91269821E+02 -.91270393E+02  .25482129E-03
  .46181103E+02  .46301170E+02  .46242161E+02  .34536913E-01
%%027C   %%
    680     76     83     43     15   7517
 -.91271469E+02 -.91269829E+02 -.91270576E+02  .51481492E-03
  .46301449E+02  .46336529E+02  .46318989E+02  .10159779E-01
%%026    %%
   1306     83     96     16      8   8197
 -.91260826E+02 -.91259155E+02 -.91259827E+02  .37382406E-03
  .46586266E+02  .46658833E+02  .46622509E+02  .21046048E-01
%%end____%%
```

end example file

explanation

Each profile entry has four text lines:

1) Profile id delimited with "%%".
2) Indices for:
   number of points in the profile,
   start and stop file records,
   start and stop logical records in the respective file records,
   posting record number for the first point in the profile.
3) XY statistics:
   X coordinate statistics: minimum, maximum, average and standard deviation,
   Y coordinate statistics.

Detailed Breakdown of Lines 028 and 027

Note: one logical record of a physical file record is a posting record.

The paired percent characters delimit 8 characters and the id, as it exists in the file, is mapped into this space. For instance, %% nn01 %%, indicates that the lineid is not left-justified but actually contains 2 blank characters (since the id is constrained to begin at character 1). This might cause problems for the user that types in "nn01" because the id is literally " nn01".

```
%%028    %%
   1167     2     13      1     56      1
  ndata    phys_rec       log_rec    start_post_record
```

1) There are 1167 points in the line.
2) It starts with physical record 2, 1st logical record (remember the header is in physical record 1).
3) It ends with physical record 13 , 56th logical record.
4) The profile begins with the 1st posting record in the file.

```
%%027    %%
   4009     13     53     57     25    1168
  ndata    phys_rec       log_rec    start_post_record
```

1) there are 4009 points in line 027.
2) It starts with physical record 13, 57th logical record (immediately following the last point of line 028).
3) It ends with physical record 53, 25th logical record.
4) The profile begins with the 1168th posting record in the file (immediately following the last point of line 028).

Application programs are concerned with the number of points in the profile and where they start in the file (NDATA and ISLOGR in routine XCATLR). The start and stop physical and internal logical records are mentioned here for completeness.

Map Projections

Introduction

The map projection subroutine package (PRJSYS) described later in this section is a driver for the NOAA/USGS National Mapping Division, General Cartographic Transformation Package (GCTP) (USGS, 1986). The purpose of PRJSYS is to provide an easy to use interface to GCTP and when combined with the projection I/O subsystem (PRJIO) provides for complete self-documentation. Because of the variety of projections available and a multitude of setups possible, the routines are designed to allow the programmer to take defaults, change common parameters and progress to specialized setups by adding subroutine calls.

PRJSYS was designed to facilitate two operations that were difficult in the past. The first is the complete documentation of the projection. As the Global Positioning System (GPS) becomes more prevalent with the acquisition of new data, the coordinates will be specified in NAD83 (Dewhurst, 1990). The change of horizontal datums from NAD27 should not require any modification of existing programs; when the subroutines and keywords are defined in ODDF/PRJSYS, the appropriate information appear in the plaintext description to be inherited by successive daughter files. Events have overtaken this author and the definition of horizontal datum has been included to the system as version 1.2 of the keyword dictionary.

The second operation affects the programmer by allowing an almost document free setup of projections. While extensive help messages are not part of PRJSYS, a dynamic method of defining projections is. Projections are specified by name in PRJSYS and the use of arbitrary codes is eliminated. The system allows the programmer to inquire as to available defaults and adjustable parameters. Within the limits of this author's interpretation of possible problems, the system either silently recovers or verbosely warns of errors. The design goal is that once the syntax of a few subroutines is known (and combined with some basic knowledge), then the design of a projection proceeds without reference to external documentation.

PRJSYS does not interact with the user. A higher level query routine is included at the end of this section that can serve as a template for the setup of general projections. Also included is an example of the case where a processing stream needs a specific projection without user intervention. Whenever a projection is correctly specified and activated, the GCTP package prints the current information to the user's screen. Note that if the data file is ODDF compatible, the projection, if any, is part of the data inheritance so very few programs need to setup projections. The only programs which need even the the basic attachment and forward/inverse capability are those which actually use the projection to plot tick marks, compute coordinates and the like.

Limitation

The units of measure for all coordinate values and appropriate computational parameters supplied and returned from the projection system are decimal degrees and kilometers.

Map projection list

The formal names below are generally given as person, primary property and developable surface. The MPXNAM (Map Projection indeX to NAMes) routine will accept a partial name and return the full formal name. The MPHNAM(Map Projection Help with NAMes) routine will print this list to the user's screen.

Capitals indicate the minimum characters necessary for a match

ALBERS equal area conic
AZImuthal equidistant
EQUIDIStant conic
EQUIRECtangular
GENeral vertical near side perspective
GEOgraphic
GNOmonic
LAMBERT CONformal conic
LAMBERT AZImuthal equal area
MERcator
MILler cylindrical
OBLIQUE MERcator
ORTHOgraphic
POLAR stereographic
POLYconic
SINusoidal equal area
STATE plane coordinates
STEREOgraphic
TRANSverse mercator
UNIVersal transverse mercator
VAN der grinten

Projection defaults

All of the projections have default parameters. Without going into exhaustive detail the lists below summarize the default and adjustable parameters for each projection.

The default ellipsoid is WGS84 which is the basis of the NAD83 (North American Datum) coordinate system. The ellipsoid has semi-major axis = 6378.137 km, semi-minor axis = 6356.7523 km (Synder, 1987). The default sphere has a radius of 6370.997 kilometers and has the same surface area as the Clarke 1866 ellipsoid. All projections that default to an ellipsoid may be modified to be based on a sphere, projections based on a sphere are mathematically limited to spheres (at least in the current implementation).

The default reference longitude and latitude is 0, 0 degrees for most projections and will need to be modified in almost every case to bring the coordinate system origin closer to a study area.

The default false easting and northing is 0, 0 kilometers for all projections except the universal transverse mercator. Every projection accepts a false easting and northing.

There is currently no provision for ad-hoc rotation of coordinates.

Specialized parameters are generally preset for the continental U.S.

albers equal area conic
elliptical
default standard parallels are 29.5 and 45.5 degrees.

azimuthal equidistant
spherical

equidistant conic
elliptical
default standard parallels are 33 and 45 degrees.

equirectangular
spherical
reference latitude is limited to 0 degrees
default true scale latitude is 39 degrees.

general vertical near side perspective
spherical
default height of perspective is 500 kilometers.

geographic
This is the non-projection complement to the others in this list. Input or output is longitude and latitude.

gnomonic
spherical

lambert azimuthal equal area
spherical

lambert conformal conic
elliptical
default standard parallels are 33 and 45 degrees.

mercator
elliptical
reference latitude is limited to 0 degrees
default true scale latitude is 39 degrees.

miller cylindrical
spherical
reference latitude is limited to 0 degrees.

oblique transverse mercator
elliptical
reference longitude is set via the center line parameters
default center scale factor is .9996
default center line is -98 degrees longitude and 33 and 45 degrees latitude.

orthographic

spherical

polar stereographic
elliptical
requires reference longitude and 90 degrees north or south reference latitude
default true scale latitude is 80 degrees north or south.

polyconic
elliptical

sinusoidal equal area
spherical
reference latitude is limited to 0 degrees.

state plane coordinates
not available with this release.

stereographic
spherical

transverse mercator
elliptical
default scale factor is .9996.

universal transverse mercator
ellipsoid may need to be changed for different regions on the globe
reference longitude set by zone.
The following parameters are not adjustable:
reference latitude is the equator
scaling factor = .9996
false easting = 500 km
false northing = 10,000 km

van der grinten
spherical
reference latitude is limited to 0 degrees.

Overview of projection inheritance and transfer

ODDF controls I/O to the data file and the ODDF file-header buffer retains only the last projection description encountered. The projection description in ODDF is modified in either of two ways:

1) Open an 'old' file with GOPEN or XOPEN (previously described grid or point type data) or,
2) Write a projection using POIOB (described below).

The projection description in ODDF is automatically written to the header of any new file opened via GOPEN or XOPEN, this is the actual inheritance step.

examples

Of particular importance as seen in the examples below is that the specifics of a projection (name, reference ellipsoid, standard parallels, etc) are transparent to the application program. Once a projection is set up, the application refers to it by switch number. Example number 3 is the only one where any projection parameter is mentioned.

1) A normal inheritance takes place when you open an old point or grid data file and then a new grid or point file.

```
call xopen( 10, 'test.xyz_data', 'old', 'read', ierror )
. . .
call gopen( 11, 'test.grid', 'new', 'write', ierror )
```

At the time the grid file is opened it receives a copy of the current map projection object. Note the projection libraries lib_prjsys.a and lib_prjio.a are not used because no calculation or verification is being done.

2) To make the map projection in a file available for calculations, read it into the projection system by assigning it a switch number. Activating a switch causes the GCTP routines to print current parameters to the screen and alerts the user that a projection is available.

```
iunit = 10
call xopen( iunit, 'test.xyz_data', 'old', 'read', ierror )
iswt = 1
call poiob( 'read', iswt, ierror )
. . .
deglon = -105.0d0
deglat  = 40.0d0
call mpfwd( iswt, deglon, deglat,  xkm, ykm,  ierror )
```

After the call to MPFWD (map projection forward), XKM and YKM contain a coordinate pair in whatever projection was stored in ISWT. Both 'lib_prjsys.a' and 'lib_prjio.a' (UNIX compiled subroutine archives or similar) must be loaded into the application program.

3) To define any projection and write into ODDF, call the set_name routine (MPSNAM), modify the default parameters as needed and then write the resulting switch number to the system. A projection name " " (blank) or "null" has much the same meaning (ie. no_information), but null causes the projection name to be written explicitly as null and blanks causes the project object itself to not be transferred to an output file.

```
iswt = 1
prjnam = 'geographic'
call mpsnam( 'write', iswt, prjnam, nchnam, ierror )
...
call poiob( 'write', iswt, ierror )
. . .
call gopen( 11, 'test.grid', 'new', 'write', ierror )
```

Any valid projection name is ok in MPSNAM and this is the basic way to define a map projection if one is not available from a data file. All projections have a default setup generally suitable for the continental U.S. and additional setup routines may be called to modify the projection (eg. calling the MPSSTP sets the standard parallels in the Lambert conformal conic and other appropriate conic projections). The IACTPJ routine given below is an interactive setup routine that is very general and is useful where a user sets up the projection.

Subroutine Package PRJIO

The projection data object subroutines, all named a variation of POaaaa, provide the connection between ODDF file I/O and the map projection (MPaaaa) subroutines. Their basic function is to transfer a projection description between the ODDF file header buffer and a switch number in the calculation package. See the diagram in the introductory chapter for a graphic illustration of the relationship.

Software Dependencies

PRJIO calls PRJSYS and ODDF and so lib_prjsys.a and lib_oddf.a (UNIX object module libraries) must be loaded into the application.

PROGRAM USAGE: POINIT

Initialize projection object system. Include this call before any POaaaa call. Note POINIT calls MPINIT (explained in the next section).

programming example

call poinit

There are no parameters.

PROGRAM USAGE: POIOB

Either read a projection description from the ODDF file-header buffer into a switch number in the projection calculation system (PRJSYS) or the reverse. When reading, the PRJIO system parses the description into projection parameters and activates the projection by performing a test calculation. When writing, the PRJIO system constructs a description, using the current dictionary, from the parameters in the desired switch and writes the description to the ODDF file-header buffer.

programming example

call poiob(mode, iswt, ierror)

explanation

name	type	supplied/returned	description
MODE	char*(*)	S	Either 'read' or 'write'. The system only recognizes the first character.
ISWT	integer	S	Projection switch in the range 1 to 4.

IERROR	integer	S	Returned error parameter is either zero or not-zero (ok or not-ok).

comments

Conceptually, POIOB operates the same as a Fortran read or write. During a read, the routine accesses the projection description which ODDF stored when the file was opened and parses this description into the map projection parameters under the given switch. At this point the projection is available to all the MPaaaa routines. For write the reverse is true.

Since the ODDF file-header buffer retains only the **last** map projection encountered, POIOB should be called to read the buffer before another file is opened or write to the buffer immediately before a file is opened.

ODDF versions lower than 1.6 used a routine named POIO that contained a file unit number in the calling statement. The routine remains available as originally written where the file unit parameter is an upgrade option that has not been activated.

Subroutine Package PRJSYS

Map Projection Computations

All the following routines, named a variation of MPaaaa, are independent of the data file and are only concerned with the set up and calculation of map projections using the GCTP package.

There may be 4 projections defined but not all may be active (depending on the specific projections). Each projection is assigned a logical **switch** number in the range 1 to 4. The geographic 'non-projection' is switch 5 and is always active. Once a projection is attached to a switch and initialized, the program refers to the projection by its number. See for instance the routine MPFWD.

GCTP is written in double precision and as long as pathologies are avoided, conversion between projections can be accomplished with approximately 10 to 14 significant figures. PRJSYS uses double precision also and defers the question of precision to the external system which processes and stores data.

Note in all the routines described below that both input and output coordinates are denoted by X and Y. It is implicit that geographic coordinates are also considered to be X (easting or longitude) and Y (northing or latitude). The units are either kilometers or decimal degrees. Longitude is negative west of the Greenwich meridian and latitude is negative south of the equator.

Programmer Guides

The detailed guides are broken into groups:

Basic projection routines	The initialization and interactive setup.
Computation routines	Forward, inverse and direct change computations.
Utility routines	Help with projection, datum and ellipsoid names.
Setup routines	Basic projection set, projection activation, general modification.
Descriptive character strings	Datum and ellipsoid by name.
Functional parameter setup	Modifications to specific groups of computational parameters.

Software Dependencies

The MP routines call the GCTP and GenChar subroutine packages. The GCTP package from National Mapping Division, USGS is included with the PRJSYS routines. The GenChar package is a subsystem of ODDF and is included in that library, but can also be compiled as a standalone subsystem.

Subroutine list

Basic routines

mpinit	Initialize the projection system routines named MPaaaa and the GCTP package.
iactpj	Interactive setup of a map projection.

Computation Routines

mpfwd	Forward projection of degrees to kilometers.
mpinv	Inverse projection of kilometers to degrees.
mpchng	Change kilometers from one projection to another.

PROGRAM USAGE: MPINIT

Initialize the projection system. Include this call before any MPaaaa call. Note that POINIT calls MPINIT.

programming example

call mpinit

There are no arguments.

PROGRAM USAGE : IACTPJ

This routine guides the interactive user through the setup of a projection. The source code can serve as a template for programmers.

programming example

call iactpj(iswt, kboard, ierror)

explanation

name	type	supplied/returned	description
ISWT	integer	S	Projection switch in the range 1 to 4.
KBOARD	integer	R	Keyboard response code. A KBOARD=-2 means soft abort and the user wants to back up. See the ASK subroutines chapter more details.
IERROR	integer	R	Error return, zero is ok, not-zero means the projection setup is invalid.

PROGRAM USAGE: MPFWD

Forward projection from geographic to projected coordinates.

programming example

call mpfwd(iswt, dplon, dplat, dpx, dpy, ierror)

explanation

name	type	supplied/returned	description
ISWT	integer	S	Projection switch range 1 to 4.
DPLON	double	S	Longitude in units of degrees.
DPLAT	double	S	Latitude in units of degrees.
DPX	double	R	X projected coordinate in kilometers.
DPY	double	R	Y projected coordinate in kilometers.
IERROR	integer	R	Error parameter, zero means DPX and DPY are ok. In case of error IERROR is not-zero and DPX,DPY are set to a no-data value of 10^38.

PROGRAM USAGE: MPINV

Inverse projection from projected to geographic coordinates.

programming example

call mpinv(iswt, dpx, dpy, dplon, dplat, ierror)

explanation

name	type	supplied/returned	description
ISWT	integer	S	Projection switch range 1 to 4.
DPX	double	S	X projected coordinate in kilometers.
DPY	double	S	Y projected coordinate in kilometers.
DPLON	double	R	Longitude in units of degrees.
DPLAT	double	R	Latitude in units of degrees.
IERROR	integer	R	Error parameter, zero means DPLON, DPLAT are ok. In case of error IERROR is not-zero and DPLON, DPLAT are set to a no-data value of 10^38.

PROGRAM USAGE: MPCHNG

Change coordinates from one projection to another. Either switch may be attached to the geographic 'non-projection' (making the routine similar to the function of MPFWD and MPINV). The projection types attached to ISWT and JSWT must be different in the current implementation. For example the Lambert conformal conic cannot be attached to both input and output even though they differ by some parameter like the reference longitude. The same restriction applies to the transverse mercator based projections (transverse mercator, UTM and oblique transverse mercator).

programming example

call mpchng(iswt, xin, yin, jswt, xout, yout, ierror)

explanation

name	type	supplied/returned	description
ISWT	integer	S	Switch number for the input projection.
XIN	double	S	Input X coordinate in the units assigned.
YIN	double	S	Input Y coordinate in the units assigned.
JSWT	integer	S	Switch number for the output projection.
XOUT	double	R	Output X coordinate in the units assigned.
YOUT	double	R	Output Y coordinate in the units assigned.
IERROR	integer	R	Error equals zero when ok, if there was an error, then IERROR not equal zero and the output coordinates are set to 10^38.

comments

Note in all the routines described below that both input and output coordinates are denoted by X and Y. It is implicit that geographic coordinates are also considered to be X (easting or longitude) and Y (northing or latitude).

PROGRAM USAGE: MPHNAM

Help with projection names. Prints to the user's screen a listing of projection names. The caps in the listing are the minimum number of characters necessary for comparison with the master list. The list of map projections immediately after the introduction to this chapter was derived from this routine.

programming example

call mphnam

There are no parameters.

PROGRAM USAGE: MPHHDT

Help with horizontal datum. Prints a list of recognized horizontal datum names and a paragraph about usage with ellipsoids.

programming example

call mphhdt

There are no parameters.

Below is an example help message from MPHHDT, the names are fixed but the formatting and content of the message may vary.

Begin help message

Each horizontal datum has a default ellipsoid.
Setting a horizontal datum sets the appropriate
ellipsoid or sphere (depending on the projection),
but for generality the reverse does not occur.
The priority from lowest to highest is: horizontal
datum, ellipsoid name, ellipsoid axis lengths. In
other words axis lengths are most specific and the
ellipsoid name (if any) is secondary.

The following horizontal datums have a default
ellipsoid:

NAD27
NAD83
WGS84
Old Hawaiian

End help message

PROGRAM USAGE: MPHELL

Help with ellipsoids. Prints a table of recognized ellipsoid names, semi-major and minor axis lengths, reciprocal flattening and eccentricity squared.

programming example

call mphell

There are no parameters.

The printed table is given below, spelling of the names is fixed but the exact formatting of the columns may vary.

Begin table:

Predefined ellipsoids in PRJSYS
* indicates a defining parameter.

Name	semimajor	semiminor	1/flattening	eccentricity_sqr
Clarke 1866	6378.2064*	6356.5838*	294.978698214	.006768657997
GRS 1980	6378.137*	6356.7523141	298.257222101	.006694380023
WGS84	6378.137*	6356.7523142	298.257223563	.00669437999
International 1924	6378.388*	6356.9119461	297.*	.006722670022
Sphere	6371.	6371.	------	0.0
Sphere (Clarke 1866)	6370.997	6370.997	------	0.0

End table

PROGRAM USAGE: MPXNAM

Contains two operations; it either takes a partial projection name and returns the MPsys internal index or takes the MPsys index and returns the full projection name. The routine does not reference or modify a projection switch, it only searches the master list of projection names.

programming example

call mpxnam(iopr, mapprj, ndxprj, ierror)

explanation

name	type	supplied/returned	description

IOPR	integer	S	Either 1 or 2 depending on whether the routine should return the first or second argument after IOPR.
MAPPRJ	char*(*)	S/R	Map projection name. If IOPR=1 this parameter is returned.
NDXPRJ	integer	S/R	PRJSYS internal index (this index will change with the addition of new projections). If IOPR=1 this parameter is supplied. Reverse the S/R sense of MAPPRJ and NDXPRJ when IOPR=2.
IERROR	integer	R	error parameter is zero if ok.

comments

The operation of this subroutine can be noted in the example interactive routine IACTPJ. In this routine the user is prompted for a projection name and then MPXNAM is called to see whether the supplied name is recognizable. The operation is then reversed to obtain the full projection name (eg. the user types "albers" and the system returns "albers equal area conic").

Projection setup routines

The following setup routines, denoted MPSaaa, read and write information for functional groups of parameters. All data units passed through the MPSaaa routines are either in **kilometers** or **decimal degrees** and all floating point variables are **double precision**.

subroutine list

Basic setup routines

mpsnam	set projection name
mprdy	activate a projection switch
mpsap1	set all computational parameters
mpsap2	set all descriptive names

Descriptive character strings

mpsell	set ellipsoid name
mpshdt	set horizontal datum name
mpsvdt	set vertical datum name

Computational parameter routines

mpsaxs	set ellipsoid axis lengths
mpsfen	set false easting and northing
mpshei	set perspective height
mpsobq	set oblique mercator projection parameters
mpsref	set reference longitude, latitude
mpssca	set center scale factor for transverse mercator projections
mpsstp	set standard parallels
mpstsl	set true scale latitude

PROGRAM USAGE: MPSNAM

Set the name of the projection. This routine is used to start the setup of a projection. When MODE='write', the switch is initialized to default values for the contintental U.S. When MODE='read' it returns the full name of the projection attached to ISWT.

programming example

call mpsnam(mode, iswt, name, nchnam, ierror)

explanation

name	type	supplied/returned	description
MODE	char*(*)	S	Mode is either 'read' or 'write'.
ISWT	integer	S	Switch number in range 1 to 4.
NAME	char*80	S/R	Either a supplied partial projection name (eg. 'albers' as opposed to 'albers equal area conic') or the returned name of an initialized projection.
NCHNAM	integer	R	Returned number of characters in NAME.
IERROR	integer	R	Returned error, zero is ok. The routine will warn when a supplied name cannot be parsed into a recognizable projection name.

PROGRAM USAGE: MPRDY

Activate a projection by calling the GCTP package with an initialize flag. The GCTP package then responds with a printed message which lists the various active parameters.

programming example

call mprdy(iswt, ierror)

explanation

name	type	supplied/returned	description
ISWT	integer	S	Projection switch in range 1 to 4.
IERROR	integer	R	Returned error parameter. Zero means the projection switch is ready for forward or inverse calculations. Non-zero means there was an error.

PROGRAM USAGE: MPSAP1

Set all parameters (selection 1). When all the parameters are known in advance, this routine may be used to setup (write) a new projection. The inverse operation (read) will return all parameters known to a given switch. This routine can assist the programmer by returning the number of parameters that may be modified with the setup routines.

programming example

```
 call mpsap1( mode, iswt,  lenarr,  pname,  punit,
1           axs, naxs,    ref, nref,   fen, nfen,   stp, nstp,
1           tsl, ntsl,    obq, nobq,   sca, nsca,   hei, nhei,
1           ierror  )
```

explanation

name	type	supplied/returned	description
MODE	char*(*)	S	Either 'read' or 'write'. The first character will suffice.
ISWT	integer	S	Projection switch number in the range 1 to 4.
LENARR	integer	S	Length of the AXS, FEN,...,HEI arrays. Should always equal a length of 4.
PNAME	char*(*)	S/R	Projection name. Normally declared with a length of 80 characters.
PUNIT	char*(*)	S/R	Projection units. Currently this parameter is locked with a value of "kilometers".

The functional arrays are schematic for brevity:

AAA	double	S/R	Double precision array with length 4. A very large number (10^38) in these arrays indicates no-information.
NAAA	integer	S/R	During 'read' this logical length parameter tells either how many elements of AAA are defined or how many elements are required for the given projection.

where a short description of each is:

axs Axis lengths. Semi-major axis length and optional semi-minor axis length in kilometers.
ref Reference longitude, latitude (in that order) in decimal degrees.
fen False easting and northing (in that order) in kilometers.
stp Standard parallels for conic projections (southernmost first) in decimal degrees.
tsl True scale latitude in decimal degrees.
obq The special parameter array for the oblique mercator projection in degrees. The order from 1 to 4 is southernmost longitude/latitude pair, then northernmost longitude/latitude pair of the reference great circle.
sca Center of projection scaling factor for transverse mercator projections (UTM, TM, Oblique M) is a dimension-less ratio.

hei Perspective height in kilometers.

IERROR integer R Error return parameter. Zero is ok, non-zero means not ok.

comments

The functional arrays should always be declared with a length of 4 even though most groups do not use the full amount.

When MODE='read' this routine returns the functional array and the number of active elements in each array (NAXS, NFEN, etc). For 'write' the Naaa variables are ignored and the routine takes what it needs and returns an error if an array is incorrect.

This routine calls MPSAXS, MPSREF, MPSFEN, MPSSTP, MPSTSL, MPSOBQ, MPSSCA and MPSHEI which are described below.

PROGRAM USAGE: MPSAP2

Set all parameters (selection 2). This routine sets the descriptive strings for vertical datum, horizontal datum and ellipsoid name.

When either reading or writing to MPSAP2, each of the descriptive setup routines MPSVDT, MPSHDT, MPSELL is called.

programming example

call mpsnam(mode, iswt, vdatm, hdatm, ellip, ierror)

explanation

name	type	supplied/returned	description
MODE	char*(*)	S	Mode is either 'read' or 'write'.
ISWT	integer	S	Switch number in range 1 to 4.
VDATM	char*80	S/R	Either a supplied or returned vertical datum description.
HDATM	char*80	S/R	Either a supplied or returned horizontal datum description.
ELLIP	char*80	S/R	Either a supplied or returned ellipsoid name.
IERROR	integer	R	Returned error parameter. Zero is ok, non-zero means not ok.

Descriptive strings

The vertical datum and horizontal datum may be written to the description and while the datum descriptions do not affect the map projection as such, they do affect the interpreted positions near the Earth's surface. The datum inclusion in the file header is important because data collected in NAD27 coordinates should not be mixed with the newer NAD83 coordinates. Once old data is converted to the new datum, then it needs to be tagged with the appropriate datum description to avoid mistaken reconversion. Converting coordinates between datums, however, is outside the current scope of ODDF and PRJSYS.

The ellipsoid name is included as an attribute to aid the user who might be reading an ODDF file header. A given ellipsoid name will have unique axis lengths but the inverse is not true, so the MPSELL (set ellipsoid) and the MPSAXS (set axis) routines interact to keep the two representations consistent. Computation of map projections is **always done using the axis lengths** and the system may not always recognize a poorly formed or incorrect ellipsoid name to warn the user.

PROGRAM USAGE: MPSVDT

The vertical datum description is stored as an unparsed string and the system only warns if the string does not exactly match the internal list and then only during a 'write' operation.

programming example

call mpsvdt(mode, iswt, vdatum, nchar, ierror)

Explanation

name	type	supplied/returned	description
MODE	char*(*)	S	Mode is either 'read' or 'write', only the first character is recognized.
ISWT	integer	S	Projection switch in the range 1 to 4.
VDATUM	char*(*)	S/R	Vertical datum name or descriptive string. Must be less than or equal to 80 characters.
NCHAR	integer	R	Returned number of characters in VDATUM when MODE is 'read'. NCHAR is ignored when MODE='write'.
IERROR	integer	R	Error parameter equals zero if ok.

comments

The internal list of vertical datums is preliminary and includes: "NAVD29" and "NAVD88".

PROGRAM USAGE: MPSHDT

The horizontal datum description is stored as an unparsed string and the system only warns if the string does not exactly match the internal list and then only during a 'write' operation.

programming example

call mpshdt(mode, iswt, hdatum, nchar, ierror)

Explanation

name	type	supplied/returned	description
MODE	char*(*)	S	Mode is either 'read' or 'write', only the first character is recognized.
ISWT	integer	S	Projection switch in the range 1 to 4.
HDATUM	char*(*)	S/R	Horizontal datum name or descriptive string. Must be less than or equal to 80 characters.
NCHAR	integer	R	Returned number of characters in HDATUM when MODE is 'read'. NCHAR is ignored when MODE='write'.
IERROR	integer	R	Error parameter equals zero if ok.

comments

The internal list of horizontal datums is preliminary and includes: "NAD27", "NAD83", "WGS84" and "Old Hawaiian".

PROGRAM USAGE: MPSELL

The ellipsoid designation by name is completely optional. The ellipsoid description is an unparsed string that is not checked because of the wide variety of named ellipsoids and their possible spellings. If during a 'write' operation, the given ellipsoid exactly matches a name in the internal list then the axis lengths are set accordingly; see the comments below for more detailed information. This routine is new with ODDF version 1.6 and the interaction between the MPSELL and MPSAXS routines is preliminary, currently the system is set more for flexibility than attempting to rigorously enforce a one-to-one correspondence between names and axis lengths (eg. GRS 1980 and WGS84 are two different names for two almost identical ellipsoids).

programming example

call mpsvdt(mode, iswt, ellip, nchar, ierror)

Explanation

name	type	supplied/returned	description
MODE	char*(*)	S	Mode is either 'read' or 'write', only the first character is recognized.
ISWT	integer	S	Projection switch in the range 1 to 4.
ELLIP	char*(*)	S/R	Ellipsoid name. Must be less than or equal to 80 characters.
NCHAR	integer	R	Returned number of characters in ELLIP when MODE is 'read'. NCHAR is ignored when MODE='write'.
IERROR	integer	R	Error parameter equals zero if ok.

comments

The internal list of ellipsoids is preliminary and includes: "Clarke 1866", "GRS 1980", "WGS84", "International 1924", "Sphere" and "Sphere (Clarke 1866)". The characters in the variable ELLIP must exactly match the ones given here or no action is taken.

There is not only the relation of ellipse name and axis length to consider but also the possibility that a given map projection may be available only in a spherical form (in which case the only currently assigned name is "sphere". Suffice it to say, only the axis length(s) has any meaning to the computation of the map projection and care must be exercised when specifying a non-standard combination.

The most robust way to include the ellipsoid name in the map projection description is to call the MPSAXS routine with the desired axis lengths and allow that subroutine to assign the elliposiod name if one matches. However, the whole point of the MPSELL routine is to reduce errors in specifying axis lengths by allowing textual names.

Some of the branches in the tree can be set out for use of this routine. If the ellipsoid name is not found, then a warning message is printed to the user's screen, but no other action is taken. It is therefore possible to accidently mismatch an ellipsoid name with lengths initialized by MPSNAM or after a call to MPSAXS.

In ODDF version 1.6, the MPSAXS routine will remove (set to blank) an ellipsoid name if the axis lengths do not match an internal set. Therefore two paths are available for the programmer to define these two more-or-less related items:

1) Write an ellipsoid name with MPSELL, which causes:
   A) if the name is recognized, then
      A1) the axis lengths are updated if the projection supports ellipsoids or
      A2) prints a warning message if only spherical forms can be used (no axis change).
   B) a printed warning message but **no** axis change if the name is not recognized.

2) Write a set of axis lengths with MPSAXS, which causes:
   A) an elliposid name update if the lengths are recognized, or
   B) setting the ellipsoid name to blank if the lengths are not recognized.

Computational setup routines

Subroutines: AXS, REF, FEN, STP, TSL, OBQ, SCA, HEI

Setup routines for functional groups of parameters. If your application requires a hardwired projection, you would call the MPSNAM routine to initialize it, then call the particular setup routines necessary to modify it, and finally the MPRDY routine to active it.

The MPSaaa routines correct the supplied information during write and print warning messages only when something is not salvageable. Since the required information varies substantially among the projections, defaults are available for all parameters to aid the user. After writing the information, the user may read it back to see whether an MPSaaa routine had to correct the original setup. A good working knowledge of map projections is useful.

Each of these 8 routines has the same parameter list.

schematic programming example

call mpsaaa(mode, iswt, aaa, naaa, ierror)

where *aaa* is any of: AXS, REF, FEN, STP, TSL, OBQ, SCA or HEI.

particular programming example

call mpsaxs(mode, iswt, axs, naxs, ierror)

explanation

name	type	supplied/returned	description
MODE	char*(*)	S	Mode is either 'read' or 'write'.
ISWT	integer	S	Switch number in range 1 to 4.
AAA	double	S/R	Double precision array of length 4. Unused portions of the array are filled with the no-data value.
NAAA	integer	R	Returned number of parameters required in AAA based on the projection.
IERROR	integer	R	Returned error flag.

comments

Make sure the AAA array contains **either data or no-data** (10^38) values when writing. That is, fill the arrays with no-data and then include the values you want the routine to modify.

PROGRAM USAGE: MPSAXS

Set the semi-major and semi-minor axis lengths for the reference ellipsoid. The default ellipsoid set by MPSNAM is WGS84 and the default sphere is the normalized version of the Clarke 1866. All projections that use an ellipsoid can use a sphere.

programming example

call mpsaxs(mode, iswt, axs, naxs, ierror)

explanation

name	type	supplied/returned	description
MODE	char*(*)	S	Mode is either 'read' or 'write.'.
ISWT	integer	S	Switch number.
AXS	double	S/R	Semi-major followed by optional semi-minor axis in kilometers . Array length is 4. Logical length will be either 1 or 2.
NAXS	integer	R	Returned number of parameters for this projection.
IERROR	integer	R	Returned error parameter, zero is ok.

comments

Reference spheres may be specified by entering a single length (and filling the unused portion of the AXS array with no-data values) or entering the same length twice.

This routine interacts with the ellipsoid name (see also routine MPSELL) to the extent that when writing axis lengths the list of recognized ellipsoid axis lengths is tested (to .1 meter in version 1.6) and if a match is found the name is stored. Those familiar with the slight difference between the GRS 1980 and WGS84 ellipsoids will notice this system favors the latter name. If a match is not found then the ellipsoid name is set to blanks.

Neither the axis nor the ellipsoid routines interact with the horizontal datum because there is no one-to-one correspondence.

PROGRAM USAGE: MPSREF

Set the reference longitude and latitude which then becomes the origin of the map projection coordinate system.

programming example

call mpsref(mode, iswt, ref, nref, ierror)

explanation

name	type	supplied/returned	description
MODE	char*(*)	S	Mode is either 'read' or 'write'.
ISWT	integer	S	Switch number.
REF	double	S/R	Reference longitude and/or latitude in degrees. Array length is 4. Logical length will be either 1 or 2.
NREF	integer	R	Returned number of parameters for this projection.
IERROR	integer	R	Returned error parameter, zero is ok.

comments

The coordinate reference routine MPSREF is one that will correct (override) supplied data. For instance the UTM projection uses the equator as the Y reference and will silently reset a non-zero second element in the REF array.

PROGRAM USAGE: MPSFEN

Set the false easting and northing offsets that are added to the projected coordinates.

programming example

call mpsfen(mode, iswt, fen, nfen, ierror)

explanation

name	type	supplied/returned	description
MODE	char*(*)	S	Mode is either 'read' or 'write'.
ISWT	integer	S	Switch number.
FEN	double	S/R	False easting and northing in kilometers. Array length is 4. Logical length will be either 1 or 2.
NFEN	integer	R	Returned number of parameters for this projection.
IERROR	integer	R	Returned error parameter, zero is ok.

PROGRAM USAGE: MPSSTP

Set the standard parallels for conic projections. Standard parallels are those where the scaling factor is one.

programming example

call mpsstp(mode, iswt, stp, nstp, ierror)

explanation

name	type	supplied/returned	description
MODE	char*(*)	S	Mode is either 'read' or 'write'.
ISWT	integer	S	Switch number.
STP	double	S/R	Standard parallels of latitude in degrees Array length is 4. Logical length will be either 1 or 2.
NSTP	integer	R	Returned number of parameters for this projection.
IERROR	integer	R	Returned error parameter, zero is ok.

comments

The southernmost standard parallel is specified first and the northernmost (if needed) is specified second.

PROGRAM USAGE: MPSTSL

Set the true scale latitude for non-conic projections. A true scale latitude is similar to a standard parallel used with conic projections.

programming example

call mpstsl(mode, iswt, tsl, ntsl, ierror)

explanantion

name	type	supplied/returned	description
MODE	char*(*)	S	Mode is either 'read' or 'write'.
ISWT	integer	S	Switch number.
TSL	double	S/R	True scale latitude in degrees for non-conic projections. Array length is 4. Logical length will be 1.
NTSL	integer	R	Returned number of parameters for this projection.
IERROR	integer	R	Returned error parameter, zero is ok.

PROGRAM USAGE: MPSOBQ

Set the center line for the oblique mercator projection. The center line is defined by two pairs of longitude and latitude coordinates and forms a great circle on the earth.

programming example

call mpsobq(mode, iswt, obq, nobq, ierror)

explanantion

name	type	supplied/returned	description
MODE	char*(*)	S	Mode is either 'read' or 'write'.
ISWT	integer	S	Switch number.
OBQ	double	S/R	Two pairs of reference coordinates, longitude then latitude, southmost pair followed by the northmost pair. Array length is 4. Logical length will be 4.
NOBQ	integer	R	Returned number of parameters for this projection.
IERROR	integer	R	Returned error parameter, zero is ok.

PROGRAM USAGE: MPSSCA

Set the center scale factor for the transverse mercator projections.

programming example

call mpssca(mode, iswt, sca, nsca, ierror)

explanantion

name	type	supplied/returned	description
MODE	char*(*)	S	Mode is either 'read' or 'write'.
ISWT	integer	S	Switch number.
SCA	double	S/R	Reduction-in-scale factor at the center longitude. Array length is 4. Logical length is 1.
NSCA	integer	R	Returned number of parameters for this projection.
IERROR	integer	R	Returned error parameter, zero is ok.

comments

The center scale factor is normally equal or slightly less than one. The Universal Transverse Mercator projection, for instance, has a center scale factor of .9996.

It needs to be noted the specific method (truncated series) of generating the transverse mercator projection in GCTP is subject to numerical errors and the farther away in longitude from the unitary scale meridian the worse the error. For instance the UTM projection with a scale factor of .9996 is limited about 8 degrees on either side of the reference longitude (.4 meters error). Continuing outward to 16 degrees causes an irrecoverable (at least in this computational system) error of 30 meters on the ground. Other scaling factors cause different amounts of error.

PROGRAM USAGE: MPSHEI

Set the height of the viewpoint for the general near-side vertical projection. This is a perspective projection with a vertical view angle.

programming example

call mpshei(mode, iswt, hei, nhei, ierror)

explanantion

name	type	supplied/returned	description
MODE	char*(*)	S	Mode is either 'read' or 'write'.
ISWT	integer	S	Switch number.
HEI	double	S/R	Height of viewpoint in kilometers. Array length is 4. Logical length is 1.
NHEI	integer	R	Returned number of parameters for this projection.
IERROR	integer	R	Returned error parameter, zero is ok.

Example hardcoded projection setup

The following program sets up and tests a projection used for the Decade of North American Geology (DNAG). Note the programmer knows in advance what parameters require setup.

example code for program test_dnag.f

```
      character         prjnam*80
      double precision  axs(4), ref(4), sca(4), nodata
      double precision  tstlon, tstlat, tstxkm, tstykm

      call mpinit

c Initial setup arrays to no_data.

      nodata = 1.0d38

      do i = 1, 4
        axs(i) = nodata
        ref(i) = nodata
        sca(i) = nodata
      enddo

c Start the setup.

      ipjswt  = 1
      prjnam = 'transverse mercator'
      call mpsnam( 'write', ipjswt, prjnam, nchar, ierror )

      if ( ierror .ne. 0 ) then
        print *, ' Error in projection name'
      endif

c Note N_AXS, N_REF and N_SCA are not used during writes
c to the setup routines.

      axs(1) = 6370.997d0
      axs(2) = 6370.997d0
      call mpsaxs( 'write', ipjswt, axs, naxs, ierr1 )

      ref(1) = -100.0d0
      ref(2) =    0.0d0
      call mpsref( 'write', ipjswt, ref, nref, ierr2 )

      sca(1) = .926d0
      call mpssca( 'write', ipjswt, sca, nsca, ierr3 )

      if ( ierr1 + ierr2 + ierr3  .ne. 0 ) then
        print *, ' Error in setup.'
      endif

c Activate the projection.
```

```
      call mprdy( ipjswt, ierror )

      if ( ierror .ne. 0 ) then
        print *
        print *, '  DNAG projection not active'
        print *
      else
        print *
        print *, '  DNAG projection ready'
        print *
      endif

c  Test calculation.

      tstlon  = -99.0d0
      tstlat  =  39.0d0
      call mpfwd( ipjswt,  tstlon, tstlat,  tstxkm, tstykm,  ierror )

      print *
      print *, ' Test longitude and latitude ='
      print *,  tstlon, tstlat
      print *
      print *,  DNAG projected coordinates ='
      print *,  tstxkm,  tstykm
      print *

      stop
      end
```

Screen dump of test_dnag execution

```
>>> f77 -o test_dnag  -O  test_dnag.f  ~/lib/lib_prjsys.a  ~/lib/lib_oddf.a
>>> test_dnag

 INITIALIZATION PARAMETERS (TRANSVERSE MERCATOR PROJECTION)
 SEMI-MAJOR AXIS OF ELLIPSOID =    6370997.0000 METERS
 ECCENTRICITY SQUARED         =  .0000000000000
 SCALE FACTOR AT C. MERIDIAN  =  .926000
 LONGITUDE OF C. MERIDIAN     = -100 00 00.0000
 LATITUDE OF ORIGIN           =  000 00 00.0000
 FALSE EASTING                =          .0000 METERS
 FALSE NORTHING               =          .0000 METERS
  mprdy ok: test coordinates in and out:
-100.0 .0
.0 .0

   DNAG projection ready

  Test longitude and latitude =
-99.0 39.0

  DNAG projected coordinates =
80.02080808499059 4016.13117733917
```

Example interactive projection setup routine

IACTPJ interactively sets up a projection and is available in generic form in LIB_PRJSYS.A

```
c**************************************************************************
      subroutine iactpj( iswt, kboard, ierror )

c  InterACTive_ProJection.
c
c  This is a general purpose projection setup routine and may be
c  customized for special uses.
c
c  M Webring, 11/91.


      parameter       ( lenp=4 )

      double precision  axs(lenp), ref(lenp), fen(lenp), stp(lenp),
     1                  tsl(lenp), obq(lenp), sca(lenp), hei(lenp)

      character  prompt*80
      character  pjname*80, pjunit*80

      kboard  = 0
      ierror = 0

c  check switch number.

      call mpcksn( iswt, ierror )

      if ( ierror .ne. 0 ) then
        print *, '%%iactpj: returning w/o projection'
        ierror = 1
        go to 999
      endif

c  initialize.

      ihelp  = 0
      ilist  = 0

      pjname = ' '
      len1   = lenp

c  name.

10    print *
      prompt = 'Enter projection name'

      call askc( prompt, pjname, kboard )
      if ( kboard .eq. -2 ) go to 999
```

```
      call mpxnam( 2, pjname, ndxnam, ierror )

      if ( ierror .ne. 0 ) then
        call aski4l( 'Help with names?', ihelp, kboard )
        if ( kboard .eq. -2 ) go to 10
        if ( ihelp .eq. 1 ) call mphnam
        go to 10
      endif

c Return the formal projection name.

      call mpxnam( 1, pjname, ndxnam, ierror )

c writing a projection name causes the prjsys to install the default
c parameters for the projection.

      call mpsnam( 'write', iswt, pjname, nchnam, ierror )

c read in default setup.

      call mpsap1( 'read',  iswt, len1,  pjname, pjunit,
     1 axs, naxs,  ref,  nref,  fen, nfen,  stp, nstp,
     1 tsl, ntsl,  obq, nobq,  sca, nsca,  hei, nhei,  ierror )

      call gclast( pjname, nchnam )
      call gclast( pjunit, nchunt )

      print *
      print *, ' current projection defaults:'
      print *, ' name = ', pjname(1:nchnam)
      print *, ' unit = ', pjunit(1:nchunt)

      print *, ' parameters required:'
      print *, ' axs  ref  fen  stp  tsl  obq  sca  hei'
      print 21, naxs, nref, nfen, nstp, ntsl, nobq, nsca, nhei
21    format( 8i5 )

c axis.

30    if ( naxs .gt. 0 ) then
        if ( naxs .eq. 1 ) then
          prompt = 'Enter spherical radius in km'
        else
          prompt = 'Enter semi-major and minor axis in km'
        endif
        call askf8a( prompt, axs, naxs, kboard )
        if ( kboard .eq. -2 ) go to 10
      endif

c reference.

      if ( nref .gt. 0 ) then
```

```
          if ( nref .eq. 1 ) then
            prompt = 'Enter reference longitude in deg'
          else
            prompt = 'Enter reference lon. and lat. in deg'
          endif

          call askf8a( prompt, ref, nref, kboard )
          if ( kboard .eq. -2 ) go to 30

c Write then read to get special overrides (like default ref-lat for polar
c stereographic is +90 or -90 depending on true scale latitude).

          call mpsref( 'write', iswt, ref, nref, ierror )
          call mpsref(  'read', iswt, ref, nref, ierror )

        endif

c false easting/northing

        if ( nfen .gt. 0 ) then
          prompt = 'Enter false easting and northing in km'
          call askf8a( prompt, fen, nfen, kboard )
          if ( kboard .eq. -2 ) go to 30
        endif

c standard parallels

        if ( nstp .gt. 0 ) then
          prompt = 'Enter standard parallel in deg.'
          call askf8a( prompt, stp, nstp, kboard )
          if ( kboard .eq. -2 ) go to 30
        endif

c true scale latitude.

        if ( ntsl .gt. 0 ) then
          prompt = 'Enter true scale latitude'
          call askf8a( prompt, tsl, ntsl, kboard )
          if ( kboard .eq. -2 ) go to 30
        endif

c oblique mercator.

        if ( nobq .gt. 0 ) then
          prompt = 'Enter oblique mercator lon,lat reference pairs'
          call askf8a( prompt, obq, nobq, kboard )
          if ( kboard .eq. -2 ) go to 30
        endif

c scaling factor.

        if ( nsca .gt. 0 ) then
```

```
        prompt = 'Enter transverse mercator scaling factor'
        call askf8a( prompt, sca, nsca, kboard )
        if ( kboard .eq. -2 ) go to 30
      endif

c perspective height.

      if ( nhei .gt. 0 ) then
        prompt = 'Enter perspective height in km'
        call askf8a( prompt, hei, nhei, kboard )
        if ( kboard .eq. -2 ) go to 30
      endif

c Write all parameters into projection object.

      call mpsap1( 'write', iswt, len1, pjname, pjunit,
     1 axs, naxs,  ref, nref,  fen, nfen,  stp, nstp,
     1 tsl, ntsl,  obq, nobq,  sca, nsca,  hei, nhei,  ierror )

      if ( ierror .ne. 0 ) then
        print *, '%%iactpj: mpsap1 returned error'
        go to 10
      endif

c Read back into local variables.  Parameter corrections made by the
c MPxxxx routines are now available for further updates.

      call mpsap1( 'read', iswt, len1, pjname, pjunit,
     1 axs, naxs,  ref, nref,  fen, nfen,  stp, nstp,
     1 tsl, ntsl,  obq, nobq,  sca, nsca,  hei, nhei,  ierror )

c Summary.

      print *, 'Do you want a summary listing before the'
      prompt = 'projection is initialized ?'
      call aski4l( prompt, ilist, kboard )
      if ( kboard .eq. -2 ) go to 10

      if ( ilist .eq. 1 ) then

        call mpprt( iswt, ierror )

        prompt = 'Is this ok ?'
        call aski4l( prompt, iok, kboard )
        if ( kboard .eq. -2 ) go to 10
        if ( iok   .eq.  0 ) go to 30

      endif

c put projection into ready state.

      call mprdy( iswt, ierror )
```

```
c  error recovery.

        if ( ierror .ne. 0 ) then

          print *
          print *, '%%iactpj: incorrect projection specification'

          prompt = 'Do you want a summary listing ?'
          call aski4l( prompt, ilist, kboard )
          if ( ilist .eq. 1 ) call mpprt( iswt, kerr )

          print *
          prompt = 'Do you want to try again ?'
          call aski4l( prompt, ians, kboard )

          if ( ians .eq. 0  .or.  kboard .eq. -2 ) then
            print *, ' destroying incorrect projection object'
            call mpinsn( iswt, ierr )
            go to 999
          else
           go to 10
          endif

        endif

999     return
        end
```

Subroutine Package ASK

Command Line Question/Answer Interface

Purpose

1 Supply consistent command line interface for user interaction.

2 Isolate application programs from minor variations in compilers.

3 Provide two extra modes of response in addition to answering a question, ie. no_operation (the **ok** response, take the default, keep going) and no_existence (the **mu** response, inappropriate question, back up).

4 Provide transparent journaling of prompts and responses.

General characteristics

The routines described below generally print an application program supplied prompting message and the current default answer or parameter value enclosed in square brackets. The user then has three courses of action:

1) Enter a new value and then press the carriage-return key to update the value.

2) Press only the carriage-return key to accept the current value: the **ok** response.

3) Signal the application that the question is inappropriate and that a new question or function is desired: the **mu** response.

The update occurs only after the user types an answer on the keyboard and then presses the carriage-return key. This key is denoted below as <carriage-return>.

The **mu** response is signalled by entering "MU" or "mu" and then <carriage-return>. These two characters, both caps or both lowercase, in isolation on the line; or a **<ctl><d>** (End_Of_Text) will trigger the mu response. EOT is commonly set to <ctl><d> in UNIX environments and to <ctl><z> in VAX-VMS environments. The ctl-d form is convenient from the user's point of view but since the response is somewhat dependent on the specific operating environment, may not work without external setup.

Mu may be considered to be a short word on a par with yes, no and ok meaning 'I cannot/will_not answer this question and a default answer is not allowed'. The program response to mu depends on the situation but typically the interactive question/answer sequence will back up to some previous point so the user can choose a new course of action. Mu does not carry the strong semantic connotations that "quit", "cancel" or "exit" have and rather than exiting a dialog box as in recent GUI (Graphical User Interface) usage, mu allows the equivalent of moving around inside a given command line dialog.

The three user actions listed are returned to the application program as the KBOARD parameter in the descriptions that follow.

Software dependencies

The general character (GenChar) subsystem is called by the ASK package.

SUBROUTINE USAGE GUIDE

Prior to calling these routines the updated variable (the second item in the calling sequence) should be set to some default value. While this is not strictly necessary, the purpose of the default is to aid the user in selecting a response.

A user keyboard response consisting only of the character strings "mu" or "MU" followed by a <carriage-return> or the EOT form of the mu response may be entered. The mu response is returned to the calling program via KBOARD=-2 (see the description for ASKYN for the full KBOARD definition). The calling program then determines the appropriate response.

Subroutine list

askin initialize the ASK subsystem.

askc update a character string.

askf4 update a 4 byte real (single precision) value.
askf8 update a 8 byte real (double precision) value.
aski4 update a 4 byte (longword) integer value.

askf4a update a 4 byte real array.
askf8a update a 8 byte real array.
aski4a update an 4 byte integer array.

aski4l update a 4 byte integer interpreted as logical true or false (1 or 0).
askyn read a 'yes' or 'no' from users terminal.

askpr print a character string to users terminal.
askrd read a character string from users terminal.

askdsp de/activate printing to the display.
askjnl de/activate a file unit for journaling.

PROGRAM USAGE: ASKIN

The ASKIN routine initializes the data object common to the ASK package. The default input device is the user's keyboard. The default output is the screen and journaling is turned off. A call to this routine is unnecessary when the ODDF system as a whole is being used. If the ASK package is being used in a standalone fashion, then one of the first executable statements should be: *call askin*; there are no parameters.

PROGRAM USAGE: ASKC

The ASKC routine updates a character string.

operating characteristics

1) The ASKC routine will not recycle since there is no character string which is intrinsically invalid. The calling program must define character string validity.

2) As with the rest of the ASK system, carriage-return is the signal to take-the-default. To set the update string to blanks, enter a quoted blank string. Either " " or ' ' will work.

3) Note the above definition of the MU response: the string must be the only characters of a user's entry to operate. The characters mu and MU may appear as usual in combination with other characters. The <ctl><d> form normally works at any time but is dependent on the operating environment.

programming example

call askc(prompt, chrstr, kboard)
if (kboard .eq. -2) go to nnn

Where *nnn* typically points to a statement label earlier in the interactive sequence.

explanation

name	type	supplied/returned	description
PROMPT	char*(*)	S	Prompt is a supplied character string printed to the user's terminal.
CHRSTR	char*(*)	S/R	Character string to be interactively updated.
KBOARD	integer	R	Returned user response code. KBOARD = 0 something entered via the keyboard, -1 current value accepted, -2 question rejected.

PROGRAM USAGE: ASKF4, ASKF8, ASKI4

These routines update a floating point number, a double precision floating point number or a longword integer. The numeric designation indicates the number of bytes the updated variable contains. The ASKF8, and ASKI4 routines are the same as the example below, just change the update variable type (the second item in the parameter list). None of the routines change the value of the update variable until a valid response is received.

programming example

```
real*4 fnum
...
call askf4( prompt, fnum, kboard )
if ( kboard .eq. -2 ) go to nnn
```

Where NNN typically points to a statement label earlier in the interactive sequence.

explanation

name	type	supplied/returned	description
PROMPT	char*(*)	S	A character string to be printed to the users screen.
FNUM	real*4	S/R	A real (floating point) variable to be interactively updated. ASKF4 is real type, ASKF8 is double precision type, ASKI4 is integer type.
KBOARD	integer	R	Returned user response code. KBOARD = 0 something entered via the keyboard, -1 current value accepted, -2 question rejected.

comments

The routine must be able to parse the keystrokes entered by the user into a valid number based on the variable type. In case the user's response cannot be parsed, the ASKI4, ASKF4, and ASKF8 routines allow two more attempts and then not receiving a valid character string will set the variable to a large positive number (the appropriate no-data value). Of course a carriage-return or MU response will leave the variable with the same value and the program will continue.

PROGRAM USAGE: ASKF4A, ASKF8A, ASKI4A

These routines update a single precision real, double precision real or longword integer array. The example below describes a single precision array, for double precision or integer, change the variable type and the subroutine name as appropriate.

operating characteristics

1) The ASKxxA routines will not recycle in case an update element is invalid. The routines merely skip that element.

2) Spaces and/or commas may delimit elements.

3) Once a carriage-return is issued, the routines process the user's entry. The user may update as few elements as desired.

Given rules 1, 2, and 3 the user has a variety of syntax that may be used. If the current (default) array elements are: 1 2 3 4. The user may enter any of the following variations of obtain 1 5 3 7 (underlines indicate user input):

1 5 3 7 Variable number of spaces to delimit elements,

1,5, 3, 7 Combination of commas and spaces,

,5,,7 Commas to delimit only the changed elements,

* 5, #, 7 Invalid entry to cause an element skip. Because different machines interpret garbage strings differently, this method is discouraged.

programming example

call askf4a(prompt, farray, nele, kboard)
if (kboard .eq. -2) go to nnn

explanation

name	type	supplied/returned	description
PROMPT	char*(*)	S	Same as above.
FARRAY	real*4	S/R	Same as above, except FARRAY is an real type array. ASKF4A is real type, ASKF8A is double precision type, ASKI4A is integer.
NELE	integer	S	Length of farray. **Must be greater than zero.**
KBOARD	integer	R	Same as above.

PROGRAM USAGE: ASKI4L

This routine treats the updated variable as a logical switch where zero is false and one is true.

operating characteristics

1) The second variable in the parameter list (IANS) may equal only 0 or 1.

2) The IANS parameter is displayed on the users terminal as [n] or [y] for false and true respectively.

3) The routine **will reset IANS to zero** (false) if it is neither 0 nor 1 when entering the routine.

programming example

call aski4l(prompt, ians, kboard)
if (kboard .eq. -2) go to nnn

explanation

name	type	supplied/returned	description
PROMPT	char*(*)	S	The prompt that is printed on the users terminal.
IANS	integer	S/R	The answer is a logical state in response to keyboard input. IANS = 1 true or yes. = 0 false or no.
KBOARD	integer	R	User response code. Since ASKI4L is returning a logical answer, KBOARD is the full response of ASKYN described below.

PROGRAM USAGE: ASKYN

Return only the keyboard response code specified by the user's input. There is no prompting by the routine. The functionality of this routine has been supplanted by ASKI4L, but is described here for possible speciality uses.

programming example

```
call askyn( kboard )
if ( kboard .eq. -2 ) go to kkk
if ( kboard .eq. -1 ) go to lll
if ( kboard .eq.  0 ) go to mmm
if ( kboard .eq.  1 ) go to nnn
```

explanation

name	type	supplied/returned	description
KBOARD	integer	R	Returned state in response to keyboard input.

interpretation of keyboard input

Your_response	KBOARD_equals	logical_interpretation	action
y, Y, yes, YES	1	true	yes branch
n, N, no, NO	0	false	no branch
<carriage-return>	-1	do not change	current yes or no branch
mu, MU, <control><d>	-2	does not exist	back up the decision tree

Future upgrades to the system may include a ">>" metacharater meaning 'fast_forward' which would continue the program execution but turn off interaction until a resume_interaction marker is reached. The value of KBOARD in response to ">>" would be 2 and be the conceptual inverse to "mu", the backup metacharacter.

Routines ASKPR and ASKRD

These routines are a low level of the ASK package and are called by the question/answer (eg. ASKC or ASKF4) routines. If journaling is turned on, then any display/keyboard I/O moderated by these routines is also directed to the journal.

PROGRAM USAGE: ASKPR

Print a message to the user's screen and/or write the message to the journal file. The message may be as long as desired and the routine will format the printed message by issuing a carriage_return/line_feed pair (ASCII characters 13 and 10) at word breaks such that the resulting lines are less than 78 characters. In addition, line formatting characters (either carriage_return/line_feed or line_feed) may be embedded in the character string by the programmer to cause line breaks as needed. This routine is useful for online help messages.

programming example

call askpr(prompt)

explanation

name	type	supplied/returned	description
PROMPT	char*(*)	S	Character string to be printed to users screen. If journaling has been activated, the prompt is also directed to the journal file.

PROGRAM USAGE: ASKRD

Read a line of text from the user's keyboard and return it in the character string TXTREC. If journaling is turned on, the response is written to the journal, and if the response is EOT, then a "mu" is written to the journal (rather than the non-printing EOT control character). If the user's input is "mu", "MU", or a <control><d> (EOT), then TXTREC is set to blank and KBOARD to -2.

programming example

call askrd(txtrec, kboard)

explanation

name	type	supplied/returned	description
TXTREC	char*(*)	R	Character string read from the users terminal. Control characters are replaced with blanks. If journaling has been activated, the prompt is also directed to the journal file.
KBOARD	integer	R	Response code either 0, -1, or -2.

Routines ASKDSP and ASKJNL

These two routines allow the programmer to alter the printing to the user's display or to journal files. The routines are not interactive.

PROGRAM USAGE: ASKDSP

Activate or deactivate printing to the user's screen or return the current state of printing to the user's screen. Only the printing mediated by the ASKPR routine is affected. The display acts independently of the journal.

programming example

call askdsp(mode, idsp, ierror)

explanation

name	type	supplied/returned	description
MODE	char(*)	S	Mode is either 'read' or 'write'.
IDSP	integer	S/R	IDSP is either 0 to suppress printing to the screen or 1 to activate printing to the screen. If MODE is 'read', then IDSP returns the current state. If MODE is 'write', then IDSP sets the current state.
IERROR	integer	R	Error return is zero if MODE and IDSP specified correctly.

PROGRAM USAGE: ASKJNL

Activate or deactivate writing to a journal file or return the current state of writing to the journal. Any string supplied to the ASKPR routine, prompt supplied to the interactive routines (eg. ASKC or ASKF4), or user's keyboard response is directed to the the journal file while journaling is active. Multiple journal files may be used (one at a time) and the file may be any text (stream formatted) file opened for other purposes.

programming example

call askjnl(mode, iunit, jour, ierror)

explanation

name	type	supplied/returned	description
MODE	char(*)	S	Mode is either 'read' or 'write'.
IUNIT	integer	S	File unit number in range 7 to 99 (5 is the Fortran unit number for the user's keyboard and 6 is the display, 1 to 4 are traditionally for system files). The file unit is opened/closed externally to the ASK package by the calling program.
JOUR	integer	S/R	JOUR is either 0 to suppress writing to the journal or 1 to activate writing to the journal. If OPR is 'read', then JOUR returns the current state. If OPR is 'write', then JOUR sets the current state.
IERROR	integer	R	Error return is zero if MODE and JOUR specified correctly.

ASK programming example

Below is a subroutine that sets up several variables for use in an application program. Of note in this example is the *"if (kboard .eq. -2)"* test for the **mu** response. The values asked for in this routine may be revisited many times and if "mu" is entered in the first group of statements then the routine exits with KBOARD = -2 that the calling may test for and act upon. Tests for KBOARD equal 0 and -1 (something_entered and take_default) are more rare and depend on the pattern of interaction the programmer needs, but users quickly become accustomed to backing up and redoing something so tests for -2 accompany each question. The interactive routine following the user's guide for map projections is an example of a more complex real-world application of the ASK routines.

```
c*************************************************************************
      subroutine setpra( nltrue, isgrec, ntxtlr, kboard )

c  SETup_Physical_Record_Access

      print *
      print *, ' INPUT_FILE PHYSICAL_RECORD ACCESS'


10    prompt = 'Is this a sequential access file ?'
      call aski4l( prompt, nltrue, kboard )
      if ( kboard .eq. -2 ) go to 999

20    prompt = 'Is a single text line a complete posting record ?'
      call aski4l( prompt, isgrec, kboard )
      if ( kboard .eq. -2 ) go to 10

      prompt = 'Enter number of text_lines per posting_record'
      call aski4( prompt, ntxtlr, kboard )
      if ( kboard .eq. -2 ) go to 20


999   return
      end
```

Subroutine Package FileVersion

File Utility Interface

Purpose

1) Provide a single point for resolving hardware and operating system dependencies. The most fundamental of which is whether the hardware stores numbers in most_significant or least_significant byte order and the encoding style of real numbers.

2) Provide a common set of file opening calls that is independent of the particular Fortran implementation. For instance, when opening a direct access file on Data General and Digital Equipment machines the record length is specified in 4 byte words and on Hewlett Packard and Sun machines the record length is specified in bytes.

3) Simplify the usage of new, old and unknown file status types.

4) Give UNIX a file version capability which aids users doing sequential processing steps. The versions are denoted by a colon, "test.dat:03" for instance.

5) Aid the programmer with filename utilities for common manipulations, for instance, changing a component from ".dat" to ".grid" or checking filenames for non-visible characters.

FileVersion capabilities

1) This package performs the filename attachment to file unit number in a manner similar to the Fortran open statement.

2) On UNIX systems the file appears in a given directory with a filename that has an appended ":01" through a maximum version of ":20".

2) File unit numbers in the range 1 to 99 are supported. Units 1 to 4 are normally reserved for ODDF system use and 5 and 6 are the Fortran terminal I/O units.

3) Three file types are recognized: FMT is formatted stream (ie. ASCII text), BIN is unformatted binary sequential, and DA is unformatted direct access.

4) The package will close files with either 'keep' or 'delete' options.

5) The package recognizes status types: 'new', 'old', and 'unknown'.

6) Filenames may be specified by the user without version numbers. The system will automatically take file version action for types 'new' and 'old'.

7) The package corrects filenames for common errors.

FileVersion limitations

1) You cannot have any directories or files in your path that have a colon in their names. This prevents versions of versions.

2) The highest version allowed is 20. Typically an external routine is available to remove lower versions of a file and change the highest remaining version to ":01".

3) File versions are automatically turned off on VMS machines since the operating system handles file versions and on MS/DOS machines due to filename length and character restrictions.

4) The direct access record length is specified in longwords and therefore is limited to multiples of 4 bytes. This ensures a given program will transport more easily among different types of hardware.

File version bypass

The presence of a file named "fvfalse.cf" (ie. file_version_false.configuration_file) in your UNIX $HOME directory globally turns off file versions. The contents of this file are not used.

Opening status

Fortran allows three types of status or existence when opening a file: new, old and unknown. Briefly, a **new** file does not exist and is about to be created and an **old** file already exists. The **unknown** status type is a mixture of the two other types.

The various opening status implementations and error handling of files provided a major reason for developing the FileVersion package. Error handling for old and new files causes extra coding in application programs and can interrupt program flow from the user's point of view. The use of 'unknown' leaves a program in an uncertain state in case of minor errors (eg. you want to read an 'unknown' file that doesn't exist, so the operating system may create an empty file) and leaves open the possibility for the user to inadvertently overwrite a file.

Without detailing the various permutations of reading and writing to existent and nonexistent files with each of the 3 status types and which side effects are undesirable, suffice it to say there will probably be one that will trip the unwary programmer. FileVersion presents the programmer and user with a few clearly defined options that serve for day-to-day use.

From the PROGRAMMER'S point of view

NEW files are always opened with a version number higher that others in the given directory, version number ":01" being the lowest. The 'new' file open will fail only if the user does not have write permission in a given directory or 20 versions are exceeded. If a programmer consistently uses the 'new' option (especially for temporary files generated by a program), then multiple copies of a program can be run in the same directory without interfering with each other.

OLD files must exist or an error is generated.

UNKNOWN files are not processed via the version number portions of the system. The user's are on their own, except the package will warn them if higher versions are present. Use of the 'unknown' file type is not

recommended when the program is interacting with users, but is ok for hardwired filenames typically used as intermediate files or when a file should be created without a version number.

and from the USER'S point of view.

NEW files with version numbers means the same filename may be reused for related operations and the data or files generated by earlier operations remains available. The user cannot get an error message that the file already exists and an existing file cannot be overwritten. Higher versions will have later creation times since output files are created with a system generated version number. However, the user does have to make sure the maximum of 20 versions are not exceeded (typically an external routine is available that removes lower versions and changes the highest version to ":01").

OLD files can be opened with a specific version or without a version. If you specify "test.dat" and a program opens the highest existing version "test.dat:04" (which is not **exactly** what you asked for), the FileVersion package then prints a message telling exactly what file and version was opened.

UNKNOWN files from a user's point of view mean an existing file may get overwritten due to a simple oversight or typographic error.

Warning and advisory messages

The package does not interact with the user; it only advises when something may not be what the user is expecting.

The package will advise the user when a file is opened whose name is not exactly as requested (eg. a filename without version number was supplied and the system opened the last version available in the case of an old file). In this manner the user always knows which version of a file is being used by an application.

When opening with status 'unknown', the package will warn when higher versions exist, but attempt to open the filename as given.

When a file has been opened correctly, the error code equals zero. When not opened correctly, the error code is not-zero, and the calling routine must determine the problem.

Software dependencies

FileVersion calls the GenChar subroutine package.

SUBROUTINE USAGE GUIDE

These routines provide a low level access for the attachment of files to a program. The open routines OPNFMT, OPNBIN and OPNDA mimic the Fortran statements for the three basic types of **unstructured data**. The routines for opening and closing the **grid and point structured data** files are detailed in the chapters covering GridIO and PostIO respectively.

The application programmer will typically use one of the filename construct routines, open a file with the constructed name, and close the file when I/O operations are complete. Other subroutine descriptions are provided so the application programmer can setup more complex access.

There are two major groupings of subroutine names: FVnnnn are general calls and OPNnnn are file opening calls. The FVnnnn routines have several subdivisions of which the FVFNnn are filename construct routines.

Subroutine List

fvinit	FileVersion initialization, the routine where machine dependencies are resolved.
fvoddf	Returns a character string describing the version of ODDF being used.
opnbin	Binary file open.
opnda	Direct access file open.
opnfmt	Formatted file open.
fvclos	File closing routine.
fvfnal	Filename construct. Add a suffix to the first component.
fvfncd	Filename construct. Remove directory components.
fvfnmk	Filename construct. Change last component.
fvfnok	Filename validity check.

PROGRAM UASGE: FVINIT

Initialize the FileVersion system. When FileVersion is used in isolation of the rest of ODDF, this routine must be called. A call to this routine is unnecessary when a program has initialized the entire ODDF system via the PFINIT routine.

FVINIT is the location for specifying all machine and operating system dependencies for the ODDF system and contains a variable to describe the operating system and a multiplier to convert longwords to the Fortran direct access record length. The overall ODDF system version is also assigned to a character string in this routine.

Future upgrades will probably include descriptions of the floating point number format (ie. IEEE or VMS) and a byte ordering description (ie. Most_Significant_Byte or LSB). This will enable the ODDF system to convert binary numbers created on one type of hardware as required by the current host machine.

programming example

call fvinit

There are no parameters.

PROGRAM USAGE: FVODDF

This subroutine returns a character string that describes the version of ODDF the current application is using. The character is similar to: ODDF = "version = 1.5.5, dictionary = 1.2.1". A similar character string is the first line in grid and point data file headers.

programming example

*character sysver*80*
...
call fvoddf(sysver)

explanation

name	type	supplied/returned	description
SYSVER	char*80	R	Returned character string that describes the version of ODDF that is linked to the driver that contains the FVODDF call.

FILE OPENING ROUTINES: OPNFMT, OPNBIN, OPNDA

These routines are designed to look like regular Fortran opens. FMT and BIN are sequential, BIN is unformatted, and DA is direct access unformatted. Fortran status='scratch' is not implemented, use FVCLOS with the delete argument instead. Each of these opening routines calls FVFNOK (see below) to ensure a reasonably valid filename is presented to the Fortran runtime system.

programming example

call opnfmt(iunit, filen, stat, ierror)
call opnbin(iunit, filen, stat, ierror)
call opnda(iunit, filen, stat, lenrec, ierror)

explanation

name	type	supplied/returned	description
IUNIT	integer	S	Fortran unit number in the range 1 to 99 and is the number used for subsequent references to this file. Units 1 to 4 are reserved for system use and 5,6 are interactive input and output.
FILEN	char*(*)	S	Filename. Must be less than 80 characters.
STAT	char*(*)	S	Status new, old, unknown. The first character of STAT is either "n", "o", or "u" in either lower or uppercase. Subsequent characters are not checked.
LENREC	integer	S	OPNDA only. Direct access record length in longwords on all machines (4 bytes per longword).
IERROR	integer	R	Returned error parameter is zero when file opened ok.

FILE CLOSING ROUTINE: FVCLOS

This routine will close a file that was previously opened with the FileVersion package.

programming example

call fvclos(iunit, dispose)

explanation

name	type	supplied/returned	description
IUNIT	integer	S	Unit number in the range 1 to 99.
DISPOSE	char*(*)	S	Status after closing, DISPOSE may be lower or uppercase. DISPOSE = 'keep' or ' ' (blank) does not delete the file, 'delete' deletes the file.

PROGRAM USAGE: FVFNCD

Create a filename without directory information.

programming example

call fvfncd(filen1, filen2)

explanation

name	type	supplied/returned	description
FILEN1	char*(*)	S	File name with either UNIX, VMS, or DOS construction.
FILEN2	char*(*)	R	Returned filename without directory info.

PROGRAM USAGE: FVFNMK

Make a new filename by stripping directory info and changing the suffix (the final component of the name, eg. "test.grd" to "test.dat").

programming example

call fvfnmk(filen1, suffix, filen2)

explanation

name	type	supplied/returned	description
FILEN1	char*(*)	S	Filename with either UNIX, VMS, or DOS construction.
SUFFIX	char*(*)	S	New suffix.
FILEN2	char*(*)	R	Returned filename.

PROGRAM USAGE: FVFNA1

Make a new filename by stripping directory info and appending a string to the first component of the name, eg. "test.grd" to "test_2.grd".

programming example

call fvfna1(filen, chrstr, file2)

explanation

name	type	supplied/returned	explanation
FILEN1	char*(*)	S	File name with either UNIX, VMS, or DOS construction.
CHRSTR	char*(*)	S	Character added to the first component.
FILEN2	char*(*)	R	Returned filename.

PROGRAM USAGE: FVFNOK

Check a filename for validity and attempt repairs. Routines OPNBIN, OPNDA, and OPNFMT call FVFNOK.

operating characteristics

The conditions necessary for a returned error code of zero (ok) are:

1) Both supplied and returned filename strings must be non-zero in length.
2) There must be visible characters in the returned string (trailing blanks are always ignored).
If either of these conditions fail, then an advisory message is printed to the user's screen.

Repair possibilities:

Repairs are silent, there are no advisory messages.
1) The filename is left justified to remove preceding blanks.
2) All control characters and embedded blanks are substituted by an underscore ("_"). Control characters are ASCII code indices 0 through 31 (base 10) and include such functions as horizontal tab, alert (bell), and backspace.
3) All characters with indices greater than 126 (base 10) are replaced with the underscore, ie. the "~" (tilda) character is the last character in the ASCII character sequence.

Conditions the routine does not check:

1) The case of any character.
2) The presence of special characters like punctuation, for instance, a ";" in UNIX means end-of-shell-command. Operating system meta-characters should be avoided by the user to simplify recovery of the filename.

programming example

call fvfnok(filen1, filen2, ierror)

explanation

name	type	supplied/returned	description
FILEN1	char*(*)	S	Supplied filename.
FILEN2	char*(*)	R	Returned filename. The FILEN2 character string must be at least as long as the number of visible characters in FILEN1.
IERROR	integer	R	Returned error code. Zero is ok, non-zero is not-ok. The calling program defines 'ok'.

Subroutine package GenChar

General operations on character strings

Purpose

1 Provide a consistent definition of character string operations for the ODDF system.

2 Provide external usage to simplify application programs.

General Characteristics

All of the routines in this package may be called by user applications. The routines described below are independent of other subroutine packages and most call no other General Character (GenChar) subroutines. Some print warning messages to the user's screen but all have default operation in case of an error. None of these routines use a common area and the package as a whole does not require initialization.

Disclaimer

The desired behavior of each routine is stated in the descriptions or comments. Any behavior not stated must be considered as undocumented and subject to change without notice. Undocumented behavior is normally caused by compilers on various machines interpreting the source code according to their own undocumented assumptions and/or the occasional oversight by the author. Changes in the documented behavior will hopefully be rare and will be detailed in update notices.

Classification of Characters

The GenChar routines were written to support the PDS Object Description Language parse and construction of attributes in the form: keyword = value. Some of the definitions, notably alpha-numeral, are concerned with keyword detection and verification, so the '+' symbol which might be considered to be an alpha-numeric character is not a numeral for keyword purposes.

The ASCII encoding sequence may be summarized as:

character number	**ASCII characters**	**GenChar class**
0-31	eg. null, linefeed	control characters
32	space	generic word delimiter and string pad
33-47	!"#$%&'()*+,-./	not classified
48-57	0123456789	numerals
58-64	:;<=>?@	not classified
65-90	uppercase letters	alphabetic characters
91-94	[\]^	not classified
95	_ (underscore)	alphabetic character
96	`	not classified
97-122	lowercase letters	alphabetic characters
122-126	{|}~	not classified
127-255	unassigned	not classified

Print control characters are considered to be the following:

8	bell	not used
9	backspace	not used
10	linefeed	used in PDS labels
11	vertical tab	not used
12	form feed	not used
13	carriage return	used in PDS labels

Printing characters are considered to be ASCII 8 to 13 and 32 to 126. see subroutine GCRPNP.

Visible characters are ASCII 32 to 126, where the space is a visible character. Note that visible characters are a subset of printing characters. See subroutine GCRPNV.

Subroutine list and brief description

gcaptx	Append new text onto old text.
gccvc	Convert case.
gclast	Return the position of the last nonblank character.
gcleft	Left justify a character string.
gcnthw	Return the nth word in a string.
gcnxtw	Return the next word of a string.
gcpalf	Return the position of the first alphabetic character.
gcpaln	Return the position of the first alpha-numeral character.
gcpct	Return the position of the first control character.
gcpnal	Return the position of the first non-alphanumeral character.
gcrf4	Read first word into a binary 4 byte real.
gcrf8	Read first word into a binary 8 byte real (double precision).
gcri4	Read first word into a binary 4 byte integer.
gcrite	Right justify a string.
gcrpcc	Replace C comments with blanks.
gcrpcm	Replace commas with blanks.
gcrpct	Replace control characters with blanks.
gcrplt	Replace literals with blanks.
gcrpnp	Replace nonprinting characters with blanks.
gcrpnv	Replace nonvisible characters with blanks.
gcwf4	Write 4 byte real value into character string.
gcwf8	Write 8 byte real value into character string.
gcwi4	Write 4 byte integer value into character string.

Conventions used in the descriptions

Uppercase words refer to subroutine names or variables. Italics indicate example Fortran code. Word is taken to mean a character string delimited with spaces or the first/last position in the string. The spaces padding a character string to the length of a variable are ignored. Blank is sometimes used to mean the space character.

'supplied' means the value has been supplied by the calling routine.

'returned' means the value is returned to the calling routine.

char*(*) indicates a character variable with any length greater than zero and less than some very large maximum.

integer indicates a 4 byte signed integer. Other forms are not used.

real*4 indicates a 4 byte floating point variable.

real*8 indicates an 8 byte double precision floating point variable.

Substring Position

The GenChar routines return position variables, say the last character of a string, relative to the string that was passed to the routine. Therefore if you pass a substring via *call gcaaaa(txtrec(istart:nn), ... ipos, ...)*, then IPOS is relative to the start character location ISTART (ie. GCaaaa does not know it is dealing with a substring). The absolute position is then: IABS = (IPOS - 1) + ISTART.

PROGRAM USAGE: GCAPTX

Append a second text string to the first string. The last non-blank character of TXTREC is found and then NEW is appended after that character. NEW is not modified (eg. left justify) before the append operation. If a word break between TXTREC and NEW is desired, a space may precede the characters in NEW.

programming example

call gcaptx(txtrec, new, nchtxt)

explanation

name	type	supplied/returned	description
TXTREC	char*(*)	S&R	The supplied string that will have NEW appended to it.
NEW	char*(*)	S	String to be appended to TXTREC.
NCHTXT	integer	R	Position of the last non-blank character. Note substring position warning.

PROGRAM USAGE: GCCVC

Convert case, lower to upper and vice versa, of all alphabetic characters in the string.

programming example

call gccvc(txtrec, iopr)

explanation

name	type	supplied/returned	description
TXTREC	char*(*)	S&R	String to be converted, only "a" to "z" and "A" to "Z" are affected. The result is returned in this string.
IOPR	integer	S	Set to 1 for uppercase output, 2 for lowercase. Anything else is results in no changes.

PROGRAM USAGE: GCLAST

Return the position of the last non-blank character.

programming example

call gclast(txtrec, ipos)

explanation

name	type	supplied/returned	description
TXTREC	char*(*)	S	Input character variable.
IPOS	integer	R	Position of the last character. Note substring position warning.

comments

The NULL character (ASCII 0) is considered a blank.

PROGRAM USAGE: GCLEFT

Left justify a string. Strip preceding blanks and move the remaining non-blank character string left to position one, pad with blanks from the end of the non-blank character string to the end of the character variable.

programming example

call gcleft(txtrec, lastch)

explanation

name	type	supplied/returned	description
TXTREC	char*(*)	S&R	Character variable that contains the supplied string and the returned left-justified string.
LASTCH	integer	R	position of the last non-blank character. Note subsring position warning.

comments

If there are no nonblank characters, then LASTCH = 0.

PROGRAM USAGE: GCNTHW

Return the Nth word, where a 'word' is a string bounded by spaces or the first or last position of the character variable.

programming example

call gcnthw(txtrec, nthwrd, ipos, wrd, nchwrd)

explanation

name	type	supplied/returned	description
TXTREC	char*(*)	S	Input character variable.
NTHWRD	integer	S	The number or position in TXTREC of the desired word.
IPOS	integer	R	Position of the first character of WRD in TXTREC. Note substring position warning.
WRD	char*(*)	R	Returned word.
NCHWRD	integer	R	Number of characters in WRD.

comments

This routine makes sequential calls to GCNXTW (next word). If there are more than 256 spaces to the next word, the routine fails. If there are no more words, then NCHWRD = IPOS = 0 and WRD = ' '.

PROGRAM USAGE: GCNXTW

Return the next word, where a word is a string bounded by spaces or the first or last position of the character variable.

programming example

call gcnxtw(txtrec, ipos, wrd, nchwrd)

explanation

name	type	supplied/returned	description
TXTREC	char*(*)	S	Input character variable.
IPOS	integer	R	Position of the first character of WRD in TXTREC. Note substring position warning.
WRD	char*(*)	R	Returned word.
NCHWRD	integer	R	Number of characters in WRD.

comments

If there is no word, then NCHWRD = IPOS = 0 and WRD = ' '. If the returned word is longer than the length of the variable WRD, then the content of WRD is set to blanks but NCHWRD and IPOS have the correct values they would have had if WRD were long enough.

PROGRAM USAGE: GCPALF

Return position of the first alphabetic character. Alphabetic characters are : a to z, A to Z and "_".

programming example

call gcpalf(txtrec, ipos)

explanation

name	type	supplied/returned	description
TXTREC	char*(*)	S	Supplied character string.
IPOS	integer	R	Returned position of the first alphanumeric character. Note the substring position warning.

PROGRAM USAGE: GCPALN

Return position of the first alpha-numeral character. Alphanumeral characters are : a to z, A to Z, 0 to 9, and the underscore character "_".

programming example

call gcpaln(txtrec, ipos)

explanation

name	type	supplied/returned	description
TXTREC	char*(*)	S	Supplied character string.
IPOS	integer	R	Returned position of the first alphanumeric character. Note the substring position warning.

PROGRAM USAGE: GCPCT

Return the position of the first control character. Control characters are ASCII 0 to 31 (base 10).

programming example

call gcpct(txtrec, ipos)

Arguments same as for GCPALN.

PROGRAM USAGE: GCPNAL

Return the position of the first non-alphanumeral character. This routine is the inverse of GCPALN.

programming example

call gcpnal(txtrec, ipos)

Arguments same as for GCPALN.

comments

The space character (ASCII 32 base 10) is ignored in GCPNAL.

PROGRAM USAGE: GCRF4, GCRF8, GCRI4

Return a binary real*4, real*8, or integer*4 value interpreted from the first word in a supplied character string. A word is a character string delimited by spaces or the string boundaries. Note the input string TXTREC precedes the output variables F4, F8, I4.

programming example

call gcrf4(txtrec, f4, ierror)
call gcrf8(txtrec, f8, ierror)
call gcri4(txtrec, i4, ierror)

explanation

name	type	supplied/returned	description
TXTREC	char*(*)	S	Supplied character variable.
F4	real*4	R	Interpreted number read from the first word in TXTREC.
F8	real*8	R	Interpreted double precision number read from the first word in TXTREC.
I4	integer	R	Interpreted number read from the first word in TXTREC.
IERROR	integer	R	Error code. Zero means ok. In case of read error, not only is IERROR set nonzero but F4, F8, or I4 is set close to their maximum values.

comments

The 'maximum' values are a convenient transportable value somewhat less than the hardware coded values:
Real*4 3.4e38 (3.4 x 10 ^ 38)
Real*8 3.4d38
Integer*4 2147483647

The strings "NaN", "+INF", "-INF" (ie. the IEEE NotANumber and INFinity) are not recognized and cause the error condition.

These three routines do the operation of a free-field read from a character variable. This functionally equivalent Fortran example sets F4 to 1234.56:

txtrec = '1234.56 is a number'
*read(txtrec, *, iostat=ierr) f4*

PROGRAM USAGE: GCRITE

Right justify a string. Move a string right inside a character variable until all trailing blank characters are removed, pad the beginning of the character variable with blanks up to the position of the first non-blank character.

programming example

call gcrite(txtrec, ifirst)

explanation

name	type	supplied/returned	description
TXTREC	char*(*)	S&R	Character variable where the non-blank characters are moved to the last positions. GCRITE is the inverse of GCLEFT.
IFIRST	integer	R	Position of the first non-blank character in the returned variable. Note substring position warning. If there are no non-blank characters IFIRST = 0.

PROGRAM USAGE: GCRPCM

Replace commas in a string with blanks.

programming example

call gcrpcm(txtrec)

explanation

name	type	supplied/returned	description
TXTREC	char*(*)	S&R	Supplied character string and returned string with commas replaced with blanks.

PROGRAM USAGE: GCRPCT

Replace control characters with blanks. Control characters are ASCII 0 to 31 (base 10).

programming example

call gcrpct(txtrec)

explanation

name	type	supplied/returned	description
TXTREC	char*(*)	S&R	Supplied character string and returned string with control characters replaced with blanks.

PROGRAM USAGE: GCRPNP

Replace nonprinting characters with blanks.

programming example

call gcrpnp(txtrec)

explanation

name	type	supplied/returned	description
TXTREC	char*(*)	S&R	Supplied character string and returned string with nonprinting characters replaced with blanks.

PROGRAM USAGE: GCRPNV

Replace nonvisible characters with blanks.

programming example

call gcrpnp(txtrec)

explanation

name	type	supplied/returned	description
TXTREC	char*(*)	S&R	Supplied character string and returned string with nonvisible characters replaced with blanks.

comments

The print control characters are **not** considered to be visible characters.

PROGRAM USAGE: GCWF4, GCWF8, GCWI4

Encode a real*4, real*8, or integer*4 binary value into a character string. In the real*4 and real*8 case, format for best readability. Note the input (F4, F8, I4) precedes the output (TXTREC).

programming example

call gcwf4(f4, txtrec, ierror)
call gcwf8(f8, txtrec, ierror)
call gcwi4(i4, txtrec, ierror)

explanation

There are three routines described here, the only difference among them is the variable type of the first parameter.

name	type	supplied/returned	description
F4	real*4	S	Supplied single precision floating point number,
F8	real*8	S	Supplied double precision number,
F4	integer	S	Supplied single precision fixed point (integer) number.
TXTREC	char*(*)	R	Returned left-justified character string. Length guide: real*4 numbers need at least 16 characters. real*8 numbers need at least 24 characters. int*4 numbers need at least 12 characters.
IERROR	integer	R	Error code, zero means ok. In case of read error, not only is IERROR set nonzero but TXTREC is set to blank.

comments

The IEEE binary forms for NaN, +INF, -INF are not recognized and cause the error condition above.

These three routines do the operation of a free-field write to a character variable. This Fortran example sets TXTREC to "1234.56":

f4 = 1234.56
*write(txtrec, *, iostat=ierr) f4*

The text representation of the floating point numbers is formatted such that numbers with most-significant digits greater than 100,000 or less than .01 are in exponential form. The appropriate number of significant figures for the data type are maintained.

References

Cordell, Lindrith, Phillips, J.D., and Godson, R.H., 1992, U.S. Geological Survey Potential-Field Geophysical Software, Version 2.0: U.S. Geological Survey Open-File Report 92-18.

Dewhurst, W.T., 1990, NADCON - The Application of Minimum-Curvature-Derived Surfaces in the Transformation of Positional Data from the North American Datum of 1927 to the North American Datum of 1983: NOAA Technical Memorandum NOS NGS-50.

Jet Propulsion Laboratory, 1992, Planetary Data System Standards Reference, Version 3.0: JPL D-7669, part 2.

Martin, T.Z., Martin, M.D., Davis, R.L., Mehlman, R., Braun, M., Johnson, M., 1988, Standards for the Preparation and Interchange of Data Sets, Version 1.1: Jet Propulsion Laboratory, D-4683.

Phillips, J.D., 1997, Potential-field software for the PC, version 2.2: U.S. Geological Survey Open-File Report 97-725, 34 p. [Online edition: ftp://greenwood.cr.usgs.gov/pub/open-file-reports/ofr-97-0725/ , software: ftp://musette.cr.usgs.gov/pub/pf/]

Snyder, J.P., 1987, Map Projections - A Working Manual: U.S. Geological Survey Professional Paper 1395.

U.S. Geological Survey, National Mapping Division, 1986, GCTP - General Cartographic Transformation Package, Software Documentation: U.S. Geological Survey, SD1-4-6.

Appendix A

Extracting grid data without using ODDF

The generic Fortran program below can extract the data from an ODDF grid file. A person reads the plaintext header and then modifies the code using the following steps.

The example grid header in the introductory chapter of this report contains the following attributes:

record_bytes = 7200
^qube = 2
axis_start = (-95.99167, 25.00833)
axis_interval = (.16666668E-01, .16666668E-01)
core_items = (1800, 1500)
core_item_type = real
core_item_bytes =4

The following relation holds: 7200 record_bytes = 1800 columns * 4 bytes. The data starts in file record 2 and continues for 1500 row records. The relevant numbers are updated in the code below and the program compiled and executed to translate the binary encoded grid values to a sequence of ASCII encoded records.

Begin example code

```
real*4  row(1800)

open( 10,  file='my_file',  status='old',  form= 'unformatted',  access='direct',  recl=7200 )

ncol = 1800
nrow = 1500
xs = -95.99167
ys =  25.00833
dx = .16666668E-01
dy = .16666668E-01

irec = 2
do irow = 1, nrow

  read( 10, rec=irec )  row

  y = ys + dy * float( irow -1 )

  do icol = 1, ncol
    x = xs + dx * float( icol - 1 )
    write( 11, '(2e16.8, 2i6, e16.8)'  ) x, y, icol, irow, row(icol)
  enddo

  irec = irec + 1
enddo
```

```
stop
end
```

End example code

Appendix B

Extracting point data without using ODDF

The posting file example is more complicated than the grid example because of the buffering of multiple posting records into a single file record (the express record) and the two different data types (character and real) in each logical record. The translation is a two step process, the first step reads the file record and the second cycles through the logical records writing to the ASCII output file.

In this appendix, "word" is applied to a grouping of 4 bytes, which may be interpreted as characters, a longword integer or a single precision floating point number. Although it is beyond this example, it may be noted the conversion between Most_Significant_Byte and Least_Significant_Byte ordering is done by reversing the byte order of **all** words in the express file record regardless of interpretation.

Derivation of the file record structure

The following steps derive the first and third values of the header that starts each binary express record. These words have a constant value in any one posting file. The first value gives the number of words in each posting logical record. The second word gives the number of logical records and has a range of zero to a maximum dependent on the record length. The third word gives the length of the record that remains after the 3 word header. The posting data occupies the space given by multiplying the values of the first and second words and pad values occupy any remaining space.

The example posting file header in the introductory chapter of this report contains the following attributes:

record_bytes = 2048
file_records = 366
^express_series = 3
logical_record_bytes = 20
id_bytes = 8

From the comments in the file label, we see the layout of the file record is three words describing the dimensions of the data array followed by the data array. In symbolic form we have: NWORD, NLOG, NTOTAL followed by DATA. The record contents are all grouped in 4 byte words of various data types.

The file record length is RECORD_BYTES / 4 = 512 words per record, and subtracting 3 header words leaves DATA a maximum 1-dimensional length of 509 (ie. NTOTAL = 509).

The LOGICAL_RECORD_BYTES = 20 so there are 5 words per logical record (ie. NWORD = 5). The maximum number of logical records per physical record is then 509 / 5 = 101 with 4 words remaining that we discard. Note we have derived NWORD and NTOTAL from the PDS header, their presence in each express file record is redundant and is used for file structure verification.

Derivation of the logical record structure

The length of each posting logical record is 5 words (NWORD=5) and since an id is present then the remainder of the logical record consists of 3 data channels.

Example code

The example code uses the 2-dimensional form of DATA to read only what we need, the pads at the end of each record are not read.

Begin example code

```
      dimension  data(5,101),  realid(2)
      character  charid*8
      equivalence  ( realid, charid )

      open( 10,  file='my_file',  status='old',  form= 'unformatted',  access='direct',  recl=2048 )

      do irec = 3, 366

      read( 10, rec=irec ) nword, nlog, ntotal, data

       do ilog = 1, nlog
         realid(1) = data(1,ilog)
         realid(2) = data(2,ilog)
         write( 11, 37 )  charid,  ( data(i,ilog), i = 3, 5 )
37       format( a8, 3e16.8 )
        enddo

      enddo

      stop
      end
```

End example code

Additional comments

Obtaining the id as characters from a real data array requires some sleight of hand in Fortran-77 since structured variables are not available. We equivalence 8 bytes of character data to a two word real number array in the example program above to perform the extraction. The experienced programmer can also extract the bytes from the data array and interpret them as a character string, but while more transportable, is more difficult than this example requires.

An alternate form of the 2-dimensional read given above uses a one-dimensional integer IDATA array:

read(10, rec=irec) nword, nlog, ntotal, (idata(i), i=1, ntotal) **or**
write(10, rec=irec) nword, nlog, ntotal, (idata(i), i=1, ntotal) ,

which is slightly more general but then a pointer is needed to step through IDATA by NWORDs to extract the posting records and is therefore more complicated from the user's point of view. In addition, the contents of IDATA would need an equivalence to floating point values. The express file record is created (written) with the integer 1-dimensional form to force the contents of all the binary words to known values and byte order.