FreeForm

A Flexible System of Format Descriptions for Data Access

The User's Guide

November 1998

NOTICE

This document was originally written at the National Oceanic and Atmospheric Administration's National Environmental Satellite, Data, and Information Service, which is part of the National Geophysical Data Center in Boulder, Colorado.

This document has been updated to include information on FreeForm ND, the last release of FreeForm. FreeForm is now supported only for use with the Distributed Oceanographic Data System; see http://dods.gso.uri.edu for more information.

THANKS

To the developers of FreeForm and FreeForm ND: Tom Carey, S.D Davis, Liping Di, Rich Fozzard, Kevin Frender, Ted Habermann, Mark A. Ohrenshall, John Rex, Mark Van Gorp, and others. Special thanks to Mark A. Ohrenshall for his meticulous documentation of the design and development of FreeForm ND.

CHAPTER 1

Introduction

The FreeForm ND Data Access System is a flexible system for specifying data formats to facilitate data access, management, and use. How many data sets have you not examined or used because they were not in the correct format for your applications? How many others have foregone analysis of your data for the same reason? FreeForm ND can save you countless hours of changing the formats of data sets prior to analyzing them.

The large variety of data formats is a primary obstacle in the way of creating flexible data management and analysis software. FreeForm ND was conceived, developed, and implemented at the National Geophysical Data Center (NGDC) to alleviate the problems that occur when you need to use data sets with varying native formats or to write format-independent applications.

The most recent version of FreeForm, FreeForm ND, is now supported only for use with the Distributed Oceanographic Data System (DODS).

The Format Problem

Programmers can readily describe a format for a specific data set, but a compiled application cannot be used with other data sets until either the data or the program is modified. Two possible methods for handling data in a variety of formats are to reformat all the data into a standard format or to develop programs that can read data in many different formats.

Standard Formats

A number of standard formats have been proposed over the years and the specifications for these formats have generally improved. However, standard formats do not enjoy widespread use, which will probably continue to be the case.

Many scientists have large amounts of data on hand in non-standard formats. Converting to standard formats is cumbersome and time-consuming. In addition, there are so many standard formats that format-independent applications are required even if only standard formats are used.

Smart Programs

Software developers can create programs that use data in many different formats. This approach has several advantages:

- The programs are flexible enough to allow the introduction of new data formats.
- The scientist collecting the data is not forced to conform to any single data format.
- The information contained in the original data is not lost through reformatting.

The FreeForm ND Solution

FreeForm ND uses a variation on the smart program approach. With FreeForm ND, you specify formats outside application programs by writing text files that describe the formats of your data sets. The applications then use these format files as they process data. FreeForm ND-based applications are in effect format-independent and you do not need to modify the data or the applications.

FreeForm ND provides a mechanism for data description that is flexible and easy to use. A set of ready-to-use programs for manipulating a wide variety of data in standard and non-standard formats is also provided. FreeForm ND lets you concentrate on your specialty rather than trying to figure out how to access and manipulate data in multiple formats. Additionally, the application programmer can use FreeForm ND libraries and data constructs to develop format-independent applications.

The FreeForm ND System

The FreeForm ND Data Access System comprises a format description mechanism, a library of C functions, object-oriented constructs for data structures, and a set of programs (built using the FreeForm ND library and data objects) for manipulating data.

There are two types of FreeForm ND users. Data users and providers create format description files and run FreeForm ND programs such as **newform**. Programmers use the FreeForm ND library and data objects to write data management and analysis applications.

FreeForm ND includes the following programs for accessing and manipulating data in various formats:

newform	reformats data
readfile	reads binary files
checkvar	creates variable summaries

The FreeForm ND data objects provide an interface between application and data files.

Programmers use the FreeForm ND library routines to develop applications.

Data users and providers write format description files that FreeForm ND-based programs use to correctly access data.

FreeForm ND Files

The FreeForm ND file set is available only through the DODS distribution system. You can download the FreeForm ND files from the DODS website (http://dods.gso.uri.edu).

About this Guide

This guide provides instructions for writing format descriptions, using FreeForm ND programs, and writing your own FreeForm ND-based applications. The content of each chapter is outlined below.

This chapter introduces the FreeForm ND Data Access System and summarizes typographic conventions and the contents of this guide.

"Quick Tour of FreeForm ND" provides a brief introduction to writing format descriptions and using several of the FreeForm ND programs.

"Format Descriptions for Tabular Data" provides detailed information about writing format descriptions to facilitate access to data in tabular formats.

“Format Descriptions for Array Data” provides detailed information about writing format descriptions to facilitate access to data in non-tabular (array) formats.

“FreeForm ND Conventions” presents FreeForm ND file name conventions, the search rules for locating format files, and standard command line arguments for FreeForm ND programs.

“Format Conversion” shows you how to use the FreeForm ND program **newform** to convert data from one format to another and also how to read the data in a binary file.

“Conversion Variables” discusses FreeForm ND conversion variables, which let you translate between a number of representations of space and time values.

“Header Formats” tells you how to work with header formats.

“Data Checking” discusses the FreeForm ND program **checkvar**, which you can use to check data distribution and quality.

“Developing FreeForm ND Applications” summarizes how to use the FreeForm ND Data Access System to build FreeForm ND-based programs.

“Conversion Variable Names” lists the conversion variable names that FreeForm ND recognizes.

“Error Handling” presents a list of common FreeForm ND error messages.

“Query Syntax” lists the operators, symbols, and functions you can use to construct queries.

Conventions

The following typographic conventions are used throughout this guide (except in "Query Syntax").

- File names, executable program names, commands, and user input are in **boldface**.
- Emphasized words, book titles, and axis names (e.g., *x* axis) are in *italics*.
- Code examples, data file contents, and system output are in `this typeface`.
- Key names (e.g., Return) have an initial capital letter.
- A position box is used to indicate column position of field values in data files. It is shown at the beginning of a data list in the documentation, but does not appear in the data file itself.

```
         1         2         3         4         5         6
12345678901234567890123456789012345678901234567890123456789012345
67890
```

CHAPTER 2

Quick Tour of FreeForm ND

This chapter provides you a quick introduction to writing format descriptions and using several FreeForm ND programs. You will look at a format description file, convert data from one format to another, read the data in a binary file, and create summary files. This chapter introduces only tabular format data (arranged in rows and columns). See “Format Descriptions for Array Data” on page 33 for information on handling data in non-tabular formats.

Writing Format Descriptions

You can easily create FreeForm ND format description files that describe the formats of input and output data and headers. Free-Form ND-based programs then use these files to correctly access and manipulate data in various formats. An example format description file is shown and described below.

For complete information about writing format descriptions, see the next two chapters.

latlon.fmt

```
/ This is the format description file for data files lat-
lon.bin
/ and latlon.dat. Each record in both files contains two
fields,
/ latitude and longitude.

binary_data "binary format"
latitude 1 8 double 6
longitude 9 16 double 6

ASCII_data "ASCII format"
latitude 1 10 double 6
longitude 12 22 double 6
```

> *Note:* You can display **latlon.fmt** on your screen by changing to the directory containing the FreeForm ND example files and using the appropriate command (**cat** or **more**).

This format description file contains two format descriptions. The first describes data in the binary data file **latlon.bin** and the second describes data in the ASCII data file **latlon.dat** (contents shown below).

The binary and ASCII variables both have the same names. The binary variables are defined to occupy 8 bytes each (positions 1-8 and 9-16). The ASCII variable `latitude` occupies 10 bytes (positions 1 to 10) and `longitude` occupies 11 bytes (positions 12-22). Both the binary and ASCII variables are stored as doubles because they have more than seven digits and include a decimal point (see the **lat-lon.dat** listing below). The precision of 6 for all the variables indicates that there are six digits to the right of the decimal point.

latlon.dat

```
         1         2         3         4         5
123456789012345678901234567890123456789012345678
-47.303545 -176.161101
 -0.928001    0.777265
-28.286662   35.591879
 12.588231  149.408117
-83.223548   55.319598
 54.118314 -136.940570
 38.818812   91.411330
-34.577065   30.172129
 27.331551 -155.233735
 11.624981 -113.660611
 77.652742  -79.177679
 77.883119  -77.505502
-65.864879  -55.441896
-63.211962  134.124014
 35.130219 -153.543091
 29.918847  144.804390
-69.273601   38.875778
-63.002874   36.356024
 35.086084  -21.643402
-12.966961   62.152266
```

Changing Formats

The FreeForm ND program **newform** is used to convert data from one format to another. Format descriptions for all the data (input and output) involved in the conversion must be included in a format description file.

In this example, you will use **newform** to convert ASCII data in the input file **latlon.dat** to binary data in the output file **latlon2.bin**. First you need to create a format description file like the following that describes the data in these two files.

latlon2.fmt

```
/ This is the format description file for data files lat-
lon.dat
/ and latlon2.bin. Each record in both files contains two
fields,
/ latitude and longitude.

ASCII_data "ASCII format"
latitude 1 10 double 6
longitude 12 22 double 6

binary_data "binary format"
latitude 1 4 long 6
longitude 5 8 long 6
```

The ASCII and binary variables both have the same names. The ASCII variable `latitude` occupies 10 bytes (positions 1-10) and `longitude` occupies 11 bytes (positions 12-22). The ASCII variables are defined to be of type `double` because they have more than seven digits and include a decimal point. (See the **latlon.dat** listing above.) The binary variables are defined to occupy four bytes each (positions 1-4 and 5-8) and to be of type `long`. The precision for all is 6.

To convert the ASCII data in latlon.dat to binary data, you would:

1. Change to the directory that contains the FreeForm ND files.
2. Enter the following command:

   n**ewform latlon.dat -f latlon2.fmt -o latlon2.bin**

   This command creates a new binary data file called **latlon2.bin** with the 20 latitude and longitude values in **latlon.dat** stored as binary longs.

For complete information about using **newform**, see “Format Conversion” on page 55.

Viewing Binary Data Files

The FreeForm ND Data Access System includes an interactive utility program, **readfile**, for reading binary files. You can use readfile to read the binary file **latlon2.bin** and check that the data are correct.

To read **latlon2.bin**, you would:

1. Change to the directory that contains the FreeForm ND files.
2. On the command line, enter **readfile latlon2.bin**
3. The data are stored as longs, so enter **l** to view the first value.

   The number `-47303545`, corresponding to the first number in **latlon.dat** (but with implied precision, i.e., without a decimal point), should appear.
4. To check additional numbers, continue to enter **l** or press Return.

   The numbers should correspond to those in **latlon.dat**.
5. When you want to quit **readfile**, enter **q**.

For complete information about using **readfile**, see "Format Conversion" on page 55.

Creating Summary Files

The FreeForm ND-based utility program **checkvar** creates a summary file for each variable in a data file, a list of maximum and minimum values, and a summary of processing activity. A variable summary file (also called a histogram data file) contains histogram information that shows the variable's distribution in the data file. In this example, you will use **checkvar** to create a processing summary file and variable summary files for the two variables latitude and longitude in the file **latlon2.bin**.

Generating the Summaries

To create summary files for latlon2.bin, you would:

1. Change to the directory that contains the FreeForm ND files.
2. Enter the following command:

   checkvar latlon2.bin -o checkvar.out

   A summary of processing information and the maximum and minimum for each variable are displayed on the screen. The following three files are created:

   a **checkvar.out** recaps processing activity, maximums and minimums

   b **latitude.lst** shows distribution of the latitude values in latlon2.bin

   c **longitude.lst** shows distribution of the longitude values in **latlon2.bin**
3. To view the files, use the appropriate command, i.e., **cat** or **more**.

Interpreting the Summaries

The three files output by **checkvar** are shown and discussed below. To remind yourself of the input values, refer to **latlon.dat** since it contains the same values as **latlon2.bin** in ASCII representation.

checkvar.out

```
Input file: latlon2.bin
No requested precision, Approximate number of sorting
bins = 100

Input data format(latlon2.fmt)
binary_input_data"binary format"
The format contains 2 variables; length is 8.

Output data format(latlon2.fmt)
ASCII_output_data"ASCII format"
```

```
The format contains 2 variables; length is 24.

Histogram data precision: 5, Number of sorting bins: 20
latitude: 20 values read
minimum: -83.223548 found at record 5
maximum:  77.883119 found at record 12
Summary file: latitude.lst

Histogram data precision: 5, Number of sorting bins: 20
longitude: 20 values read
minimum: -176.161101 found at record 1
maximum:  149.408117 found at record 4
Summary file: longitude.lst
```

The processing summary file **checkvar.out** first shows the name of the input data file (`latlon2.bin`). Since precision and a maximum number of bins were not specified on the command line, `No requested precision` and the default value for sorting bins of `100` are shown.

A summary of each format shows the type of format (in this case, `Input data format` and `Output data format`) and the name of the format file containing the format descriptions (`latlon2.fmt` describes both the input and output formats; note that **checkvar** ignores output formats). Next, you see the format descriptor as resolved by FreeForm ND (e.g., `binary_input_data`) and the format title (e.g., `"binary format"`). Then the number of variables in a record and total record length are given; for ASCII, record length includes the end-of-line character.

A section for each variable processed by **checkvar** indicates the histogram precision and actual number of sorting bins. Under some circumstances, the precision of values in the histogram file may be different than the precision you specified on the command line. No precision was specified on the command line in this case, so the default maximum precision of 5 is used. The second line shows the name of the variable (`latitude` and `longitude`) and the number of values in the data file for the variable (`20` for both `latitude` and `longitude`).

The minimum and maximum values for the variable are shown next (`-83.223548` is the minimum and `77.883119` is the maximum value for `latitude`). The maximum and minimum values are given here with a precision of 6, which is the precision specified in the relevant format description file. The locations of the maximum and minimum values in the input file are indicated. (`-83.223548` is the fifth latitude value in **latlon2.bin** and `77.883119` is the twelfth).

Finally, the name of the histogram data (or variable summary) file generated for each variable is given. The two example histogram files, `latitude.lst` and `longitude.lst`, are shown next.

latitude.lst	**longitude.lst**
-83.223551	-176.161111
-69.273611	-155.233741
-65.864881	-153.543101
-63.211971	-136.940571
-63.002881	-113.660621
-47.303551	-79.177681
-34.577071	-77.505511
-28.286671	-55.441901
-12.966971	-21.643411
-0.928011	0.777261
11.624981	30.172121
12.588231	35.591871
27.331551	36.356021
29.918841	38.875771
35.086081	55.319591
35.130211	62.152261
38.818811	91.411331
54.118311	134.124011
77.652741	144.804391
77.883111	149.408111

The histogram files consist of two columns. The first indicates boundary values for data bins and the second gives the number of data points in each bin. The boundary values are determined dynamically by **checkvar** and often do not correspond exactly to data values in the input file, even if the **checkvar** and data file precisions are the same.

The first data bin in **latitude.lst** contains data points in the range -83.22355 (inclusive) to -69.27361 (exclusive). The first bin has one data point, `-83.223548` (refer to **latlon.dat** on page 7). The fifth data bin contains latitude values from -63.00288 (inclusive) to -47.30355 (exclusive); the data point in the fourth bin is `-63.002874`.

For complete information about using **checkvar**, see "Data Checking" on page 101.

CHAPTER 3

Format Descriptions for Tabular Data

Format descriptions define the formats of input and output data and headers. FreeForm ND provides an easy-to-use mechanism for describing data. FreeForm ND programs and FreeForm ND-based applications that you develop use these format descriptions to correctly access data. Any data file used by FreeForm ND programs must be described in a format description file.

This chapter explains how to write format descriptions for data arranged in tabular format — rows and columns — only. For data in non-tabular formats, see the next chapter.

FreeForm ND Variable Types

The data sets you produce and use may contain a variety of variable types. The characteristics of the types that FreeForm ND supports are summarized in the table below, which is followed by a description of each type.

Name	Minimum Value	Maximum Value	Binary Size(bytes)	Precisio (signifi
char			*	
uchar	0	255	1	
short	-32,767	32,767	2	
ushort	0	65,535	2	
long	-2,147,483,647	2,147,483,647	4	
ulong	0	4,294,967,295	4	
float	10^{-37}	10^{-38}	4	6**
double	10^{-307}	10^{-308}	8	15**
constant			*	
initial			record length	
convert			*	

* User-specified
** Can vary depending on environment

> *Note:* The sizes in the table are machine-dependent. Those given are for most Unix workstations.

char

The **char** variable type is used for character strings. Variables of this type, including numerals, are interpreted as characters, not as numbers.

uchar

The **uchar** (unsigned character) variable type can be used for integers between 0 and 255 (28- 1). Variables that can be represented by the **uchar** type (for example: month, day, hour, minute) occur in many data sets. An advantage of using the **uchar** type in binary formats is that only one byte is used for each variable. Variables of this type are interpreted as numbers, not characters.

short

A **short** variable can hold integers between -32,767 and 32,767 (215-1). This type can be used for signed integers with less than 5 digits, or for real numbers with a total of 4 or fewer digits on both sides of the decimal point (-99 to 99 with a precision of 2, -999 to 999 with a precision of 1, and so on).

ushort

A **ushort** (unsigned short) variable can hold integers between 0 and 65,535 (216 - 1).

long

A **long** variable can hold integers between -2,147,483,647 and +2,147,483,647 (231 - 1). This variable type is commonly used to represent floating point data as integers, which may be more portable. It can be used for numbers with 9 or fewer digits with up to 9 digits of precision, for example, latitude or longitude (-180.000000 to 180.000000).

ulong

The **ulong** (unsigned long) variable type can be used for integers between 0 and 4,294,967,295 (232 - 1).

float, double

Numbers that include explicit decimal points are either **float** or **double** depending on the desired number of digits. A **float** has a maximum of 6 significant digits, a **double** has 15 maximum. The extra digits of a **double** are useful, for example, for precisely specifying time of day within a month as decimal days. One second of time is approximately 0.00001 day. The number specifying day (maximum = 31) can occupy up to 2 digits. A **float** can therefore only specify decimal days to a whole second (31.00001 occupies seven digits). A **double** can, however, be used to track decimal parts of a second (for example, 31.000001).

constant

FreeForm ND has two variable types, **constant** and **initial**, for sequences of characters (or bytes) that are the same for all records in a file. A **constant** variable is placed into the output buffer on initialization. The constant value is the same as the name of the variable. For example, given the variable description below:

```
NGDCDATA 1 8 constant 0
```

the string NGDCDATA, which is both the variable name and value, is placed in characters 1-8 of each output record.

FreeForm ND recognizes the special constant EOL as an end-of-line character, which is used with multi-line records. The variable descriptions shown next are for a data record that includes several variables, the end-of-line character, then several more variables.

```
year 1 2 short 0
      .
      .   (more variables)
      .
latitude_sec 75 80 float 2
EOL 81 82 constant 0
longitude_deg 83 89 float 3
longitude_min 72 78 float 2
      .
      .   (remaining variables)
      .
```

The variable longitude_deg starts a new line in the data file.

initial

The variable type **initial** can be used when you want to set more than one constant value at a time. It provides an initialization template for the output record. This template is read from a file with the same name as the **initial** variable. For example, suppose you have the following variable description:

```
seattle.ini 1 80 initial 0
```

The **initial** variable is named `seattle.ini`, so the initialization template file `seattle.ini` is read and used to initialize the output records. Assume the Seattle template contains the following values, which are written to an earthquake record:

```
SEA19                                    SEA
```

The other values in the output record are written over this template resulting in a record that looks like the following:

```
          SEA19   5  -146.34172   -47.39710   1011 SEA
910802
```

Note: The length of the template file must equal the length of the record in the output format. The file name and extension are of your choosing.

convert

The **convert** variable type allows you to access an extensive set of functions for constructing output variables that do not exist in input files, but can be computed from variables which do. FreeForm ND can transparently identify and call conversion functions during the data access process if you use properly named input and output variables in variable descriptions.

See “Conversion Variables” on page 75 for examples and “Conversion Variable Names” on page 123 for a complete list of names for conversion variables.

header

Older versions of FreeForm ND included header variables. You can now specify header formats in format description files.

For details, see “Format Descriptions” on page 29 and also “Header Formats” on page 89.

FreeForm ND File Types

FreeForm ND supports binary, ASCII, and dBASE file types. Binary data are stored in a fixed amount of space with a fixed range of values. This is a very efficient way to store data, but the files are machine-readable rather than human-readable. Binary numbers can be integers or floating point numbers.

Numbers and character strings are stored as text strings in ASCII. The amount of space used to store a string is variable, with each character occupying one byte.

The dBASE file type, used by the dBASE product, is ASCII text without end-of-line markers.

Format Description Files

Format description files accompany data files. A format description file can contain descriptions for one or more formats. You include descriptions for header, input, and output formats as appropriate. Format descriptions for more than one file may be included in a single format description file.

An example format description file is shown next. The sections that follow describe each element of a format description file.

```
/ This format description file is for   / comment lines
/ data files latlon.bin and latlon.dat.

binary_data "Default binary format"     / format type and title
latitude 1 4 long 6                     / variable description
longitude 5 8 long 6                    / variable description
```

```
                                          / blank line(s) mark the end
                                          / of a format description

ASCII_data "Default ASCII format"         / format type and title
latitude 1 10 double       6              / variable description
longitude 12 22 double 6                  / variable description
```

You can include blank lines between format descriptions and comments in a format description file as necessary. Optional comment lines begin with a slash (/). FreeForm ND ignores comments.

Format Descriptions

A format description file comprises one or more format descriptions. A format description consists of a line specifying the format type and title followed by one or more variable descriptions.

Example:

```
/ This is an example format description
binary_data "Default binary format"       / format type and title
latitude 1 4 long 6                       / variable description
longitude 5 8 long 6                      / variable description
```

Format Type and Title

A line specifying the format type and title begins a format description. Format descriptors, for example, `binary_data`, are used to indicate format type to FreeForm ND. The format title, for example, `"Default binary format"`, briefly describes the format. It must be surrounded by quotes and follow the format descriptor on the same line. The maximum number of characters for the format title is 80 including the quotes.

Format Descriptors

Format descriptors indicate (in the order given) file type, read/write type, and file section. Possible values for each descriptor component are shown in the following table.

File Type	Read/Write Type (optional)	File Section
ASCII binary dBASE	input output	data file_header record_header file_header_separate* record_header_separate*

*The qualifier `separate` indicates there is a header file separate from the data file.

The components of a format descriptor are separated by underscores (_). For example, `ASCII_output_data` indicates that the format description is for ASCII data in an output file. The order of descriptors in a format description should reflect the order of format types in the file. For instance, the descriptor `ASCII_file_header` would be listed in the format description file before `ASCII_data`. The format descriptors you can use in FreeForm ND are listed in the next table.

Data	Header	S
XXX_data XXX_input_data XXX_output_data	XXX_file_header XXX_file_header_separate XXX_record_header XXX_record_header_separate XXX_input_file_header XXX_input_file_header_separate XXX_input_record_header XXX_input_record_header_separate XXX_output_file_header XXX_output_file_header_separate XXX_output_record_header XXX_output_record_header_separate	

where `XXX` = `ASCII`, `binary`, or `dBASE`
Example: `XXX_data` = `ASCII_data`, `binary_data`, or `dBASE_data`

* The `RETURN` descriptor lets FreeForm ND skip over end-of-line characters in the data.
**The `EOL` descriptor is a constant indicating an end-of-line character should be inserted in a multi-line record.

For more information about header formats, see “Header Formats” on page 89.

Variable Descriptions

A variable description defines the name, start and end column position, type, and precision for each variable. The fields in a variable description are separated by white space. Two variable descriptions are shown below with the fields indicated. Each field is then described.

```
/ Here are two example variable descriptions.
```

```
latitude        1     10    double      6
longitude       12    22    double      6
name            start end   type        precision
```

Name

The variable name is case-sensitive, up to 63 characters long with no blanks. The variable names in the example are `latitude` and `longitude`. If the same variable is included in more than one format description within a format description file, its name must be the same in each format description.

Start Position

The column position where the first character (ASCII) or byte (binary) of a variable value is placed. The first position is 1, not 0. In the example, the variable `latitude` is defined to start at position 1 and `longitude` at 12.

End Position

The column position where the last character (ASCII) or byte (binary) of a variable value is placed. In the example, the variable `latitude` is defined to end at position 10 and `longitude` at 22.

Type

The variable type can be a standard type such as **char**, **float**, **double**, or a special FreeForm ND type. The type for both variables in the example is **double**. See “FreeForm ND Variable Types” on page 23 for descriptions of supported types.

Precision

Precision defines the number of digits to the right of the decimal point. For **float** or **double** variables, precision only controls the number of digits printed or displayed to the right of the decimal point in an ASCII representation. The precision for both variables in the example is 6.

CHAPTER 4

Format Descriptions for Array Data

If the tabular format discussed in the previous chapter doesn't describe your data well, FreeForm ND's array notation may prove useful. Describing a data file's organization as a set of as n-dimensional arrays allows for much more flexibility in writing format definitions. It also enables subsetting, pixel-manipulation, and reorienting arrays of arbitrary dimension and character.

Array Descriptor Syntax

FreeForm ND allows you to describe the same fundamental FreeForm ND data types in array notation. The arrays can have any number of dimensions, any number of elements in each dimension, and either an increasing or a decreasing sequencing in each dimension. Furthermore, elements in any dimension may be separated from each other (demultiplexed) and may even be placed in separate files. However, every element of an array must be of the same type.

Array descriptors are a string of *n* dimension descriptions for arrays of *n* dimensions. FreeForm ND accepts descriptions with the most significant dimension first (i.e. row-major order for 2 dimensions, plane-major order in 3 dimensions).

Individual dimension descriptions are enclosed in brackets. Each dimension description can contain various keywords and values which specify how the dimension is set up. Some of the specifications are optional; if you do not specify them, they default to a specific value.

> *Note:* You must not mix array and tabular formats within the input and output sections of the format definition file. Only one type of notation can be used within each section of the format description file, although the sections may use different forms. For example, a file's input format could use array definitions, but the output format might be entirely tabular.

The dimension description variables include:

dimension name (REQUIRED)

A name for the dimension. This can be any ASCII string enclosed in double-quotes ("). The name for each dimension must be unique throughout the array descriptor.

This example specifies that a dimension named "latitude" exists:

```
["latitude" 0 to 180]
```

starting and ending indices (REQUIRED)

A starting and ending index specifying a range for the dimension. The starting and ending indices are specified as integers separated by the word "to" following the dimension name. As long as both numbers are integral, there are no other restrictions on their values.

This example specifies that the dimension "temperature" has indices ranging from -50 to +50:

```
["temperature" -50 to 50]
```

granularity (optional)

A specification for the density of elements in the indices. The number provided after the "by" keyword specifies how many index positions are to be skipped to find the next element.

This example specifies that index values 0, 50, 100, 150 and 200 are the only valid index values for the dimension "height":

```
["height" 0 to 200 by 50]
```

grouping (optional)

A specification for splitting an array across "partitions" (files or buffers in memory). The number provided after the "gb" or "groupedby" keyword specifies how many elements of the dimension are in each partition. If no value is specified, the default is 0 (no partitioning). Each partition must have the same number of elements. Every more-significant dimension description (those to the left) must also have a grouping specified- "dangling" grouping specifications are not allowed. If a dimension is not partitioned, but is required to have a grouping specification because a less-significant dimension is partitioned, a grouping of M can be specified, where M = absolute value(end index - starting index) + 1

This example specifies that the dimension "latitude" is partitioned into 9 chunks of 10 "bands" of latitude each:

```
["latitude" 1 to 90 gb 10]
```

separation (optional)

A specification for "unused space" in the array. The number provided after the "sb" or "separatedby" keyword specifies how many bytes of data following each element in the dimension should not be considered part of the array. An "element in the dimension" is considered to be everything which occurs in one index of that dimension. separation takes on a slightly different meaning if the dimension also has a specified grouping. In dimensions with a specified grouping, the separation occurs at the end of each partition, not after every element.

This example specifies a 2-dimensional array with 4 bytes between the elements in the "columns" and an additional 2 bytes at the end of every row:

```
["lat" -90 to 90 sb 2]["lon" -180 to 179 sb 4]
```

Handling Newlines

The convention of expecting a newline to follow each record of ASCII data becomes troublesome when dealing with array data, especially when expressed using format description notation that is intended for tabular data. It is the FreeForm ND convention that there is an implicit newline after the last variable of an ASCII format.

For example, these two format descriptions are equivalent:

```
ASCII_data "broken time — BIP"
year 1 2 uint8 0
month 3 4 uint8 0
day 5 6 uint8 0
hour 7 8 uint8 0
minute 9 10 uint8 0
second 11 14 uint16 2

dBASE_data "broken time — BIP"
year 1 2 uint8 0
month 3 4 uint8 0
day 5 6 uint8 0
hour 7 8 uint8 0
minute 9 10 uint8 0
second 11 14 uint16 2
EOL 15 16 constant 0
```

However, the EOL variable shown here assumes that newlines are two bytes long, which is true only on a PC. FreeForm ND adjusts to this by assuming that ASCII data always has native newlines, and it updates the starting and ending position of EOL variables and subsequent variables accordingly.

The EOL variable is typically used to define a record layout that spans multiple lines. However, the EOL variable in combination with the dBASE format type can completely replace the ASCII format type.We recommend using the dBASE format type when describing ASCII arrays, to ensure that separation, if specified, takes into account the length of any newlines.

In this output format a newline separates each band of data, but it would be just as easy to omit the newlines entirely.

```
dBASE_input_data "broken time — BIP"
year 1 2 array["x" 1 to 10 sb 14] of uint8 0
month 3 4 array["x" 1 to 10 sb 14] of uint8 0
day 5 6 array["x" 1 to 10 sb 14] of uint8 0
hour 7 8 array["x" 1 to 10 sb 14] of uint8 0
minute 9 10 array["x" 1 to 10 sb 14] of uint8 0
second 11 14 array["x" 1 to 10 sb 12] of uint16 2
EOL 15 16 array["x" 1 to 10 sb 14] of constant 0

dBASE_output_data "broken time - BSQ"
year 1 2 array["x" 1 to 10] of uint8 0
EOL 21 22 constant 0
month 23 24 array["x" 1 to 10] of uint8 0
EOL 43 44 constant 0
day 45 46 array["x" 1 to 10] of uint8 0
EOL 65 66 constant 0
hour 67 68 array["x" 1 to 10] of uint8 0
EOL 87 89 constant 0
minute 90 91 array["x" 1 to 10] of uint8 0
EOL 110 111 constant 0
second 112 115 array["x" 1 to 10] of uint16 2
EOL 132 133 constant 0
```

Note: The separation size now takes into account the two-character PC newline. To use this format description with a native ASCII file on UNIX platforms, it would be necessary to change the separation sizes of 12 and 14 to 11 and 13, respectively.

Examples

The following examples should be helpful in understanding the array notation.

Tabular versus Array Descriptions

Array notation can simply replace the tabular format description, as in these examples.

A single element can be described in tabular format:

```
year 1 2 uint8 0
```

or as an array:

```
year 1 2 array["x" 1 to 10] of uint8 0
```

An image file can be described in tabular format:

```
binary_input_data "grid data"
data 1 1 uint8 0
```

or as an array:

```
        binary_input_data "grid data"
        data 1 1 array["rows" 1 to 180] ["cols" 1 to 360]
of uint8 0
```

Multiplexed data can be described in tabular format:

```
ASCII_data "broken time — tabular"
year 1 2 uint8 0
month 3 4 uint8 0
day 5 6 uint8 0
hour 7 8 uint8 0
```

```
minute 9 10 uint8 0
second 11 14 uint16 2
```

or as an array:

```
ASCII_data "broken time -- BIP"
year 1 2 array["x" 1 to 10 sb 12] of uint8 0
month 3 4 array["x" 1 to 10 sb 12] of uint8 0
day 5 6 array["x" 1 to 10 sb 12] of uint8 0
hour 7 8 array["x" 1 to 10 sb 12] of uint8 0
minute 9 10 array["x" 1 to 10 sb 12] of uint8 0
second 11 14 array["x" 1 to 10 sb 10] of uint16 2
```

These two format descriptions communicate much the same information, but the array example also indicates that the data file is blocked into ten data values for each variable.

In this example, the data is not multiplexed:

```
ASCII_data "broken time -- BSQ"
year 1 2 array["x" 1 to 10] of uint8 0
month 21 22 array["x" 1 to 10] of uint8 0
day 41 42 array["x" 1 to 10] of uint8 0
hour 61 62 array["x" 1 to 10] of uint8 0
minute 81 82 array["x" 1 to 10] of uint8 0
second 101 104 array["x" 1 to 10] of uint16 2
```

The starting position indicates the file offset of the first element of each array, the same as with the alternative definition given for starting position in tabular data format descriptions.

Array Manipulation

Consider a 6x6 array of data with an "XXXX" header and a "YY" trailer on each line. Each data element is a space, a row ("y") index, a comma, and a column ("x") index, as shown below:

```
XXXX 0,0 0,1 0,2 0,3 0,4 0,5YY
XXXX 1,0 1,1 1,2 1,3 1,4 1,5YY
XXXX 2,0 2,1 2,2 2,3 2,4 2,5YY
```

```
XXXX 3,0 3,1 3,2 3,3 3,4 3,5YY
XXXX 4,0 4,1 4,2 4,3 4,4 4,5YY
XXXX 5,0 5,1 5,2 5,3 5,4 5,5YY
```

The goal is to produce a data file that looks like the data below. To do that, we need to strip the headers and trailers, and transpose rows and columns:

```
 0,0 1,0 2,0 3,0 4,0 5,0
 0,1 1,1 2,1 3,1 4,1 5,1
 0,2 1,2 2,2 3,2 4,2 5,2
 0,3 1,3 2,3 3,3 4,3 5,3
 0,4 1,4 2,4 3,4 4,4 5,4
 0,5 1,5 2,5 3,5 4,5 5,5
```

The key to writing the input format description is understanding that the input data file is composed of four interleaved arrays:

1. The "XXXX" headers
2. The data
3. The "YY" trailers
4. The newlines

The array of headers is a one-dimensional array composed of six elements (one for each line) with each element being four characters wide and separated from the next element by 28 bytes (24 + 2 + 2 — 24 bytes for a row of data plus 2 bytes for the trailer plus two bytes for the newline).

The array of data is a two-dimensional array of six elements in each dimension with each element being four characters wide, each row is separated from the next by eight bytes (columns are adjacent and so have zero separation), and the first element begins in the fifth byte of the file (counting from one).

The array of trailers is a one-dimensional array composed of six elements with each element being two characters wide, each element is separated from the next by 30 bytes, and the first element begins in the 29th byte of the file.

The array of newlines is a one-dimensional array composed of six elements with each element being two characters wide (on a PC), each element is separated from the next by 30 bytes, and the first element begins in the 31st byte of the file.

The FreeForm ND input format description needed is:

```
  dBASE_input_data "one"
  headers 1 4 ARRAY["line" 1 to 6 separation 28] OF text 0
  data 5 8 ARRAY["y" 1 to 6 separation 8]["x" 1 to 6] OF
text 0
  trailers 29 30 ARRAY["line" 1 to 6 separation 30] OF
text 0
  PCnewline 31 32 ARRAY["line" 1 to 6 separation 30] OF
text 0
```

The output data is composed of two interleaved arrays:

1. The data
2. The newlines

The array of data now has a separation of two bytes between each row, the first element begins in the first byte of the file, and the order of the dimensions has been switched.

The array of newlines now has a separation of 24 bytes and the first element begins in the 25th byte of the file. Each array can be operated on independently. In the case of the data array we simply transposed rows and columns, but we could do other reorientations as well, such as resequencing elements within either or both dimensions.

The FreeForm ND output format description needed is:

```
  dBASE_output_data "two"
  data 1 4 ARRAY["x" 1 to 6 separation 2]["y" 1 to 6] OF
text 0
  PCnewline 25 26 ARRAY["line" 1 to 6 separation 24] OF
text 0
```

Sampling and Data Manipulation

With a wider range of descriptive possibilities, FreeForm can more easily be used for sampling and subsetting data, as in these examples.

The following array descriptor pair subsets a two-dimensional array, retrieving one quarter (the north-west quarter of the earth).

INPUT: `["latitude" -90 to 90] ["longitude" -179 to 180]`
OUTPUT: `["latitude" 0 to 90] ["longitude" -179 to 0]`

The following array descriptor pair flips a two-dimensional array row-wise (vertically).

INPUT: `["row" 0 to 100] ["column" 13 to 42]`
OUTPUT: `["row" 100 to 0] ["column" 13 to 42]`

The following array descriptor pair rotates a two-dimensional array 90 degrees (exchanging rows and columns).

INPUT: `["row" 0 to 10] ["column" 0 to 42]`
OUTPUT: `["column" 0 to 42] ["row" 0 to 10]`

The following array descriptor pair outputs every other plane from a three-dimensional array (essentially cutting the depth resolution in half).

INPUT: `["plane" 1 to 18] ["row" 0 to 10] ["column" 0 to 42]`
OUTPUT: `["plane" 1 to 18 by 2] ["row" 0 to 10] ["column" 0 to 42]`

The following array descriptor pair replicates every plane from a three-dimensional array three times (essentially tripling the depth).

INPUT: `["plane" 1 to 54 by 3] ["row" 0 to 10] ["column" 0 to 42]`
OUTPUT: `["plane" 1 to 54] ["row" 0 to 10] ["column" 0 to 42]`

This array descriptor pair outputs the middle 1/27 of a three dimensional array with depth and width exchanged and height halved and flipped:

INPUT: `["plane" 1 to 27] ["row" 1 to 27] ["column" 1 to 27]`
OUTPUT: `["column" 10 to 18] ["row" 18 to 10 by 2] ["plane" 10 to 18]`

CHAPTER 5

FreeForm ND Conventions

File name conventions have been defined for FreeForm ND. If you follow these conventions, FreeForm ND can locate format files through a default search sequence. Using the file name conventions also lets you reduce the number of arguments on the command line. In addition to standard file names, FreeForm ND programs recognize various standard command line arguments.

File Name Conventions

Naming conventions have been established for files accessed by FreeForm ND. Although you are not required to follow these conventions, using them lets you enter abbreviated commands when you are using FreeForm ND-based programs. FreeForm ND can then automatically execute several operations:

- Determination of input and output formats when they are not explicitly identified in the relevant format descriptions in format files

- Location of format files when they are not specified on the command line

File Name Extensions

The expected extensions for data files are as follows:

.dat = ASCII, e.g., **latlon.dat**

.dab = dBASE, e.g., **latlon.dab**

.bin = binary or anything that is not **.dat** or **.dab**, e.g., **latlon.bin**

The expected extension for format description files is **.fmt**, e.g., **latlon.fmt**. You should not use mixed case extensions for format description files if you want to take advantage of FreeForm ND's default search capabilities. If you explicitly specify the names of format description files on the command line, you can use mixed case extensions.

> *Note:* Previous versions of FreeForm ND used variable description files (formerly called format specification files) each of which contained variable descriptions for one file. Expected extensions for these files were **.afm** (ASCII), **.bfm** (binary), and **.dfm** (dBASE). Variable descriptions for one or more files can now be incorporated into a single format description file. It is recommended that you convert and combine (as appropriate) existing variable description files into format description files.

File Name Relationships

FreeForm ND-based programs expect certain relationships between data file and format description file names as outlined below.

- The data file is named **datafile.ext** where **datafile** is the file name of your choosing and **ext** is the extension.
  Example: **latlon.dat**
- The corresponding format description file should be named **datafile.fmt**.
  Example: **latlon.fmt**
- If one format description file is used for multiple data files, all with the same extension, the format description file should be named **ext.fmt**.
  Example: **ll.fmt** is the format description file for **lldat1.ll**, **lldat2.ll**, and **lldat3.ll**.

Again, although not required, it is to your advantage to use these conventions.

Determining Input and Output Formats

You can optionally include read/write type (`input` or `output`) in format descriptors, e.g., `ASCII_input_data`. You may not want to specify the read/write type in some circumstances. For example, you may need to translate from native ASCII to binary, then back to ASCII. ASCII is the input format in the first translation and the output format in the second translation, vice versa for binary. You would need to edit the format description file before executing the second translation if you included read/write type in the format descriptors.

> *Note:* If you use the **-ft** option, you do not need to edit the format description file. See "Specifying Format Description Source" on page 52 later in this chapter.

If you do not specify read/write type, FreeForm ND can nevertheless determine which format in a format description file is input and which is output as long as you have adhered to FreeForm ND filenaming conventions.

- If the input format is not specified, and
  the input data filename extension is **.bin**, assume binary input.
  the input data filename extension is **.dab**, assume dBASE input.
  the input data filename extension is **.dat**, assume ASCII input.
  the input data filename extension is anything else, assume binary input.
- If the output format is not specified, and
  the input format is binary, the output is ASCII or dBASE, whichever is found first.
  the input format is dBASE, the output is ASCII or binary, whichever is found first.
  the input format is ASCII, the output is binary or dBASE, whichever is found first.

Note: The appropriate format descriptions must be in the format description file(s) used by FreeForm ND for a translation. If, for example, FreeForm ND determines the input format is binary and the output format is ASCII, there must be a format description for each type.

The **checkvar** program needs only an input format.

Locating Format Files

FreeForm ND programs use the following search sequence to find a format file (format or variable description file) for the data file **datafile.ext** when the format file name is not explicitly specified on the command line. In summary, FreeForm ND searches the directory specified by the GeoVu keyword **format_dir** (defined in a equivalence table or in the environment), the current or working directory, and the data file's home directory. The rules are applied in the order given below until a format file is found or all rules have been exhausted. If the relevant format file does not follow FreeForm ND conventions for name or location, it should be explicitly specified on the command line.

> *Note:* GeoVu is a FreeForm ND-based application for data access and visualization. FreeForm ND applications other than GeoVu use GeoVu keywords.

For information about equivalence tables, see the *GeoVu Tools Reference Guide*.

Search Sequence

1. Search the directory given by the GeoVu keyword **format_dir** for a format description file named **datafile.fmt**.
2. Search the directory given by the GeoVu keyword **format_dir** for variable description files named **datafile.afm**, **datafile.bfm**, and **datafile.dfm**.

   > *Note:* Step 2 is included to accommodate variable description files that were created using previous versions of FreeForm ND. It is recommended that you convert existing variable description files to format description files.

3. Search the directory given by the GeoVu keyword **format_dir** for a format description file named **ext.fmt**.

If the GeoVu keyword **format_dir** is not found, FreeForm ND continues the search for a format file as follows.

4. Search the current (default) directory for a format description file named **datafile.fmt**.
5. Search the current directory for variable description files named **datafile.afm**, **datafile.bfm**, and **datafile.dfm**. Use the criteria in step 2 for determining input and output format files.
6. Search the current directory for a format description file named **ext.fmt**.

If the data file's home directory is not the same as the current directory, FreeForm ND continues the search for a format file with steps 7-9. The data file's home directory is given by the directory path component of the data file name. If the data file name has no directory path component, the home directory search is not done.

7. Search the data file's home directory for a format description file named **datafile.fmt**.
8. Search the data file's home directory for variable description files named **datafile.afm**, **datafile.bfm**, and **datafile.dfm**. Use the criteria in step 2 for determining input and output format files.
9. Search the data file's home directory for a format description file named **ext.fmt**.

Case Sensitivity

FreeForm ND adheres to the following rules for case sensitivity (in applicable operating systems) when it searches for a format file for the data file **datafile.ext**.

- FreeForm ND preserves the case of **datafile**, for example, the default format file for the data file **LATLON.BIN** is **LATLON.fmt** (or **LATLON.bfm**).
- FreeForm ND searches for a format file with a lower case extension. That is, the format file must have its extension in lower case no matter what the case of **datafile**. For example, the default format file for the data file **LatLon.dat** is **LatLon.fmt** (or **LatLon.afm**), and **TIMEDATE.fmt** (or **TIMEDATE.bfm**) is the default format file for **TIMEDATE.bin**.
- In searching for a format description file of type **ext.fmt**, FreeForm ND preserves the case of **ext**. For example, for data files named **lldat1.LL**, **lldat2.LL**, and **latlon3.LL**, the default format description file is **LL.fmt**.

Command Line Arguments

FreeForm ND programs can take various command line arguments. The most widely used or standard arguments are discussed in this section. They are used for several different purposes: identifying input and output files, identifying format files and titles, changing run-time operation parameters, and defining data filters.

The only required argument for any FreeForm ND program is the name of the input file or file to be processed. All other arguments are optional and can be in any order following the input file name. The command line of a FreeForm ND program with the standard arguments has the following form:

application_name input_file [-f format_file] [-if input_format_file] [-of output_format_file] [-ft "title"] [-ift "title"] [-oft "title"] [-b local_buffer_size] [-c +/-count] [-v var_file] [-q query_file] [-o output_file]

> *Note:* To see a summary of command line usage for a FreeForm ND program, enter the program's name on the command line without any arguments.

Specifying Input and Output Files

input_file

Name of the file to be processed. Following FreeForm ND naming conventions, the standard extensions for data files are **.dat** for ASCII format, **.bin** for binary, and **.dab** for dBASE.

-o output_file

Option flag followed by the name of the output file. The standard extensions are the same as for input files.

Specifying Format Description Source

FreeForm ND offers a number of command line options for specifying the source of the format descriptions that a program must find in order to process data. The proper option or combination of options to use depends on how you have constructed your format files.

-f format_file

Option flag followed by the name of the format description file describing both input and output data.

-if input_format_file

Option flag followed by the name of the format description file describing the input data. Also use this option for an input variable description file written using earlier versions of FreeForm ND.

-of output_format_file

Option flag followed by the name of the format description file describing the output data. Also use this option for an output variable description file written using earlier versions of FreeForm ND.

-ft "title"

Option flag followed by the title (enclosed in quotes) of the format to be used for both input and output data, in which case there is no reformatting. The title follows format type on the first line of a format description in a format description file.

-ift "title"

Option flag followed by the title (enclosed in quotes) of the desired input format.

-oft "title"

Option flag followed by the title (enclosed in quotes) of the desired output format.

> *Note:* Previous versions of FreeForm ND used variable description files (**.afm**, **.bfm**, **.dfm**). It is recommended that you convert and combine (as appropriate) existing variable description files into format description files.

The various options available for specifying the source of a format description offer you a great deal of flexibility–in naming files, setting up format description files, and on the command line. In using these options, you need to consider the content of your format description files and how FreeForm ND will interpret the arguments on the command line.

Changing Run-time Parameters

FreeForm ND includes three arguments that let you change run-time parameters according to your needs. One argument lets you specify local buffer size, another indicates the number of records to process, and the third indicates which variables to process.

-b local_buffer_size

Option flag followed by the size of the memory buffer used to process the data and format files.

Default buffer size is 32,768. You many want to decrease the buffer size if you are running with low memory. Keep in mind that too small a buffer may result in unexpected behavior.

-c count

Option flag followed by a number that specifies how many data records at the head or tail of the file to process.

If **count** > 0, **count** records at the beginning of the file are processed. If **count** < 0, **count** records at the tail or end of the file are processed.

-v var_file

Option flag followed by the name of a variable file. The file contains names of the variables in the input data file to be processed by the FreeForm ND program. Variable names in **var_file** can be separated by one or more spaces or each name can be on a separate line.

Defining Filters

The query option lets you define data filters via a query file so you can precisely specify which data to process. The FreeForm ND program will process only those records meeting the query criteria.

-q query_file

Option flag followed by the name of the file containing query criteria. See “Query Syntax” on page 135 for query syntax.

CHAPTER 6

Format Conversion

The FreeForm ND utility program **newform** lets you convert data from one format to another. This allows you to pass data to applications in the format they require. You may also want to create binary archives for efficient data storage and access. With **newform**, conversion of ASCII data to binary format is straightforward. If you wish to read the data in a binary file, you can convert it to ASCII with **newform**, or use the interactive program **readfile**. You can also convert data from one ASCII format to another ASCII format with **newform**.

newform

The FreeForm ND-based program **newform** is a general tool for changing the format of a data file. The only required command line argument, if you use FreeForm ND naming conventions, is the name of the input data file. The reformatted data is written to standard output (the screen) unless you specify an output file. If you reformat to binary, you will generally want to store the output in a file.

You must create a format description file (or files) with format descriptions for the data files involved in a conversion before you can use **newform** to perform the conversion. The standard extension for format description files is **.fmt**. If you do not explicitly specify the format description file on the command line, which is unnecessary if you use FreeForm ND naming conventions, **newform** follows the FreeForm ND search sequence to find a format file.

For details about FreeForm ND naming conventions and the search sequence, see “FreeForm ND Conventions” on page 45.

The **newform** command has the following form:

> **newform input_file [-f format_file] [-if input_format_file] [-of output_format_file] [-ft "title"] [-ift "title"] [-oft "title"] [-b local_buffer_size] [-c +/-count] [-v var_file] [-q query_file] [-o output_file]**

For descriptions of the arguments, see “Command Line Arguments” on page 51.

If you want to convert an ASCII file to a binary file, and you follow the FreeForm ND naming conventions, the command is simply:

> **newform datafile.dat -o datafile.bin**

where **datafile** is the file name of your choosing.

> *Note:* If data files and format files are not in the current directory or in the same directory, you can specify the appropriate path name. For example, if the input data file is not in the current directory, you can enter:
> **newform /path/datafile.dat -o datafile.bin**

To read the data in the resulting binary file, you can reformat back to ASCII using the command:

> **newform datafile.bin -o datafile.ext**

or you can use the **readfile** program.

readfile

FreeForm ND includes **readfile**, a simple interactive binary file reader. The program has one required command line argument, the name of the file to be read. You do not have to write format descriptions to use **readfile**.

The **readfile** command has the following form:

readfile binary_data_file

When the program starts, it shows the available options:

```
Options:
          c:   char  1 byte character
          s:   short 2 byte signed integer
          l:   long  4 byte signed integer
          f:   float 4 byte single-precision floating
point
          d:   double8 byte double-precision floating
point
         uc:   uchar 1 byte unsigned integer
         us:   ushort2 byte unsigned integer
         ul:   ulong 4 byte unsigned integer

          b:   Toggle between "big-endian" and your ma-
chine’s native byte order
          p:   Set new file position
          P:   Show present file position and length
          h:   Display this help screen
          q:   Quit

Type option codes to view binary encoded values.
Tip: Pressing return repeats the last option.
```

The options let you interactively read your way through the specified binary file. The first position in the file is 0. You must type the character(s) indicating variable type (e.g., **us** for unsigned short) to view each value, so you need to know the data types of variables in the file and the order in which they occur. If successive variables are of the same type, you can press Return to view each value after the first of that type.

You can toggle the byte-order switch on and off by typing **b**. The byte-order option is used to read a binary data file that requires byte swapping. This is the case when you need cross-platform access to a file that is not byte-swapped, for example, if you are on a Unix machine reading data from a CD-ROM formatted for a PC. When the switch is on, type **s** or **l** to swap short or long integers respectively, or type **f** or **d** to swap floats or doubles. The **readfile** program does not byte swap the file itself (the file is unchanged) but byte swaps the data values internally for display purposes only.

To go to another position in the file, type **p**. You are prompted to enter the new file position in bytes. If, for example, each value in the file is 4 bytes long and you type **16**, you will be positioned at the first byte of the fifth value. If you split fields (by not repositioning at the beginning of a field), the results will probably be garbage. Type **P** to find out your current position in the file and total file length in bytes. Type **q** to exit from **readfile**.

You can also use an input command file rather than entering commands directly. In that case, the **readfile** command has the following form:

readfile binary_data_file < input_command_file

Creating a Binary Archive

By storing data files in binary, you save disk space and make access by applications more efficient. An ASCII data file can take two to five times the disk space of a comparable binary data file. Not only is there less information in each byte, but extra bytes are needed for decimal points, delimiters, and end-of-line markers.

It is very easy to create a binary archive using **newform** as the following examples show. The input data for these examples are in the ASCII file **latlon.dat** (shown below). They consist of 20 random latitude and longitude values. The size of the file on a Unix system is 460 bytes.

latlon.dat

```
-47.303545 -176.161101
 -0.928001    0.777265
-28.286662   35.591879
 12.588231  149.408117
-83.223548   55.319598
 54.118314 -136.940570
 38.818812   91.411330
-34.577065   30.172129
 27.331551 -155.233735
 11.624981 -113.660611
 77.652742  -79.177679
 77.883119  -77.505502
-65.864879  -55.441896
-63.211962  134.124014
 35.130219 -153.543091
 29.918847  144.804390
-69.273601   38.875778
-63.002874   36.356024
 35.086084  -21.643402
-12.966961   62.152266
```

Simple ASCII to Binary Conversion

In this example, you will use **newform** to convert the ASCII data file **latlon.dat** into the binary file **latlon.bin**. The input and output data formats are described in **latlon.fmt**.

latlon.fmt

```
/ This is the format description file for data files lat-
lon.bin
/ and latlon.dat. Each record in both files contains two
fields,
/ latitude and longitude.

binary_data "binary format"
latitude 1 8 double 6
longitude 9 16 double 6

ASCII_data "ASCII format"
latitude 1 10 double 6
longitude 12 22 double 6
```

The binary and ASCII variables both have the same names. The binary variable `latitude` occupies positions 1 to 8 and `longitude` occupies positions 9-16. The corresponding ASCII variables occupy positions 1-10 and 12-22. Both the binary and ASCII variables are stored as doubles and have a precision of 6.

Converting to Binary

To convert from an ASCII representation of the numbers in **latlon.dat** to a binary representation:

1. Change to the directory that contains the FreeForm ND example files.
2. Enter the following command:

   newform latlon.dat -o latlon.bin

Because FreeForm ND filenaming conventions have been used, **newform** will locate and use **latlon.fmt** for the translation. The **newform** program creates a new data file (effectively a binary archive) called **latlon.bin**. The size of the archive file is 2/3 the size of **latlon.dat**. Additionally, the data do not have to be converted to machine-readable representation by applications.

There are two methods for checking the data in **latlon.bin** to make sure they converted correctly. You can reformat back to ASCII and view the resulting file, or use **readfile** to read **latlon.bin**.

Reconverting to Native Format

Use the following **newform** command to reformat the binary data in **latlon.bin** to its native ASCII format:

newform latlon.bin -o latlon.rf

The ASCII file **latlon.rf** matches (but does not overwrite) the original input file **latlon.dat**. You can confirm this by using a file comparison utility. The **diff** command is generally available on Unix platforms.

To use **diff to compare the latlon ASCII files, enter the command:**

diff latlon.dat latlon.rf

The output should be something along these lines:

```
Files are effectively identical.
```

Note: The **diff** utility may detect a difference in other similar cases because FreeForm ND adds a leading zero in front of a decimal and interprets a blank as a zero if the field is described as a number. (A blank described as a character is interpreted as a blank.)

Reading the Binary File

To use **readfile** to read the data in **latlon.bin:**

1. Enter the following command:

   readfile latlon.bin

2. The data are stored as doubles, so enter **d** to view each value (or press Return to view each value after the first).
3. Enter **q** to quit **readfile**.

Conversion to a More Portable Binary

In this example, you will use **newform** to reformat the latitude and longitude values in the ASCII data file **latlon.dat** into binary longs in the binary file **latlon2.bin**. The input and output data formats are described in **latlon2.fmt**.

latlon2.fmt

```
/ This is the format description file for data files lat-
lon.dat
/ and latlon2.bin. Each record in both files contains two
fields,
/ latitude and longitude.

ASCII_data "ASCII format"
latitude 1 10 double 6
longitude 12 22 double 6

binary_data "binary format"
latitude 1 4 long 6
longitude 5 8 long 6
```

The ASCII and binary variables both have the same names. The ASCII variable `latitude` occupies positions 1-10 and `longitude` occupies positions 12-22. The ASCII variables are defined to be of type `double`. The binary variables occupy four bytes each (positions 1-4 and 5-8) and are of type `long`. The precision for all is 6.

Converting to Binary Long

In the previous example, both the ASCII and binary variables were defined to be doubles. Binary longs, which are 4-byte integers, may be more portable across different platforms than binary doubles or floats.

To **convert the ASCII data in latlon.dat to binary longs:**

1. Change to the directory that contains the FreeForm ND example files.
2. Enter the following command:

newform latlon.dat -f latlon2.fmt -o latlon2.bin

It creates the binary archive file **latlon2.bin** with the 20 latitude and longitude values in **latlon.dat** stored as binary longs.

This example duplicates one in chapter 2. If you completed that example, an error message will indicate that **latlon2.bin** exists. You can rename, move, or delete the existing file.

The size of the archive file **latlon2.bin** is about 1/3 the size of **latlon.dat**. Also, the data do not have to be converted to machine representation by applications. The main tradeoff in achieving savings in space and access time is that although binary longs are more portable than binary doubles or floats, any binary representation is less portable than ASCII.

Note: There may be a loss of precision when input data of type **double** is converted to **long**.

Reading the Binary File

Once again, you can use **readfile** to check the data in the binary archive you created.

1. Enter the following command:

   readfile latlon2.bin

2. The data are stored as longs, so enter **l** to view each value (or press Return to view each value after the first).
3. Enter **q** to quit **readfile**.

If desired, you can enter the commands to **readfile** from an input command file rather than directly from the command line. The example command file **latlon.in** is shown next.

latlon.in

```
llllllp0 llPq
```

The 6 `l`'s (`l` for `long`) cause the first 6 values in the file to be displayed. The sequence `p0` causes a return to the top (position 0) of the file. A position number (`0`) must be followed by a blank space. The 2 `l`'s display the first two values again. The `P` displays the current file position and length, and `q` closes **readfile**.

If you enter the following command:

readfile latlon2.bin < latlon.in

you should see the following output on the screen:

```
long:  -47303545
long: -176161101
long:    -928001
long:     777265
long:  -28286662
long:   35591879
New File Position = 0
long:  -47303545
long: -176161101
File Position: 8       File Length: 160
```

The floating point numbers have been multiplied by 106, the precision of the long variables in **latlon2.fmt**.

Including a Query

You can use the query option (**-q query_file**) to specify exactly which records in the data file **newform** should process. The query file contains query criteria. Query syntax is summarized in Appendix C.

In this example, you will specify a query so that **newform** will reformat only those value pairs in **latlon.dat** where latitude is positive and longitude is negative into the binary file **llposneg.bin**. The input and output data formats are described in **latlon2.fmt**.

The query criteria are specified in the following file.

llposneg.qry

```
[latitude] > 0 & [longitude] < 0
```

To convert the desired data in **latlon.dat** to binary and then view the results:

1. Enter the following command:

   newform latlon.dat -f latlon2.fmt -q llposneg.qry -o llposneg.bin

   The **llposneg.bin** file now contains the positive/negative latitude/longitude pairs in binary form.

2. To view the data, first convert the data in **llposneg.bin** back to ASCII format:

   newform llposneg.bin -f latlon2.fmt -o llposneg.dat

3. Enter the appropriate command to display the data in **llposneg.dat**, e.g.**more**:

   The following output appears on the screen:

```
54.118314 -136.940570
27.331551 -155.233735
11.624981 -113.660611
77.652742  -79.177679
77.883119  -77.505502
35.130219 -153.543091
35.086084  -21.643402
```

   Note: As demonstrated in the examples above, you can check the data in a binary file either by using **readfile** or by converting the data back to ASCII using **newform** and then viewing it.

File Names and Context

In the preceding examples, the read/write type (`input` or `output`) was not included in the format descriptors (`ASCII_data` and `binary_data`). FreeForm ND naming conventions were used, so **newform** can determine from the context which format should be used for input and which for output. Consider the command:

newform latlon.dat -o latlon.bin

The input file extension is **.dat** and the output file extension is **.bin**. These extensions provide context indicating that ASCII should be used as the input format and binary should be used as the output format. The format description file that **newform** will look for is the file with the same name as the input file and the extension **.fmt**, i.e., **latlon.fmt**.

If you use the following command:

newform latlon.bin

to translate the binary archive **latlon.bin** back to ASCII, **newform** identifies the input format as binary and uses the ASCII format for output. The ASCII data is written to the screen because an output file was not specified.

For information about FreeForm ND file name conventions, see "FreeForm ND Conventions" on page 45.

"Nonstandard" Data File Names

If you are working with data files that do not use FreeForm ND naming conventions, you need to more explicitly define the context. For example, the files **lldat1.ll**, **lldat2.ll**, **lldat3.ll**, **lldat4.ll**, and **lldat5.ll** all have latitude and longitude values in the ASCII format given in the format description file **lldat.fmt**. If you wanted to archive these files in binary format, you could *not* use a command of the form used in the previous examples, i.e., **newform datafile.dat -o datafile.bin** with **datafile.fmt** as the default format description file.

First, the ASCII data files do not have the extension **.dat**, which identifies them as ASCII files. Second, you would need five separate format description files, all with the same content: **lldat1.fmt**, **lldat2.fmt**, **lldat3.fmt**, **lldat4.fmt**, and **lldat5.fmt**. Creating the format description file **ll.fmt** solves both problems.

ll.fmt

```
/ This is the format description file that describes lat-
lon
/ data in files with the extension .ll

ASCII_input_data "ASCII format for .ll latlon data"
latitude 1 10 double 6
longitude 12 22 double 6

binary_output_data "binary format for .ll latlon data"
latitude 1 4 long 6
longitude 5 8 long 6
```

The name used for the format description file, **ll.fmt**, follows the FreeForm ND convention that one format description file can be utilized for multiple data files, all with the same extension, if the format description file is named **ext.fmt**. Also, the read/write type (`input` or `output`) is made explicit by including it in the format descriptors `ASCII_input_data` and `binary_output_data`. This provides the context needed for FreeForm ND programs to determine which format to use for input and which for output.

Use the following commands to produce binary versions of the ASCII input files:

newform lldat1.ll -o llbin1.ll

newform lldat2.ll -o llbin2.ll

newform lldat3.ll -o llbin3.ll

newform lldat4.ll -o llbin4.ll

newform lldat5.ll -o llbin5.ll

If you want to convert back to ASCII, you can switch the words `input` and `output` in the format description file **ll.fmt**. You could then use the following commands to convert back to native ASCII format with output written to the screen:

newform llbin1.ll

newform llbin2.ll

newform llbin3.ll

newform llbin4.ll

newform llbin5.ll

It is also possible to convert back to ASCII without switching the read/write types `input` and `output` in **ll.fmt**. You can specify input and output formats by title instead. In this case, you want to use the output format in **ll.fmt** as the input format and the input format in **ll.fmt** as the output format. Use the following command to convert **llbin1.ll** back to ASCII:

**newform llbin1.ll -ift "binary format for .ll latlon data"
 -oft "ASCII format for .ll latlon data"**

Notice that **newform** reports back the read/write type actually used. Since `ASCII_input_data` was used as the output format, **newform** reports it as `ASCII_output_data`.

Now assume that you want to convert the ASCII data file **llvals.asc** (not included in the example file set) to the binary file **latlon3.bin**, and the input and output data formats are described in **latlon.fmt**. The data file names do not provide the context allowing **newform** to find **latlon.fmt** by default, so you must include all file names on the command line:

newform llvals.asc -f latlon.fmt -o latlon3.bin

"Nonstandard" Format Description File Names

If you are using a format description file that does not follow FreeForm ND file naming conventions, you must include its name on the command line. Assume that you want to convert the ASCII data file **latlon.dat** to the binary file **latlon.bin**, and the input and output data formats are both described in **llvals.frm** (not included in the example file set). The data file names follow FreeForm ND conventions, but the name of the format description file does not, so it will not be located through the default search sequence. Use the following command to convert to binary:

newform latlon.dat -f llvals.frm -o latlon.bin

Suppose now that the input format is described in **latlon.fmt** and the output format in **llvals.frm**. You do not need to explicitly specify the input format description file because it will be located by default, but you must specify the output format description file name. In this case, the command would be:

newform latlon.dat -of llvals.frm -o latlon.bin

You can always unambiguously specify the names of format description files and data files, whether or not their names follow FreeForm ND conventions. Assume you want to look only at longitude values in **latlon.bin** and that you want them defined as integers (longs) which are right-justified at column 30. You will reformat the specified binary data in **latlon.bin** into ASCII data in **longonly.dat** and then view it. The input format is found in **latlon.fmt**, the output format in **longonly.fmt**.

longonly.fmt

```
/ This is the format description file for viewing longi-
tude as an
/ integer value right-justified at column 30.

ASCII_data "ASCII output format, right-justified at 30"
longitude 20 30 long 6
```

In this case, you have decided to look at the first 5 longitude values. Use the following command to unambiguously designate all files involved:

newform latlon.bin -if latlon.fmt -of longonly.fmt -c 5
-o longonly.dat

When you view **longonly.dat**, you should see the following 5 values:

```
         1         2         3         4
1234567890123456789012345678901234567890
```

```
-176161101
    777265
  35591879
 149408117
  55319598
```

Changing ASCII Formats

You may encounter situations where a specific ASCII format is required, and your data cannot be used in its native ASCII format. With **newform**, you can easily reformat one ASCII format to another. In this example, you will reformat California earthquake data from one ASCII format to three other ASCII formats commonly used for such data.The file **calif.tap** contains data about earthquakes in California with magnitudes > 5.0 since 1980. The data were initially distributed by NGDC on tape, hence the **.tap** extension. The data format is described in **eqtape.fmt**:

eqtape.fmt

```
/ This is the format description file for the NGDC .tap
format,
/ which is used for data distributed on floppy disks or
tapes.

ASCII_data ".tap format"
source_code 1 3 char 0
century 4 6 short 0
year 7 8 short 0
month 9 10 short 0
day 11 12 short 0
hour 13 14 short 0
minute 15 16 short 0
second 17 19 short 1
latitude_abs 20 24 long 3
latitude_ns 25 25 char 0
longitude_abs 26 31 long 3
longitude_ew 32 32 char 0
```

```
depth 33 35 short 0
magnitude_mb 36 38 short 2
MB 39 40 constant 0
isoseismal 41 43 char 0
intensity 44 44 char 0


/ The NGDC record check format includes
/ six flags in characters 45 to 50. These
/ can be treated as one variable to allow
/ multiple flags to be set in a single pass,
/ or each can be set by itself.

ngdc_flags 45 50 char 0
diastrophic 45 45 char 0
tsunami 46 46 char 0
seiche 47 47 char 0
volcanism 48 48 char 0
non_tectonic 49 49 char 0
infrasonic 50 50 char 0

fe_region 51 53 short 0
magnitude_ms 54 55 short 1
MS 56 57 char 0
z_h 58 58 char 0
cultural 59 59 char 0
other 60 60 char 0
magnitude_other 61 63 short 2
other_authority 64 66 char 0
ide 67 67 char 0
depth_control 68 68 char 0
number_stations_qual 69 71 char 0
time_authority 72 72 char 0
magnitude_local 73 75 short 2
local_scale 76 77 char 0
local_authority 78 80 char 0
```

Three other formats used for California earthquake data are hypoellipse, hypoinverse, and hypo71. Subsets of these formats are described in the format description file **hypo.fmt**. The format descriptions include the parameters required by the AcroSpin program that is distributed as part of the IASPEI Software Library (Volume 2). AcroSpin shows 3D views of earthquake point data.

hypo.fmt

```
/ This format description file describes subsets of the
/ hypoellipse, hypoinverse, and hypo71 formats.

ASCII_data "hypoellipse format"
year 1 2 uchar 0
month 3 4 uchar 0
day 5 6 uchar 0
hour 7 8 uchar 0
minute 9 10 uchar 0
second 11 14 ushort 2
latitude_deg_abs 15 16 uchar 0
latitude_ns 17 17 char 0
latitude_min 18 21 ushort 2
longitude_deg_abs 22 24 uchar 0
longitude_ew 25 25 char 0
longitude_min 26 29 ushort 2
depth 30 34 short 2
magnitude_local 35 36 uchar 1

ASCII_data "hypoinverse format"
year 1 2 uchar 0
month 3 4 uchar 0
day 5 6 uchar 0
hour 7 8 uchar 0
minute 9 10 uchar 0
second 11 14 ushort 2
latitude_deg_abs 15 16 uchar 0
latitude_ns 17 17 char 0
latitude_min 18 21 ushort 2
longitude_deg_abs 22 24 uchar 0
longitude_ew 25 25 char 0
longitude_min 26 29 ushort 2
depth 30 34 short 2
magnitude_local 35 36 uchar 1
number_of_times 37 39 short 0
maximum_azimuthal_gap 40 42 short 0
nearest_station 43 45 short 1
```

```
rms_travel_time_residual 46 49 short 2

ASCII_data "hypo71 format"
year 1 2 uchar 0
month 3 4 uchar 0
day 5 6 uchar 0
hour 8 9 uchar 0
minute 10 11 uchar 0
second 12 17 float 2
latitude_deg_abs 18 20 uchar 0
latitude_ns 21 21 char 0
latitude_min 22 26 float 2
longitude_deg_abs 27 30 uchar 0
longitude_ew 31 31 char 0
longitude_min 32 36 float 2
depth 37 43 float 2
magnitude_local 44 50 float 2
number_of_times 51 53 short 0
maximum_azimuthal_gap 54 57 float 0
nearest_station 58 62 short 1
rms_travel_time_residual 63 67 float 2
error_horizontal 68 72 float 1
error_vertical 73 77 float 1
s_waves_used 79 79 char 0
```

The parameters from the California earthquake data in the NGDC format needed for use with the AcroSpin program can be extracted and converted using the following commands:

newform calif.tap -if eqtape.fmt -of hypo.fmt -oft "hypoellipse format" -o calif.he

newform calif.tap -if eqtape.fmt -of hypo.fmt -oft "hypoinverse format" -o calif.hi

newform calif.tap -if eqtape.fmt -of hypo.fmt -oft "hypo71 format" -o calif.h71

If you develop an application that accesses seismicity data in a particular ASCII format, you need only to write an appropriate format description file in order to convert NGDC data into the format used by the application. This lets you make use of the data that NGDC provides in a format that works for you.

CHAPTER 7

Conversion Variables

Space and time values such as latitude and longitude, date, and time of day can be represented in various ways. For example, latitude and longitude can be given in degrees and minutes, or as floating point numbers (among other possibilities). FreeForm ND conversion variables make it possible to translate between a number of representations of space and time values. You tell FreeForm ND that a conversion is needed by including the appropriate standard conversion variable name in the relevant format description file. When FreeForm ND reads a format description file and finds a conversion variable, it automatically performs the indicated conversion.

Accessing Conversion Functions

FreeForm ND's conversion functions are invoked by using standard conversion variable names in the input and output format descriptions. FreeForm ND attempts a conversion only if the input and output names for a variable differ, and both names are included in FreeForm ND's list of standard conversion variables (see "Conversion Variable Names" on page 123). If a variable name in an output format does not correspond to a name in the input format, FreeForm ND searches the input variables for standard conversion variable names.

For example, assume the following variable is described in the input format description:

```
latitude 1 10 double 6
```

The output format description includes the following variable descriptions, but not one for `latitude`:

```
latitude_deg 1 7 short 0
latitude_min 13 15 short 0
latitude_sec 21 23 short 0
```

FreeForm ND will transparently identify and call conversion functions to construct the specified output values (latitude in units of degrees, minutes, and seconds) using the input value given by the variable `latitude`.

Latitude and Longitude Conversions

Space is often delineated by latitude and longitude in geophysical applications. Latitude and longitude values can be represented most directly as floating point numbers, but often are not. Data sets frequently give latitude and longitude in other representations such as degrees and minutes, or absolute value of degrees, decimal minutes, and N/S or E/W to designate hemisphere.

FreeForm ND includes a set of functions that perform conversions between a number of the most common representations of latitude and longitude. In order to access these conversions, you must use the following standard variable names.

Name	Description	Example Value
latitude longitude	Signed floating point number that completely describes a latitude or longitude coordinate value	-47.583333 -176.75
latitude_abs longitude_abs	Absolute value of a latitude or longitude coordinate (may include fractions of a degree)	47.583333 176.75
latitude_deg longitude_deg	Degrees component of a latitude or longitude coordinate value (may be signed)	-47 -176
latitude_deg_abs longitude_deg_abs	Absolute value of the degrees component of a latitude or longitude coordinate	47 176
latitude_min longitude_min	Minutes component of a latitude or longitude coordinate value	30.5 45.0
latitude_sec longitude_sec	Seconds component of a latitude or longitude coordinate value	30.0 0.0
latitude_sign longitude_sign	Sign of a latitude or longitude coordinate value + or - (data type is **char**)	-
latitude_ns longitude_ew	Hemisphere: N for north, S for south, E for east, W for west (**char**)	S
geog_quad_code	A geographic quadrant defined by DMA (Defense Mapping Agency), 1 = NE, 2 = NW, 3 = SE, 4 = SW (**char**)	4

FreeForm ND uses the convention that northern latitudes and eastern longitudes are positive.

Degrees, Minutes, and Seconds

In this example you will convert latitude and longitude values in **latlon2.bin** from long integers (with implied precision) to latitude and longitude values given in degrees, minutes and seconds. The binary file **laton2.bin** was created earlier from the ASCII file **latlon.dat** (see “Quick Tour of FreeForm ND” on page 13). The input and output formats are described in **ll_d_m_s.fmt**. Conversion variable names are included in the input and output formats.

ll_d_m_s.fmt

```
/ This is the format description file for the data files latlon2.bin and
/ ll_d_m_s.dat. Each record of the input binary file latlon2.bin con-
tains
/ two fields, latitude and longitude. These values are stored as inte-
gers.
/ Each record of the output ASCII file ll_d_m_s.dat contains latitude
and
/ longitude given in units of degrees, minutes, and seconds.

binary_data "binary input format"
latitude 1 4 long 6
longitude 5 8 long 6

ASCII_data "ASCII output format"
latitude_deg 1 7 short 0
latitude_min 13 15 short 0
latitude_sec 21 23 short 0
longitude_deg 27 31 short 0
longitude_min 37 39 short 0
longitude_sec 45 47 short 0
```

To convert the data to the new ASCII format use the following command:

newform latlon2.bin -f ll_d_m_s.fmt -o ll_ d_m_s.dat

The ASCII file **ll_d_m_s.dat** is created with the 20 latitude and longitude values given in degrees, minutes, and seconds. If a degree value is between 0 and -1, then either the minute or second value is signed. When you view **ll_d_m_s.dat**, you should see the following values:

```
         1         2         3         4         5
12345678901234567890123456789012345678901234567890
    -47       18        13    -176         9        40
      0      -55        41       0        46        38
    -28       17        12      35        35        31
     12       35        18     149        24        29
```

```
-83      13      25      55      19      11
 54       7       6    -136      56      26
 38      49       8      91      24      41
-34      34      37      30      10      20
 27      19      54    -155      14       1
               .
               .
               .
```

You can convert the data file **ll_d_m_s.dat** back to its original ASCII format (in **latlon.dat**) but the values will be somewhat different than those in **latlon.dat**. The ASCII format for **ll_d_m_s.dat** uses whole seconds, which are not precise enough to represent decimal degrees to six decimal places. Fractional seconds are required to preserve the values of decimal degrees to six places. If **ushort** variables with a precision of 3 were specified in **ll_m_d_s.fmt**, fractional seconds could be represented.

Absolute Degrees and Minutes

In the following two examples, you will create new ASCII data files from **latlon2.bin** that give latitude and longitude in absolute degrees and minutes with hemisphere indicated in the first case and geographic quadrant in the second case. Conversion variable names are used in the input and output formats in both examples.

With Hemisphere

You will convert the data in **latlon2.bin** to latitude and longitude values given in absolute degrees and minutes. FreeForm ND converts the sign (+ or -) of the input data to N for north, S for south, E for east, or W for west as appropriate. Southern latitudes and western longitudes are negative. The input and output formats are described in **degabsm.fmt**.

degabsm.fmt

```
/ This is the format description file for the data files latlon2.bin and
/ degabsm.dat. Each record of the input binary file latlon2.bin con-
tains
```

```
/ two fields, latitude and longitude. These values are stored as inte-
gers.

binary_data "binary input format"
latitude 1 4 long 6
longitude 5 8 long 6

/ Each record of the output ASCII file degabsm.dat contains latitude and
/ longitude given in units of absolute degrees and minutes. The hemi-
sphere
/ is indicated by the variables latitude_ns and longitude_ew. The value
can be
/ the character N for north, S for south, E for east, or W for west.

ASCII_data "ASCII output format"
latitude_deg_abs 6 7 short 0
latitude_min_abs 10 15 float 2
latitude_ns 17 17 char 0
longitude_deg_abs 24 26 short 0
longitude_min_abs 28 34 float 3
longitude_ew 36 36 char 0
```

To convert the data to absolute degrees and minutes with hemisphere included, use the following command:

newform latlon2.bin -f degabsm.fmt -o degabsm.dat

When you view **degabsm.dat**, you should see the following values:

```
         1         2         3         4
1234567890123456789012345678901234567890
     47  18.21  S      176   9.666 W
      0  55.68  S        0  46.636 E
     28  17.20  N       35  35.513 E
     12  35.29  N      149  24.487 E
     83  13.41  S       55  19.176 E
               .
               .
               .
```

With Quadrant

You will convert the data in **latlon2.bin** to latitude and longitude values given in absolute degrees and minutes with the geographic quadrant indicated by a character code. The input and output formats are described in **degmgeog.fmt**.

degmgeog.fmt

```
/ This is the format description file for the data files latlon2.bin and
/ degmgeog.dat. Each record of the input binary file latlon2.bin con-
tains
/ two fields, latitude and longitude. These values are stored as inte-
gers.

binary_data "binary input format"
latitude 1 4 long 6
longitude 5 8 long 6

/ Each record of the output ASCII file degmgeog.dat contains latitude
and
/ longitude given in units of absolute degrees and minutes. The
/ geographic quadrant of the data is indicated by a numeric character
code.
/
/ 1 = Northeast
/ 2 = Northwest
/ 3 = Southeast
/ 4 = Southwest
/

ASCII_data "ASCII output format"
latitude_deg_abs 6 7 short 0
latitude_min_abs 10 15 float 2
longitude_deg_abs 21 23 short 0
longitude_min_abs 26 31 float 2
geog_quad_code 40 40 char 0
```

To convert the data to absolute degrees and minutes with quadrant, use the following command:

newform latlon2.bin -f degmgeog.fmt -o degmgeog.dat

When you view **degmgeog.dat**, you should see the following values:

```
           1         2         3         4
1234567890123456789012345678901234567890123456789012345
      47   18.21     176    9.67         4
       0   55.68       0   46.64         3
      28   17.20      35   35.51         1
      12   35.29     149   24.49         1
```

```
    83   13.41      55   19.18        3
                   .
                   .
                   .
```

Date and Time Conversions

Time is a variable found in many scientific data sets and it can have various representations. In ASCII formats meant to be read by application users, time is often represented with six variables: year, month, day, hour, minute, second. In formats meant to be read by computers, it makes sense to represent time as a floating point number in days and decimal fractions of a day, or perhaps seconds and fractions of a second.

FreeForm ND can perform conversions between various representations of dates when standard conversion variable names are included in the format descriptions. Several examples are given below.

Year, Month, Day

In this example you will convert a date string in the form of month/day/year to a date string in the form of year, month, day with no separators. The format description file **yymmdd.fmt** describes the input and output formats and the input data is stored in **mdy.dat**. Notice that this is a conversion from one ASCII data format to another.

yymmdd.fmt

```
/ This is the format description file for the data files
mdy.dat and
/ yymmdd.dat.

ASCII_input_data "ASCII input format"
date_mm/dd/yy 1 10 char 0
```

```
ASCII_output_data "ASCII output format"
date_yymmdd 1 12 char 0
```

mdy.dat

```
   1/26/20
   7/25/78
  11/19/99
         .
         .
         .
```

To convert the data from m/d/y format to yymmdd format, use the following command:

newform mdy.dat -f yymmdd.fmt -o yymmdd.dat

The resulting file **yymmdd.dat** will contain the following values:

yymmdd.dat

```
    200126
    780725
    991119
       .
       .
       .
```

Serial Dates

If you have time data in an ASCII format and the data will be read primarily by an application, you may want to convert it to a binary format. FreeForm ND supports a binary representation of time as a serial day starting at January 1, 1980.

FreeForm ND conversion functions let you convert from an ASCII representation to the binary serial date representation. As an example, you will convert the ASCII data in **time.dat**, which contains 10 random times from this century, to a binary serial date format in **serial.bin**. The format description file **serial.fmt** describes the input and output formats for **time.dat** and **serial.bin**. It also contains a format description for **serial.dat**, which will contain the data in **serial.bin** in an ASCII format.

serial.fmt

```
/ This is the format description file for the data files time.dat, se-
rial.bin,
/ and serial.dat. Each record of the ASCII file time.dat contains six
/ fields: year, month, day, hour, minute, second.

ASCII_data "ASCII ymdhms date"
year 2 5 ushort 0
month 10 11 uchar 0
day 19 20 uchar 0
hour 28 29 uchar 0
minute 37 38 uchar 0
second 43 47 float 2

/ Each record of the binary file serial.bin contains one field,
/ serial date, defined as a double that occupies 8 bytes and has
/ 8 places of precision.

binary_data "binary serial date"
serial_day_1980 1 8 double 8
/ Each record of the ASCII file serial.dat contains one field,
/ serial date.

ASCII_data "ASCII serial date"
serial_day_1980 1 16 double 8
```

time.dat

```
 1920     1        26        11        26    49.79
 1978     7        25         1        36    14.89
 1999    11        19        14         4     4.78
                             .
                             .
                             .
```

To convert the dates from the ASCII format in **time.dat** to the binary serial date format, use the following command:

> **newform time.dat -f serial.fmt -ift "ASCII ymdhms date" -o serial.bin**

Then view the binary file **serial.bin** with either of the following commands:

newform serial.bin -oft "ASCII serial date" -o serial.dat

or

readfile serial.bin

You should see the following values:

```
-21889.52303484
  -524.93316100
  7262.58616644
-20525.28111250
  5046.80073889
      .
      .
      .
```

Calculated Input Variables

Calculated input variables are very similar to calculated output variables; both perform arithmetic on the data being processed. However, calculated input variables are processed immediately after data is read, and calculated output variables are processed after input data has been converted. Often this distinction is unimportant, and calculated output variables should be used whenever possible. Yet there are times you must use, or would prefer to use, calculated input variables. Here are two examples.

Deriving Record Count

A data file contains record headers, but the record headers contain the record number of the current and the next record header. FreeForm ND requires that the number of data records after each record header be defined (or derived). This format file shows how a calculated input variable is used to derive the number of data records that follow each record header.

```
binary_input_file_header "File Header"
stuff 1 24 text 0

/ equation to determine count:
/ next-head-r# - this-head-r# - (2160 / 24)
/ note: 2160 / 24 is # records occupied by record header
/ next-head-r# - this-head-r# - 90

input_eqv
begin constant
this_header_record_number_eqn text [next_header_record_number]-
[this_header_record_number]-90
end constant

begin name_equiv
count this_header_record_number
end name_equiv

binary_input_record_header "Record Header"
this_header_record_number_eqn 1    4 long 0
next_header_record_number     5    8 long 0
extra_stuff                   9 2160 text 0

binary_input_data "Inventory Points"
stuff 1 24 text 0

binary_output_data "Inventory Points"
stuff 1 24 text 0
```

In this example, the file header, record header, and data are read, but only the data is output. When calculating the number of data records we subtract the number of records in the record header (the record header is 2160 bytes = 90 records X 24 bytes/record). The **_eqn** suffix on the variable named **this_header_record_number** indicates that this is a calculated variable (just as with calculated output variables). The equation for this variable is defined in an input equivalence section. Furthermore, a name equivalence between **count** and **this_header_record_number** is required. Alternatively, we could rename **this_header_record_number_eqn** to **count_eqn**.

Note that the calculated value for **this_header_record_number** overwrites (replaces) the value read from the file for

this_header_record_number. In this case, we are not writing record headers so it doesn't matter. However, if we were writing record headers and wanted to preserve the original meaning of this variable, we could "undo" the calculation with a calculated output variable.

A Year 2000 Solution

A calculated input variable can be used to make a two-digit date field appear as a four-digit date field (or more digits, up to about 15, which is the maximum allowed by a double-precision float). Here is a format file from Year 2000 Compliant Data, rewritten to use calculated input variables:

```
input_eqv
begin constant
year_eqn char 1900 + [year]
end constant

ASCII_input_data        "Year without Century, but seems as if it did"
year_eqn           1  2   uint8 0
month              3  4   uint8 0
day                5  6   uint8 0
hour               8  9   uint8 0
minute            10 11   uint8 0
second            12 17 float32 2
latitude_deg_abs  18 20   uint8 0
latitude_ns       21 21    text 0
latitude_min      22 26 float32 2
longitude_deg_abs 27 30   uint8 0
longitude_ew      31 31    text 0
longitude_min     32 36 float32 2
depth             37 43 float32 2
magnitude_local   44 50 float32 2
spaces            51 79    text 0

ASCII_output_data       "no calculated output variables here"
year               1  4  uint16 0
month              5  6   uint8 0
day                7  8   uint8 0
hour              10 11   uint8 0
minute            12 13   uint8 0
second            14 19 float32 2
latitude_deg_abs  20 22   uint8 0
latitude_ns       23 23    text 0
```

```
latitude_min      24 28 float32 2
longitude_deg_abs 29 32   uint8 0
longitude_ew      33 33    text 0
longitude_min     34 38 float32 2
depth             39 45 float32 2
magnitude_local   46 52 float32 2
spaces            53 81    text 0
```

Almost immediately after the data is read the year field is transmogrified from an uint8 into a float64 field, and each year value has 1900 added to it. The difference between using calculated input variables and calculated output variables to solve the Year 2000 problem for data is like the difference between applications which read data and the data itself. In the above example, the data is not Y2K compliant, but thanks to the above format file, any FreeForm ND application will think it is. This avoids mass conversion of data (not a trivial task), but the data remains in noncompliance with respect to any software except FreeForm ND.

CHAPTER 8

Header Formats

Headers are one of the most commonly encountered forms of metadata–data about data. Applications need the information contained in headers for reading the data that the headers describe. To access these data, applications must be able to read the headers. Just as there are many data formats, there are numerous header formats. You can include header format descriptions, which have exactly the same form as data format descriptions, in format description files.

Header Treatment in FreeForm ND

FreeForm ND is not 100 percent backwards compatible with FreeForm ND in the area of header treatment.

Headers have traditionally been handled differently from data in FreeForm ND. If a header format was not specified as either input or output, it was taken as both input and output. **newform** did little in processing headers, and FreeForm ND relied on extraneous utilities to work with headers.

New Behavior

In FreeForm ND, header formats are treated the same as data formats. This means that header formats must be identified as either input or output, explicitly or implicitly. If done explicitly, then either the input or the output descriptor will form the format type (e.g., **ASCII_input_header**). If done implicitly, then the same ambiguity resolution rules that apply to data formats will be applied to header formats. This means that ASCII header formats will be taken as input for data files with a **.dat** extension, dBASE header formats will be taken as input for data files with a **.dab** extension, and binary header formats will be taken as input for all other data files.

If an embedded header and the data have different file types, then either the header format or data format (preferably both) must be explicitly identified as input or output (for example, an ASCII header embedded in a binary data file). Obviously, ambiguous formats with different file types cannot both be resolved as input formats.

The same header format is no longer used as both an input and an output header format.

In FreeForm ND, **newform** honors output header formats that are separate (e.g., **ASCII_output_header_separate**). The header is written to a separate file which, unless otherwise specified, is named after the output data file with a .hdr extension. This requires that you name the output file using the **-o** option flag; redirected output cannot be used with separate output headers. The output header file name and path can be specified using the same keywords that tell FreeForm ND how to find an input separate header file (i.e., **header_file_ext**, **header_file_name**, and **header_file_path**).

When defining keywords to specify how an output header file is to be named, you must use a new type of equivalence section, **input_eqv**, which must appear in the format file along with **output_eqv**.

Header Types

FreeForm ND recognizes two types of headers. File headers describe all the data in a file whereas record headers describe the data in a single record or data block. FreeForm ND can read headers included in the data file or stored in a separate file. Header formats, like data formats, are described in format description files. For a list of the header descriptors you can use in format descriptions, see “Format Descriptions for Tabular Data” on page 23.

File Headers

A file header included in a data file is at the beginning of the file. Only one file header can be associated with a data file. Alternatively, a file header can be stored in a file separate from the data file.

In the following example, a file header is used to store the minimum and maximum for each variable and the data are converted from ASCII to binary. There are two variables, `latitude` and `longitude`. The file header format and data formats are described in the format description file **llmaxmin.fmt**.

llmaxmin.fmt

```
ASCII_file_header "Latitude/Longitude Limits"
minmax_title 1 24 char 0
latitude_min 25 36 double 6
latitude_max 37 46 double 6
longitude_min 47 59 double 6
longitude_max 60 70 double 6

ASCII_data "lat/lon"
latitude 1 10 double 6
longitude 12 22 double 6

binary_data "lat/lon"
latitude 1 4 long 6
longitude 5 8 long 6
```

The example ASCII data file **llmaxmin.dat** contains a file header and data as described in **llmaxmin.fmt**.

llmaxmin.dat

```
       1       2       3       4       5       6       7
1234567890123456789012345678901234567890123456789012345
678901234567890

Latitude and Longitude:   -83.223548 54.118314  -
176.161101 149.408117
-47.303545 -176.161101
-25.928001    0.777265
-28.286662   35.591879
 12.588231  149.408117
-83.223548   55.319598
 54.118314 -136.940570
 38.818812   91.411330
-34.577065   30.172129
 27.331551 -155.233735
 11.624981 -113.660611
```

This use of a file header would be appropriate if you were interested in creating maps from large data files. By including maximums and minimums in a header, the scale of the axes can be determined without reading the entire file.

FreeForm ND naming conventions have been followed in this example, so to convert the ASCII data in the example to binary format, use the following simple command:

newform llmaxmin.dat -o llmaxmin.bin

The file header in the example will be written into the binary file as ASCII text because the header descriptor in **llmaxmin.fmt** (`ASCII_file_header`) does not specify read/write type, so the format is used for both the input and output header.

Record Headers

Record headers occur once for every block of data in a file. They are interspersed with the data, a configuration sometimes called a format sandwich. Record headers can also be stored together in a separate file.

The following format description file specifies a record header and ASCII and binary data formats for aeromagnetic trackline data.

aeromag.fmt

```
ASCII_record_header "Aeromagnetic Record Header Format"
flight_line_number 1 5 long 0
count 6 13 long 0
fiducial_number_corresponding_to_first_logical_record 14 22 long 0
date_MMDDYY_or_julian_day 23 30 long 0
flight_number 31 38 long 0
utm_easting_of_first_record 39 48 float 0
utm_northing_of_first_record 49 58 float 0
utm_easting_of_last_record 59 68 float 0
utm_northing_of_last_record 69 78 float 0
blank_padding 79 104 char 0

ASCII_data "Aeromagnetic ASCII Data Format"
flight_line_number 1 5 long 0
fiducial_number 6 15 long 0
utm_easting_meters 16 25 float 0
utm_northing_meters 26 35 float 0
mag_total_field_intensity_nT 36 45 long 0
mag_residual_field_nT 46 55 long 0
alt_radar_meters 56 65 long 0
alt_barometric_meters 66 75 long 0
blank 76 80 char 0
latitude 81 92 float 6
longitude 93 104 float 6

binary_data "Aeromagnetic Binary Data Format"
flight_line_number 1 4 long 0
fiducial_number 5 8 long 0
utm_easting_meters 9 12 long 0
utm_northing_meters 13 16 long 0
mag_total_field_intensity_nT 17 20 long 0
mag_residual_field_nT 21 24 long 0
alt_radar_meters 25 28 long 0
alt_barometric_meters 29 32 long 0
blank 33 37 char 0
```

```
latitude 38 41 long 6
longitude 42 45 long 6
```

The example ASCII file **aeromag.dat** contains two record headers followed by a number of data records. The header and data formats are described in **aeromag.fmt**. The variable `count` (second variable defined in the header format description) is used to indicate how many data records occur after each header.

aeromag.dat

```
       1         2         3         4         5         6         7         8         9        10
1234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789
012345
  420      5     5272     178       2   413669.  6669740.   333345.  6751355.
  420     5272   413669.  6669740.   2715963   2715449      1088      1348        60.157307 -154.555191
  420     5273   413635.  6669773.   2715977   2715464      1088      1350        60.157593 -154.555817
  420     5274   413601.  6669807.   2716024   2715511      1088      1353        60.157894 -154.556442
  420     5275   413567.  6669841.   2716116   2715603      1079      1355        60.158188 -154.557068
  420     5276   413533.  6669875.   2716263   2715750      1079      1358        60.158489 -154.557693
  411      10     8366     178       2   332640.  6749449.   412501.  6668591.
  411     8366   332640.  6749449.   2736555   2736538       963      1827       60.846806 -156.080185
  411     8367   332674.  6749415.   2736539   2736522       932      1827       60.846516 -156.079529
  411     8368   332708.  6749381.   2736527   2736510       917      1829       60.846222 -156.078873
  411     8369   332742.  6749347.   2736516   2736499       922      1832       60.845936 -156.078217
  411     8370   332776.  6749313.   2736508   2736491       946      1839       60.845642 -156.077560
  411     8371   332810.  6749279.   2736505   2736488       961      1846       60.845348 -156.076904
  411     8372   332844.  6749245.   2736493   2736476       982      1846       60.845062 -156.076248
  411     8373   332878.  6749211.   2736481   2736463      1015      1846        60.844769 -156.075607
  411     8374   332912.  6749177.   2736470   2736452      1029      1846        60.844479 -156.074951
  411     8375   332946.  6749143.   2736457   2736439      1041      1846        60.844189 -156.074295
```

This file contains two record headers. The first occurs on the first line of the file and has a `count` of 5, so it is followed by 5 data records. The second record header follows the first 5 data records. It has a `count` of 10 and is followed by 10 data records.

The FreeForm ND default naming conventions have been used here so you could use the following abbreviated command to reformat **aeromag.dat** to a binary file named **aeromag.bin**:

newform aeromag.dat -o aeromag.bin

The ASCII record headers are written into the binary file as ASCII text.

Separate Header Files

You may need to describe a data set with external headers. An external or separate header file can contain only headers–one file header or multiple record headers.

Separate File Header

Suppose you want the file header used to store the minimum and maximum values for latitude and longitude (from the **llmaxmin** example) in a separate file so that the data file is homogenous, thus easier for applications to read. Instead of one ASCII file (**llmaxmin.dat**), you will have an ASCII header file, say it is named **llmxmn.hdr**, and an ASCII data file–call it **llmxmn.dat**.

llmxmn.hdr

```
Latitude and Longitude:   -83.223548 54.118314  -
176.161101 149.408117
```

llmxmn.dat

```
-47.303545 -176.161101
-25.928001    0.777265
-28.286662   35.591879
 12.588231  149.408117
-83.223548   55.319598
 54.118314 -136.940570
 38.818812   91.411330
-34.577065   30.172129
 27.331551 -155.233735
 11.624981 -113.660611
```

You will need to make one change to **llmaxmin.fmt**, adding the qualifier `separate` to the header descriptor, so that FreeForm ND will look for the header in a separate file. The first line of **llmaxmin.fmt** becomes:

```
ASCII_file_header_separate "Latitude/Longitude Limits"
```

Save **llmaxmin.fmt** as **llmxmn.fmt** after you make the change.

To convert the data in **llmxmn.dat** to binary format in **llmxmn.bin**, use the following command:

newform llmxmn.dat -o llmxmn.bin

Note: When you run **newform**, it will write the separate header to **llmxmn.bin** along with the data in **llmxmn.dat**.

Separate Record Headers

Record headers in separate files can act as indexes into data files if the headers specify the positions of the data in the data file. For example, if you have a file containing data from 25 observation stations, you could effectively index the file by including a station ID and the starting position of the data for that station in each record header. Then you could use the index to quickly locate the data for a particular station.

Returning to the **aeromag** example, suppose you want to place the two record headers in a separate file. Again, the only change you need to make to the format description file (**aeromag.fmt**) is to add the qualifier `separate` to the header descriptor. The first line would then be:

```
ASCII_record_header_separate "Aeromagnetic Record Header Format"
```

The separate header file would contain the following two lines:

```
 420     5    5272    178      2  413669. 6669740.  333345. 6751355.
 411    10    8366    178      2  332640. 6749449.  412501. 6668591.
```

The data file would look like the current **aeromag.dat** with the first and seventh lines removed.

Assuming the data file is named **aeromag.dat**, the default name and location of the header file would be **aeromag.hdr** in the same directory as the data file. Otherwise, the separate header file name and location need to be defined in an equivalence table. (For information about equivalence tables, see the *GeoVu Tools Reference Guide*.)

The dBASEfile Format

Headers and data records in dBASE format are represented in ASCII but are not separated by end-of-line characters. They can be difficult to read or to use in applications that expect newlines to separate records. By using **newform**, dBASE data can be reformatted to include end-of-line characters.

In this example, you will reformat the dBASE data file **oceantmp.dab** (see below) into the ASCII data file **oceantmp.dat**. The input file **oceantmp.dab** contains a record header at the beginning of each line. The header is followed by data on the same line. When you convert the file to ASCII, the header will be on one line followed by the data on the number of lines specified by the variable `count`. The format description file **oceantmp.fmt** is used for this reformatting.

oceantmp.fmt

```
dbase_record_header "NODC-01 record header format"
WMO_quad 1 1 char 0
latitude_deg_abs 2 3 uchar 0
latitude_min 4 5 uchar 0
longitude_deg_abs 6 8 uchar 0
longitude_min 9 10 uchar 0
date_yymmdd 11 16 long 0
hours 17 19 uchar 1
country_code 20 21 char 0
vessel 22 23 char 0
count 24 26 short 0
data_type_code 27 27 char 0
cruise 28 32 long 0
station 33 36 short 0

dbase_data "IBT input format"
depth_m 1 4 short 0
temperature 5 8 short 2

RETURN "NEW LINE INDICATOR"

ASCII_data "ASCII output format"
depth_m 1 5 short 0
```

```
temperature 27 31 float 2
```

This format description file contains a header format description, a description for dBASE input data, the special `RETURN` descriptor, and a description for ASCII output data. The variable `count` (fourth from the bottom in the header format description) indicates the number of data records that follow each header. The descriptor `RETURN` lets **newform** skip over the end-of-line marker at the end of each data block in the input file **oceantmp.dab** as it is meaningless to **newform** here. Because the end-of-line marker appears at the end of the data records in each input data block, `RETURN` is placed after the input data format description in the format description file.

oceantmp.dab

```
         1         2         3         4         5         6         7
1234567890123456789012345678901234567890123456789012345678901234567890
11000171108603131109998  468602100000002767001027670020276700302767
11001175198607200569OAM  468609100000002928001028780020287200302872
11111176458102121909998  468101100000002728009126890241110005000728
112281795780051918090PI  26810110000000268900402711
```

Each dBASE header in **oceantmp.dab** is located from position 1 to 36. It is followed by four data records of 8 bytes each. Each record comprises a depth and temperature reading. The variable `count` in the header (positions 24-26) indicates that there are 4 data records each in the first 3 lines and 2 on the last line. This will all be more obvious after conversion.

To reformat **oceantmp.dab** to ASCII, use the following command:

newform oceantmp.dab -o oceantmp.dat

The resulting file **oceantmp.dat** is much easier to read. It is readily apparent that there are 4 data records after the first three headers and 2 after the last.

oceantmp.dat

```
         1         2         3         4
1234567890123456789012345678901234567890
11000171108603131109998  46860210000
    0                     27.67
```

```
   10                    27.67
   20                    27.67
   30                    27.67
1100117519860720056 90AM  46860910000
    0                    29.28
   10                    28.78
   20                    28.72
   30                    28.72
1111117645810212190 9998  46810110000
    0                    27.28
   91                    26.89
  241                    11.00
  500                    07.28
1122817957800519180 90PI  26810110000
    0                    26.89
   40                    27.11
```

CHAPTER 9

Data Checking

The FreeForm ND-based utility program **checkvar** creates variable summary files, lists of maximum and minimum values, and summaries of processing activity. You can use this information to check data quality and to examine the distribution of the data.

Generating the Summaries

A variable summary file (or list file), which contains histogram information showing the variable's distribution in the data file, is created for each variable (or designated variables) in the specified data file. You can optionally specify an output file in which a summary of processing activity is saved.

Variable summaries (list files) can be helpful for performing quality control checks of data. For example, you could run **checkvar** on an ASCII file, convert the file to binary, and then run **checkvar** on the binary file. The output from **checkvar** should be the same for both the ASCII and binary files. You can also use variable summaries to look at the data distribution in a data set before extracting data.

The **checkvar** command has the following form:

> **checkvar input_file [-f format file] [-if input_format_file] [-of output_format_file] [-ft "title"] [-ift "title"] [-oft "title"] [-b local_buffer_size] [-c +/-count] [-v var_file] [-q query_file] [-p precision] [-m maxbins] [-md missing_data_flag] [-mm] [-o processing_summary]**

Note that the **checkvar** program needs to find only an input format description. Output format descriptions will be ignored. If conversion variables are included in input or output formats, no conversion is performed when you run **checkvar**, since it ignores output formats.

For descriptions of the standard arguments (first eleven arguments above), see “Command Line Arguments” on page 51.

-p precision

Option flag followed by the number of decimal places. The number represents the power of 10 that data is multiplied by prior to binning. A value of 0 bins on one's, 1 on tenth's, and so on. This option allows an adjustment of the resolution of the **checkvar** output.

The default is 0; maximum is 5.

> *Note:* If you use the **-p** option on the command line, the precision set in the relevant format file is overridden. The precision in the format file serves as the default.

-m maxbins

Option flag followed by the approximate maximum number of bins desired in **checkvar** output. The **checkvar** program keeps track of the number of bins filled as the data is processed. The smaller the number of bins, the faster **checkvar** runs. By keeping the number of bins small, you can check the gross aspects of data distribution rather than the details.

The number of bins is adjusted dynamically as **checkvar** runs depending on the distribution of data in the input file. If the number of filled bins becomes > 1.5 * **maxbins**, the width of the bins is doubled to keep the total number near the desired maximum.

The default is 100 bins; minimum is 6. Must be < 10,000.

Note: The precision (**-p**) and maxbins (**-m**) options have no effect on character variables.

-md missing_data_flag

Option flag followed by a flag value that **checkvar** should ignore across all variables in creating histogram data. Missing data flags are used in a data file to indicate missing or meaningless data. If you want **checkvar** to ignore more than one value, use the query (**-q**) option in conjunction with the variable file (**-v**) option.

-mm

Option flag indicating that *only* the maximum and minimum values of variables are calculated and displayed in the processing summary. Variable summary files are *not* created.

-o processing_summary

Option flag followed by the name of the file in which summary information displayed during processing is stored.

Example

You will use checkvar with a precision of 3 to create a processing summary file and summary files for the two variables latitude and longitude in the file **latlon.dat.**

latlon.dat

```
-47.303545 -176.161101
 -0.928001    0.777265
-28.286662   35.591879
 12.588231  149.408117
-83.223548   55.319598
 54.118314 -136.940570
 38.818812   91.411330
-34.577065   30.172129
 27.331551 -155.233735
 11.624981 -113.660611
 77.652742  -79.177679
 77.883119  -77.505502
-65.864879  -55.441896
-63.211962  134.124014
 35.130219 -153.543091
 29.918847  144.804390
-69.273601   38.875778
-63.002874   36.356024
 35.086084  -21.643402
-12.966961   62.152266
```

To create the summary files, enter the following command:

checkvar latlon.dat -p 3 -o latlon.sum

A summary of processing information and the maximum and minimum for each variable are displayed on the screen. The following three files are created:

- **latlon.sum** recaps processing activity, maximums and minimums
- **latitude.lst** shows distribution of the latitude values in latlon.dat
- **longitude.lst** shows distribution of the longitude values in **latlon.dat**.

Interpreting the Summaries

The processing and variable summary files output by **checkvar** from the example in the previous section are shown and discussed below.

Processing Summary

If you specify an output file on the command line, it stores the information that is displayed on the screen during processing. The file **latlon.sum** was specified as the output file in the example above.

latlon.sum

```
Input file : latlon.dat
Requested precision = 3, Approximate number of sorting
bins = 100

Input data format(latlon.fmt)
ASCII_input_data"ASCII format"
The format contains 2 variables; length is 24.

Output data format(latlon.fmt)
binary_output_data"binary format"
The format contains 2 variables; length is 16.

Histogram data precision: 3, Number of sorting bins: 20
latitude: 20 values read
minimum: -83.223548 found at record  5
maximum:  77.883119 found at record 12
Summary file: latitude.lst

Histogram data precision: 3, Number of sorting bins: 20
longitude: 20 values read
minimum: -176.161101 found at record 1
maximum:  149.408117 found at record 4
Summary file: longitude.lst.
```

The processing summary file **latlon.sum** first shows the name of the input data file (`latlon.dat`). If you specified precision and a maximum number of bins on the command line, those values are given as

`Requested precision`, in this case `3`, and `Approximate number of sorting bins`, in this case the default value of `100`. If precision is not specified, `No requested precision` is shown.

A summary of each format shows the type of format (in this case, `Input data format` and `Output data format`) and the name of the format file containing the format descriptions (`latlon.fmt`), whether specified on the command line or located through the default search sequence (as detailed in chapter 4). In this case, it was located by default. Since **checkvar** only needs an input format description, it ignores output format descriptions. Next, you see the format descriptor as resolved by FreeForm ND (e.g., `ASCII_input_data`) and the format title (e.g., `"ASCII format"`). Then the number of variables in a record and total record length are given; for ASCII, record length includes the end-of-line character (1 byte for Unix).

A section for each variable processed by **checkvar** indicates the histogram precision and actual number of sorting bins. Under some circumstances, the precision of values in the histogram file may be different than the precision you specified on the command line. The default value for precision, if none is specified on the command line, is the precision specified in the relevant format description file or 5, whichever is smaller. The second line shows the name of the variable (`latitude`, `longitude`) and the number of values in the data file for the variable (`20` for both `latitude` and `longitude`).

The minimum and maximum values for the variable are shown next (`-83.223548` is the minimum and `77.883119` is the maximum value for `latitude`). The maximum and minimum values are given here with a precision of 6, which is the precision specified in the format description file. The locations of the maximum and minimum values in the input file are indicated. (`-83.223548` is the fifth latitude value in **latlon.dat** and `77.883119` is the twelfth). Finally, the name of the histogram data (or variable summary) file generated for each variable is given (`latitude.lst` and `longitude.lst`).

Variable Summaries

The name of each variable summary file (list file) output by **checkvar** is of the form **variable.lst** for numeric variables and **variable.cst** for character variables. The data in ***.lst**, and ***.cst** files can be loaded into histogram plot programs for graphical representation. (You must be familiar enough with your program of choice to manipulate the data as necessary in order to achieve the desired result.) In Unix, there is no need to abbreviate the base file name.

> *Note:* If you use the **-v** option, the order of variables in **var_file** has no effect on the numbering of base file names of the variable summary files.

The two example variable summary files, `latitude.lst` and `longitude.lst`, are shown next.

latitude.lst

```
-83.224  1
-69.274  1
-65.865  1
-63.212  1
-63.003  1
-47.304  1
-34.578  1
-28.287  1
-12.967  1
 -0.929  1
 11.624  1
 12.588  1
 27.331  1
 29.918  1
 35.086  1
 35.130  1
 38.818  1
 54.118  1
 77.652  1
 77.883  1
```

longitude.lst

```
-176.162 1
-155.234 1
-153.544 1
-136.941 1
-113.661 1
 -79.178 1
 -77.506 1
 -55.442 1
 -21.644 1
   0.777 1
  30.172 1
  35.591 1
  36.356 1
  38.875 1
  55.319 1
  62.152 1
  91.411 1
 134.124 1
 144.804 1
 149.408 1
```

The variable summary files consist of two columns. The first indicates boundary values for data bins and the second gives the number of data points in each bin. Because a precision of 3 was specified in the example, each boundary value has three decimal places. The boundary values are determined dynamically by **checkvar** and often do not correspond to data values in the input file, even if the **checkvar** and data file precisions are the same.

The first data bin in **latitude.lst** contains data points in the range `–83.224` (inclusive) to `–69.274` (exclusive); neither boundary number exists in **latlon.dat**. The first bin has one data point, `–83.223548`. The fourth data bin contains latitude values from `–63.212` (inclusive) to `–63.003` (exclusive), again with neither boundary value occurring in the data file. The data point in the fourth bin is `–63.211962`.

CHAPTER 10

Developing FreeForm ND Applications

As applications have become increasingly complex, the concept of layered application development has gained wide acceptance. A series of layers, each of which is as self-contained as possible, is used to interface between user and data. Interactions between layers are kept as simple as possible. FreeForm ND applications use this model and also incorporate the object-oriented approach to increase application power and efficiency while simplifying design and maintenance. As an application programmer, you can use the FreeForm ND Data Access System to build your own FreeForm ND-based programs.

FreeForm ND Application Layers

FreeForm ND applications are composed of the layers shown below.

The FreeForm ND Data Access System comprises the FreeForm ND Library and Data Objects layers. You, the application programmer, write the application-specific code and the user interface that sit above and make use of the FreeForm ND layers. The FreeForm ND

Library sits closest to the data. It includes functions for creating and interpreting format description files, and for reading, converting, and writing data.

The Data Object layer above the FreeForm ND Library consists of several types of objects that provide a simplified interface to the Library. Many common data access tasks have been implemented as events that the objects know how to accomplish. These objects are implemented as structures in the C programming language. The members of a structure are attributes of the object described by the structure.

Building an Application

You build a FreeForm ND application using the FreeForm ND library functions and data objects. To use an object in an application, you must complete three steps:

1. Create the object.
2. Set the object's attributes.
3. Send events to the object to trigger the desired action.

You can also include calls to **show** functions, e.g., **db_show**, to determine current characteristics of objects as the application runs.

Example Program

The example FreeForm ND application **getll.c** extracts and converts latitude and longitude values in any data file from their native format into a signed decimal degree representation. The program first defines a data bin with the native input format for a data file that includes latitude and longitude variables in any representation. Then it defines a compile-time format for just latitude and longitude, and reformats the latitude and longitude variables from their native format into the decimal degrees format.

Compile-time formats are used to read data from any hard-coded format into memory, where the data can then be accessed by applications. Unlike other formats, a compile-time format is not intended to be written to a file (although it could be). The example program **getll.c** demonstrates how to implement a compile-time format in a FreeForm ND-based application.

Source Code–getll.c

```
/*
 * NAME:getll
 *
 * PURPOSE:This program reads latitude and longitude in any recognized
 * format, converting to values in decimal degrees.
 *
 * AUTHOR: Ted Habermann, NGDC, (303) 497-6472, hab-
er@mail.ngdc.noaa.gov
 * Modified (MAO)
 *
 * USAGE:getll data_file
 *
 * COMMENTS:
 *
 * FreeForm ND applications are designed to run on many different types
of
 * computers. One of the differences between these computers is the
names
 * of various include files. These differences are taken care of by
defining
 * your environment by defining one of the following three preprocessor
 * macros:  1) CCMSC (PC, Microsoft C), 2) SUNCC (Unix workstation,
ANSI C),
 * or 3) CCLSC (Macintosh, ANSI C).
 *
 */

#include <limits.h>

/* The FreeForm ND include file is surrounded by a definition of the
 * constant DEFINE_DATA in the main program so that extern arrays that
 * FreeForm ND uses get initialized.  The DEFINE_DATA constant must not
be
 * defined in any other files.
 */

#define DEFINE_DATA
#include <FreeForm ND.h>
```

```
#undef DEFINE_DATA

/* This include file defines the data objects */
#include <databin.h>

#define ROUTINE_NAME "getll"

/* An error call back routine — it tells make_standard_dbin which events
   are okay if they fail.  getll "dynamically" creates the output data
   format, and throws away any existing output data format, so we don't
   require an output data format in the format file.  This function
allows
   make_standard_dbin() to process other events, even if the
OUTPUT_FORMAT
   event fails to produce an output format.
*/
#ifdef PROTO
static int mkstdbin_cb(int routine_name)
#else
static int mkstdbin_cb(routine_name)
int routine_name;
#endif
    {
    return(routine_name != OUTPUT_FORMAT);
    }

/
*********************************************************************
********
 * NAME:  check_for_unused_flags()
 *
 * PURPOSE:  Has user asked for an unimplemented option?
 *
 * USAGE:  check_for_unused_flags(std_args_ptr);
 *
 * RETURNS:  void
 *
 * DESCRIPTION:  All FreeForm ND utilities do not employ all of the
"standard"
 * FreeForm ND command line options.  Check if the user has unwittingly
asked
 * for any options which this utility will ignore.
 *
 * The following "standard" command line options are not used by this
 * application:
 *
 * -v  variable file
 * -q  query file
 * -p  precision (checkvar only)
```

```
 * -md missing data flag (checkvar only)
 * -m maximum number of bins (checkvar only)
 * -mm maximum/minimum processing only (checkvar only)
 *
 * AUTHOR:  Mark Ohrenschall, NGDC
 *
 * SYSTEM DEPENDENT FUNCTIONS:
 *
 * GLOBALS:
 *
 * COMMENTS:
 *
 * KEYWORDS:
 *
 * ERRORS:

*********************************************************************
*******/

#ifdef PROTO
static void check_for_unused_flags(FFF_STD_ARGS_PTR std_args)
#else
static void check_for_unused_flags(std_args)
FFF_STD_ARGS_PTR std_args;
#endif
    {
    if (std_args->user.set_var_file)
        {
        err_push(ROUTINE_NAME, ERR_IGNORED_OPTION,
                 "variable file"
                );
        }

    if (std_args->user.set_query_file)
        {
        err_push(ROUTINE_NAME, ERR_IGNORED_OPTION,
                 "query file"
                );
        }

    if (std_args->user.set_cv_precision)
        {
        err_push(ROUTINE_NAME, ERR_IGNORED_OPTION,
                 "precision (checkvar only)"
                );
        }

    if (std_args->user.set_cv_missing_data)
        {
```

```
        err_push(ROUTINE_NAME, ERR_IGNORED_OPTION,
                 "missing data flag (checkvar only)"
                );
        }

    if (std_args->user.set_cv_maxbins)
        {
        err_push(ROUTINE_NAME, ERR_IGNORED_OPTION,
                "maximum number of histogram bins (checkvar only)"
                );
        }

    if (std_args->user.set_cv_maxmin_only)
        {
        err_push(ROUTINE_NAME, ERR_IGNORED_OPTION,
                 "maximum and minimum processing only (checkvar only)"
                );
        }

    if (err_state())
        {
        err_disp();
        }
    }

#ifdef PROTO
void main(int argc, char *argv[])
#else
void main(argc, argv)
int argc;
char *argv[]
#endif
    {
    int error = 0;    /* to hold error return values */

    char *output_buffer = NULL; /* output data buffer */
    long  output_bytes  = 0;    /* bytes written into output buffer */

    FFF_STD_ARGS  std_args; /* holds command line information */
    DATA_BIN_PTR  input         = NULL; /* the data bin */
    FILE         *pfile         = NULL; /* output file */

    if (argc == 1)
        {
        fprintf(stderr, "%s%s",
#ifdef ALPHA
"\nWelcome to getll alpha "FF_LIB_VER" "__DATE__\
" -- an NGDC FreeForm ND example application\n\n",
#elif defined(BETA)
```

```
"\nWelcome to getll beta "FF_LIB_VER" "__DATE__\
" -- an NGDC FreeForm ND example application\n\n",
#else
"\nWelcome to getll release "FF_LIB_VER\
" -- an NGDC FreeForm ND example application\n\n",
#endif
"Default extensions: .bin = binary, .dat = ASCII, .dab = dBASE\n\
\t.fmt = format description file\n\
\t.bfm/.afm/.dfm = binary/ASCII/dBASE variable description file\n\n\
getll data_file [-f format_file] [-if input_format_file]\n\
                [-of output_format_file] [-ft \"format title\"]\n\
                [-ift \"input format title\"] [-oft \"output format
                 title\"]\n\
              [-c count] No. records to process at head(+)/tail(-) of
                 file\n\
                [-o output_file] default = output to screen\n\n\
See the FreeForm ND User's Guide for detailed information.\n"
              );
        exit(EXIT_FAILURE);
        }

     /* The FreeForm ND system uses a hierarchical error handling system
     which allows each layer of an application to add error messages to
     a queue. err_push is the function which adds messages to the queue.
     It is called by any function which runs into an error. err_disp is
     the function that interactivly displays those errors to the user.
     It is called by the main application program when an error oc-
curs.*/

     /* Allocate the output buffer:
     FreeForm ND uses two types of buffers extensively and defines de-
fault
     buffer sizes in the include file FreeForm ND.h. The local or
scratch
     buffers are used as temporary work space. The cache buffers are
     used for reading large blocks of data.*/

    output_buffer = (char *)malloc((size_t)DEFAULT_CACHE_SIZE);
    if (!output_buffer)
        {
        err_push(ROUTINE_NAME, ERR_MEM_LACK, "Output Buffer");
        err_disp();
        exit(EXIT_FAILURE);
        }

    /* Collect options entered on the command line, this information
will be
       used in the call to make_standard_dbin(), below.
    */
```

```
    if (parse_command_line(argc, argv, &std_args))
        {
        free(output_buffer);

        err_disp();
        exit(EXIT_FAILURE);
        }
    check_for_unused_flags(&std_args);

    /* Create and initialize the data bin */
    if (make_standard_dbin(&std_args, &input, mkstdbin_cb))
        {
        free(output_buffer);

        err_disp();
        exit(EXIT_FAILURE);
        }

    /* make_standard_dbin may have generated an incidental error, in
case
       the OUTPUT_FORMAT event failed.  mkstdbin_cb downgrades such an
error
       from a terminal error to a warning, but an error message might
still
       have been queued.  If so, clear it.
    */
    if (err_state())
        err_clear();

    /* Has user indicated an output file? */
    if (std_args.output_file)
        {
        pfile = fopen(std_args.output_file, "wb");
        if (!pfile)
            {
            free(output_buffer);

            err_push(ROUTINE_NAME, ERR_CREATE_FILE,
                     std_args.output_file);
            err_disp();
            exit(EXIT_FAILURE);
            }
        }
    else
        {
        /* If not, write to standard output */
        pfile = stdout;
        }
```

```
    /* Rather than using an output format contained in a file, create a
       "dynamic" buffer, write an output format description into it, and
       use that to initialize the data bin's output format
    */

    sprintf(output_buffer, "\
ASCII_output_data \"hard-coded in getll.c:main()\"\n\
longitude  1 11 double 6\n\
latitude  13 25 double 6\n"
         );

  /* Use the FORMAT_BUFFER event to set the output format.  The data bin
     knows that this is an output format because of the format type,
     "ASCII_output_data".
  */
    if (db_set(input,
               FORMAT_BUFFER, output_buffer, NULL, NULL,
               END_ARGS
                )
       )
        {
    /* Error in the output format creation -- this must never happen!
       Ensure that the output buffer is syntactically correct, since it
       is hard-coded into the program!
    */
        free(output_buffer);
        if (std_args.output_file)
            fclose(pfile);

        err_push(ROUTINE_NAME, ERR_MAKE_FORM, output_buffer);
        err_disp();
        exit(EXIT_FAILURE);
        }

    /* Display some information about the data formats */
    db_show(input, FORMAT_LIST, FFF_INFO, END_ARGS);
    /* db_show writes into data bin's working buffer */
    fprintf(stderr, "%s", input->buffer);

    /*
    ** process the data
    */

    /* use PROCESS_FORMAT_LIST to fill cache and fill headers */
    while ((error = db_events(input,
                             PROCESS_FORMAT_LIST, FFF_ALL_TYPES,
                             END_ARGS
                           )
         ) == 0
```

```
        )
        {
        /* Make sure output buffer is large enough for the cache */
        db_show(input,
                  DBIN_BYTE_COUNTS, DBIN_OUTPUT_CACHE, &output_bytes,
              END_ARGS, END_ARGS
               );

        if ((unsigned long)output_bytes > (unsigned long)UINT_MAX)
            {
            error = 1;
            err_push(ROUTINE_NAME, ERR_MEM_LACK,
                      "reallocation size too big");
            break;
            }
        if (output_bytes > DEFAULT_CACHE_SIZE)
            {
            /* The default cache size was too small for the number of
               output bytes needed.  This contigency is coded for, but
              is extremely unlikely to happen.  However, it is possible
               that the program will error out if it can not resize the
               output buffer.
            */
            char *cp = NULL;

            cp = (char *)realloc(output_buffer, (size_t)output_bytes);
            if (cp)
                    {
                    output_buffer = cp;
                    }
            else
                    {
                    error = 1;
                    err_push(ROUTINE_NAME, ERR_MEM_LACK,
                             "reallocation of output_buffer");
                    break;
                    }
            }

        /* Convert the cache into the output_buffer -- this will perform
a
            binary to ASCII conversion if necessary.  (The example shows
             getll running on llmaxmin.dat, an ASCII file, but this pro-
gram
            works equally well on llmaxmin.bin, which is created by run-
ning
             newform on llmaxmin.dat.)
        */
        error = db_events(input,
```

```
                              DBIN_DATA_TO_NATIVE, NULL, NULL, NULL,
                             DBIN_CONVERT_CACHE, output_buffer, NULL,
                              &output_bytes,
                          END_ARGS
                         );
        if (error)
            break;

        /* Write the contents of the output buffer to the file
           (or screen).
        */
        if (  (long)fwrite(output_buffer, sizeof(char),
                             (size_t)output_bytes, pfile)
             < output_bytes
            )
             {
            err_push(ROUTINE_NAME, ERR_WRITE_FILE,
std_args.output_file
                                                 ? std_args.output_file
                                                   : "to screen"
                    );
            break;
            }
        }/* End Processing */

    if (std_args.output_file)
        fclose(pfile);

    free(output_buffer);

    /* Deallocate all memory assocated with the data bin */
    db_free(input);

    /* The error stack is checked to see if anything went wrong during
       processing
    */
    if ((error && error != EOF) || err_state())
        {
        err_push(ROUTINE_NAME, ERR_PROCESS_DATA, NULL);
        err_disp();
        exit(EXIT_FAILURE);
        }

    } /* end main() for program getll */
```

Using getll

In this example, you will use **getll** to extract latitude and longitude values from the ASCII data file **latlon.dat**. Their native format is signed decimal degrees, so no conversion takes place. Enter the following command:

getll latlon.dat

This command prints format summary information and a list of longitude and latitude values to the screen:

```
 -176.161101   -47.303545
    0.777265    -0.928001
   35.591879   -28.286662
  149.408117    12.588231
           .
           .
           .
```

As another example, use the following command to extract the latitude and longitude values from the file **calif.tap**.

getll calif.tap -f eqtape.fmt

The latitude and longitude values are extracted and converted from their native representation as absolute values with quadrant indicated. You should see the following signed decimal degree values written to the screen:

```
-121.815000     37.852000
-121.740000     37.737000
-116.550000     33.517000
         .
         .
         .
```

Writing out latitude and longitude values to standard output (the screen) is not a very impressive feat, but you could create a similar program to use as the front end for a graphics package. In that case, you might want to include a third output column which contains the values of a third variable. For example, with a seismological application, you might want to include values for magnitude or depth. You can easily add the third column to the **getll** program by changing the `sprintf` statement, which creates the compile-time format, so it is similar to the following:

```
sprintf(output_buffer, "longitude 1 8 double 6\nlatitude
9 16 double 6\n
         %s 17 24 double 2", *(argv+2));
```

The **getllvar** program incorporates a more general version of the `sprintf` statement and several other small changes to the **getll** code; see **getllvar.c**. Now the second command line argument is the name of the third variable (only with **getllvar**; this is at variance with standard FreeForm ND command line syntax). Try the following command:

getllvar calif.tap depth -f eqtape.fmt

You should see the following output with values for depth given in the third column:

```
  -121.815000     37.852000         11
  -121.740000     37.737000         15
  -116.550000     33.517000          6
  -125.033000     40.600000          5
  -118.840000     37.600000          5
  -118.875000     37.609000         24
  -118.832000     37.527000         12
  -118.820000     37.569000         15
         .
         .
         .
```

If you enter the following command:

getllvar calif.tap year -f eqtape.fmt

the year will be shown in the third column.

CHAPTER 11

Conversion Variable Names

FreeForm ND can automatically perform conversions between various representations of space and time values. When FreeForm ND encounters standard conversion variable names in a format description file, it performs the appropriate conversion.

This appendix lists the conversion variable names that FreeForm ND recognizes. For conversions that occur in one direction only, the conversion variable names are listed in columns titled **Input** and **Output**. Conversion names that you can use for either input or output variables are in untitled columns.

General

By adding the appropriate suffix to a variable name, you can perform several general (miscellaneous) conversions. To convert between meters and feet, add `_m` and `_ft` to the relevant variable names. For absolute and signed values, add `_abs` and `_sign`. For scientific notation, use `_base` and `_exp` to identify the base and exponent parts of a number. There may or may not be an 'E' or 'e' in the exponent; the range of the E format is E+/-999. Be sure the output field is large enough for the converted number.

varname_m	varname_ft
varname_abs varname_sign	varname
varname_base varname_exp	varname

`varname` = any character string without blanks

Latitude/Longitude

FreeForm ND supports conversions between a number of representations of latitude and longitude values with the use of the correct conversion variable names.

General Lat/Lon

By using the appropriate suffixes, you can perform conversions between a number of different representations of latitudes and longitudes.

Input	Output
varname_abs varname_ns or varname_ew	varname
varname or varname_abs varname_ns or varname_ew	varname_deg varname_min varname_sec
varname	varname_abs
varname or varname_abs	varname_deg_ varname_min_ varname_sec_
varname_deg varname_min varname_sec	varname or varname_a
varname_deg or _abs varname_min or _abs varname_sec or _abs geo_quad_code _ns, _ew or _sign	varname

varname = latitude or longitude

Degrees, Minutes, Seconds

The following variables are used for conversions between degrees, minutes, and seconds (_deg, _min, _sec) or absolute degrees, minutes, and seconds (`_deg_abs`, `_min_abs`, `_sec_abs`) and decimal degrees (no suffix). If a conversion to degrees, minutes and seconds results in a value for degrees between 0 and -1, the minutes or seconds part is signed as appropriate to avoid a value of `-0` degrees.

```
varname_deg                    varname
varname_min
varname_sec


varname_deg_abs                varname
varname_min_abs
varname_sec_abs
```

`varname` = `latitude` or `longitude`

Geographic Quadrants

Use the following variables to convert from several different representations of latitude and longitude to a geographic quadrant defined by DMA (Defense Mapping Agency) for their gravity data.

DMA defines four geographic quad codes as follows:

1 = Northeast2 = Northwest

3 = Southeast4 = Southwest

Input	Output
latitude longitude	geo_quad_code
latitude_ns longitude_ew	geo_quad_code
latitude_sign longitude_sign	geo_quad_code

Longitude East

Use the following variables for conversions between east longitudes (`longitude_east`) and `longitude` (no suffix) or longitude represented in degrees/minutes/seconds (`_deg`, `_min`, `_sec`) plus hemisphere or geographic quadrant (`_ns`, `_ew`, `geo_quad_code`) .

Input	Output
longitude_east	longitude
longitude	longitude_east
varname_min varname_sec varname_ns varname_ew geo_quad_code	longitude_east

Quadrant, Sign

Use the following variables to convert from lat/lon with quadrant to signed lat/lon or vice versa.

latitude_ns	latitude_sign
longitude_ew	longitude_sign

Earthquake Magnitude

FreeForm ND includes a conversion function that lets you extract one of the three magnitudes out of the variable `longmag`, or create `longmag` from one, two, or more of the individual magnitudes.

The variable `longmag` is a long which contains three magnitudes:

`ms2`has a precision of 2 and is multiplied by 10,000,000
`ms1`has a precision of 2 and is multiplied by 10,000
`mb`has a precision of 1 and is multiplied by 10

longmag	magnitude_mb magnitude_ms1 magnitude_ms2 magnitude_ml magnitude_local

Date and Time

FreeForm ND includes conversion functions that let you convert between various representations of date and time including decimal year; serial date with January 1, 1980 as 0; any combination of year, month, day, hour, minute, second; and IPE (Institute of Physics of the Earth) date in minutes AD.

year month day hour minute second	serial_day_1980
serial_day_1980 year month day	ipe_date ipe_date
date_dd/mm/yy time_hhmmss	date_ddmmyy time_hh:mm:ss

CHAPTER 12

Error Handling

The FreeForm ND error handling system captures errors, such as improper usage, code problems, and system errors, and places them in an error queue. For each error captured, error type and a short message are placed in the message queue. If a fatal error occurs, the program stops executing and displays all error messages in the queue.

Error Messages

The following is a list of some possible error messages with suggestions for corrections.

```
Problem opening, reading, or writing to file
```

Check that all file names and paths are correct.

```
Problem making format
```

Make sure there is a format file describing the data file formats. Check that input and output format descriptions in the format file accurately describe the data.

```
Problem making header format
```

If a header exists in the data file, it must be described in a format file. Check that the header description accurately describes the header in your data file.

```
Problem getting value
Problem processing variable list
```

The data formats may not be described correctly or there may be some inconsistencies in the data.
Check also for unprintable characters at the end of the data file.

```
File length / Record length mismatch
Record Length or CR Problem
```

This usually happens because the input format description is not correct.
Make sure the format description's last position is the last character before the end-of-line character. If you have a header, make sure it is described correctly.
The header's length must include all characters up until the last end-of-line-character before the data begins.

```
Binary Overflow
```

Try using a larger output variable type such as a long instead of a short.
Be sure you have given enough space for the values to be written.
See “Conversion Variables” on page 75 for more information.

```
Variable not found
```

The variable names in your output format must match the variable names in the input format unless you are using conversion variables.

```
Data Overflow:
******
```

Data overflow does not usually cause a fatal error and FreeForm ND functions try to anticipate them. If overflow occurs for a particular value, ***'s are written to that value's location.

If you find these in your output, check your variable positions and precision. Increase field width or use a “larger” data type.

Be sure the output format specifies space for the output variable. For instance, FreeForm ND adds a leading zero in front of decimal points. If the original data did not have a leading zero, the output will have one more digit than the input.

`Insufficient memory allocation`

The application has run out of memory. Try using the **-b** (local buffer size) option, or modify **autoexec.bat** and **config.sys** and comment out devices, TSR’s, etc.

CHAPTER 13

Query Syntax

This appendix lists the operators, symbols, and functions you can use to construct queries. The lists are followed by definitions, rules for combining elements to form equations and queries, and brief usage explanations.

Symbols and Operators–List

Arithmetic Operators

Rep.	Meaning
^	exponentiation
%	modulus
*	multiplication
/	division
+	addition
-	subtraction

Logical Operators

<table>
<tr><th>Rep.</th><th>Meaning</th></tr>
<tr><td>!</td><td>logical not (takes only 1 argument)</td></tr>
<tr><td>not</td><td>logical not (takes only 1 argument)</td></tr>
<tr><td>&</td><td>logical and</td></tr>
<tr><td>&&</td><td>logical and</td></tr>
<tr><td>and</td><td>logical and</td></tr>
<tr><td>|</td><td>logical or</td></tr>
<tr><td>||</td><td>logical or</td></tr>
<tr><td>or</td><td>logical or</td></tr>
<tr><td>x|</td><td>logical exclusive or</td></tr>
<tr><td>xor</td><td>logical exclusive or</td></tr>
<tr><td>=</td><td>equal to</td></tr>
<tr><td>==</td><td>equal to</td></tr>
<tr><td><</td><td>less than</td></tr>
<tr><td>></td><td>greater than</td></tr>
</table>

Rep.	**Meaning**
!=	not equal to
< >	not equal to
> <	not equal to
< =	less than or equal to
> =	greater than or equal to

Special Symbols

Rep.	Meaning
~	negative sign
( )	indicate order in which expressions are evaluated
[ ]	enclose variables
""	enclose string constants

Functions–List

Name	Meaning
acosh	inverse hyperbolic cosine
asinh	inverse hyperbolic sine
atanh	inverse hyperbolic tangent
asech	inverse hyperbolic secant
acsch	inverse hyperbolic cosecant
acoth	inverse hyperbolic cotangent
acos	inverse cosine (radians)
asin	inverse sine (radians)
atan	inverse tangent (radians)
asec	inverse secant (radians)
acsc	inverse cosecant (radians)
acot	inverse cotangent (radians)
cosh	hyperbolic cosine
sinh	hyperbolic sine
tanh	hyperbolic tangent
sech	hyperbolic secant
csch	hyperbolic cosecant
coth	hyperbolic cotangent

Nam e	Meaning
sqrt	square root
sign	sign of argument (1 if pos, -1 if neg, 0 if 0)
cos	cosine (radians)
sin	sine (radians)
tan	tangent (radians)
sec	secant (radians)
csc	cosecant (radians)
cot	cotangent (radians)
abs	absolute value
exp	e to the power
log	logarithm base 10
fac	factorial
deg	radians to degrees
rad	degrees to radians
rup	round to nearest larger integer
rdn	round to nearest smaller integer
rnd	round to nearest integer
sqr	square
ten	ten to the power
not	logical not
ln	logarithm base e

Definitions of Terms

Constant. A number whose value is explicitly stated in the query.

Predefined Constant. A number whose value is known explicitly by the equation interpreter, but is not stated in the equation. The two predefined constants, with names preceded by a colon, are **:e** (2.71828182846) and **:pi** (3.14159265359).

String Constant. A character string whose value is explicitly stated in the equation.

Variable. A number which is referenced by a unique name in the equation, but whose value is not stated in the equation.

String Variable. A character string which is referenced by a unique name in the equation, but whose value is not stated in the equation.

Domain Error. A problem which arises when a function or operation is undefined for certain input values, such as division by 0. If a domain error is generated, the equation is evaluated to 0.

Rules

Equations or queries are formed by combining variables and constant values with operators or functions in a meaningful way. The following set of rules applies to creating an equation.

- Variable names must be enclosed in [ ] (square brackets).
  For instance, finding the sum of a variable named **height** and another variable named **altitude** is expressed as:
  [height] + [altitude]
- String constants must be surrounded by " " (quotation marks).
  For instance, a query to see if a string variable **latitude** is equal to the string **north** is expressed as:
  [latitude] = "north"
- Constants with negative values must be proceeded by ~ (tilde).
  For instance, multiplying the variable altitude by a negative four is expressed as:
  [altitude] * ~4
- At least one variable (numeric or string) must be referenced in the equation.

- Variable names cannot contain " (quotation marks), or [ ] (square brackets).
- Spaces are ignored in equations (except inside string constants and variable names), so you can use spaces to separate the parts of your equation and make it more readable.

Pre-defined Constants

The names of the two predefined constants **e** and **pi** are preceded by a colon to differentiate them from a function name or variable name.

Example

```
[degrees] * :e + :pi
```

multiplies the variable **degrees** by **e** (2.718Ö) and adds **pi** (3.141Ö)

Operators–with Definitions

Operators cause the indicated operation to be performed on two values (string or numeric) with a third value resulting. The format for using operators is as follows:
value1 **operator** *value2*

Arithmetic Operators

Arithmetic operators cannot be used with string variables or string constants.

Symbol	Meaning	Explanation
^	exponentiation	Raises *value1* to the *value2* power. Generates a domain error if *value1* is negative and *value2* is not an integer.
%	modulus	Returns the remainder when *value1* is divided by *value2*. If *value1* = *value2* * *a* + *R*, where *a* is an integer and *R* is less than *value2*, the modulus operator returns *R*. Generates a domain error if *value2* is 0.
*	multiplication	Multiplies *value1* by *value2*.
/	division	Divides *value1* by *value2*. Generates a domain error if *value2* is 0.
+	addition	Adds *value1* and *value2*.
-	subtraction	Subtracts *value2* from *value1*.

Logical Operators

Inputs to the logical operators are evaluated to FALSE if they are equal to 0. Any value other than 0 evaluates to TRUE. Outputs of the logical operators are 0 for FALSE, 1 for TRUE.

Some of the logical operators have a “word” which is synonymous with their symbol (C language compatible), or multiple acceptable symbols. There is no advantage in speed to using any one of these

alternate names, but the equation may be more human-readable in one form than in another. For example, the following equations, which evaluate to TRUE (i.e., 1) if the variables **x** and **y** are TRUE, are all equivalent:

```
[x] & [y]
[x] && [y]
[x] and [y]
```

Symbol	**Meaning**	**Explanation**
&	logical AND	TRUE if *value1* and *value2* are both TRUE
&&	logical AND	TRUE if *value1* and *value2* are both TRUE
and	logical AND	TRUE if *value1* and *value2* are both TRUE
		Logical AND accepts only numeric arguments.
		Truth table for logical AND:
		<u>value1</u> <u>value2</u> <u>output</u> FALSE FALSE FALSE FALSE TRUE FALSE TRUE FALSE FALSE TRUE TRUE TRUE
|	**logical OR**	**TRUE if *value1* or *value2* are TRUE**
||	logical OR	TRUE if *value1* or *value2* are TRUE
or	logical OR	TRUE if *value1* or *value2* are TRUE
		Logical OR accepts only numeric arguments.
		Truth table for logical OR:
		<u>value1</u> <u>value2</u> <u>output</u> FALSE FALSE FALSE FALSE TRUE TRUE TRUE FALSE TRUE TRUE TRUE TRUE
x |	logical exclusive or (XOR)	TRUE if *value1* or *value2* are TRUE, but not both

Symbol	Meaning	Explanation
xor	logical exclusive or (XOR)	TRUE if *value1* or *value2* are TRUE, but not both
		Logical XOR accepts only numeric arguments
		Truth table for logical XOR:
		<u>value1</u> <u>value2</u> <u>output</u> FALSE FALSE FALSE FALSE TRUE TRUE TRUE FALSE TRUE TRUE TRUE FALSE
=	equal to	TRUE if *value1* is equal to *value2*
==	equal to	TRUE if *value1* is equal to *value2*
		This operator can be used with both numeric and string values as long as *value1* and *value2* are both of the same type.
<	less than	TRUE if *value1* is less than *value2*
		This operator can be used with both numeric and string values, as long as *value1* and *value2* are both of the same type.
>	greater than	TRUE if *value1* is greater than *value2* This operator can be used with both numeric and string values, as long as *value1* and *value2* are both of the same type.
!=	not equal to	TRUE if *value1* is not equal to *value2*
<>	not equal to	TRUE if *value1* is not equal to *value2*
><	not equal to	TRUE if *value1* is not equal to *value2*
		This operator my be used with both numeric and string values, as long as *value1* and *value2* are both of the same type.
<=	less than or equal to	TRUE if *value1* is less than or equal to *value2*
		This operator can be used with both numeric and string values, as long as *value1* and *value2* are both of the same type.

Symbol	Meaning	Explanation
>=	greater than or equal to	This operator can be used with both numeric and string values, as long as *value1* and *value2* are both of the same type.
!	logical NOT	TRUE if *value* is FALSE, FALSE if *value* is TRUE
not	logical NOT	TRUE if *value* is FALSE, FALSE if *value* is TRUE Logical NOT accepts only numeric arguments. The logical NOT operator, unlike all other logical operators, takes only 1 argument. Thus, the format for a logical NOT statement is as follows: ! *value* not *value*

Functions–with Definitions

The functions take only a single argument, in the following manner:

name(*value*)

Please note that the parentheses implied above are not necessary unless the function is evaluating a complex argument. In the definitions given below, the value is represented as **x**. Function definitions which require functions themselves are given in a manner compliant with the equation evaluator.

Nam e	Meaning	Explanation/Definition
acos h	inverse hyperbolic cosine	ln(x + sqrt((x ^ 2) -1)) Domain error if x < 1.
asin h	inverse hyperbolic sine	n(x + sqrt((x ^ 2) + 1))
atan h	inverse hyperbolic tangent	ln((1 + x) / (1 -x)) / 2 Domain error if x >= 1 or x <= -1
asec h	inverse hyperbolic secant	n((1 + sqrt(1 -(x ^ 2))) / x) Domain error if x <= 0 or x > 1
acsc h	inverse hyperbolic cosecant	ln(((1 / x) + (sqrt(1 + (x ^ 2))) / abs(x))) Domain error if x = 0
acot h	inverse hyperbolic cotangent	log((x + 1) / (x -1)) / 2 Domain error if -1 <= x <= 1
acos	inverse cosine (radians)	Domain error if x < -1 or x > 1
asin	inverse sine (radians)	Domain error if x < -1 or x > 1
atan	inverse tangent (radians)	
asec	inverse secant (radians)	Domain error if -1 < x < 1
acsc	inverse cosecant (radians)	Domain error if -1 < x < 1
acot	inverse cotangent (radians)	Domain error if x = 0
cosh	hyperbolic cosine	
sinh	hyperbolic sine	
tanh	hyperbolic tangent	
sech	hyperbolic secant	
csch	hyperbolic cosecant	
coth	hyperbolic cotangent	

Name	Meaning	Explanation/Definition
sqrt	square root	Domain error if x < 0
sign	sign of argument	Evaluates to 1 if x > 0, 0 if x = 0, and -1 if x < 0
cos	cosine (radians)	
sin	sine (radians)	
tan	tangent (radians)	
sec	secant (radians)	
csc	cosecant (radians)	
cot	cotangent (radians)	
abs	absolute value	
exp	e to the power	:e ^ x
log	logarithm base 10	Domain error if x < = 0
fac	factorial	Domain error if x < = 0 x is rounded to nearest smaller integer before factorial is calculated.
deg	radians to degrees	180 * x / :pi
rad	degrees to radians	:pi * x / 180
rup	round to nearest larger integer	
rdn	round to nearest smaller integer	
rnd	round to nearest integer	
sqr	square	x ^ 2
ten	ten to the power	10 ^ x
not	logical not	This is the same as the logical NOT operator, but is included here because of its function-like behavior. Evaluates to 1 if x = 0, 0 otherwise.
ln	logarithm base e	Domain error if x <= 0

Order of Operations

An equation is evaluated in the following order:

1. anything inside parentheses (left to right, sub-parentheses given precedence)
2. functions (no explicit order to function evaluation)
3. exponentiation (left to right)
4. multiplication, division, and modulus (left to right)
5. addition and subtraction (left to right)
6. logical operators (no explicit order to logical operator evaluation)

For instance, the equation

```
4 + (cos[x] - [y] ^ 3) * (([y] + 4) / 7) + ([x] > 1)
```

is evaluated as follows: ([x] = 3.14159265359, [y] = 3, bold is changed value)

```
4 + (~1 - [y] ^ 3) * (([y] + 4) / 7) + ([x] > 1)
4 + (~1 - 27) * (([y] + 4) / 7) + ([x] > 1)
4 + ~28 * (([y] + 4) / 7) + ([x] > 1)
4 + ~28 * (7 / 7) + ([x] > 1)
4 + ~28 * 1 + ([x] > 1)
4 + ~28 * 1 + 1
4 + ~28 + 1
~24 + 1
~23
```

The similar equation

```
4 + (cos[x] - [y] ^ 3) * (([y] + 4) / 7) + [x] > 1
```

is evaluated as follows: (with the same values for [x] and [y])

```
4 + (~1 - [y] ^ 3) * (([y] + 4) / 7) + [x] > 1
4 + (~1 - 27) * (([y] + 4) / 7) + [x] > 1
4 + ~28 * (([y] + 4) / 7) + [x] > 1
4 + ~28 * (7 / 7) + [x] > 1
4 + ~28 * 1 + [x] > 1
```

```
4 + ~28 + [x] > 1
~24 + [x] > 1
~20.8584073464 > 1
0
```

General Suggestions

It is best to use the supported operators and functions in order to reduce the number of items to be evaluated. This reduces the time it takes to evaluate the equation, perhaps negligible for each evaluation, but with repeated evaluation, the time savings can be substantial. For instance, the equation:

```
ln(((1 / [x]) + (sqrt(1 + ([x] ^ 2))) / abs([x])))
```

is equivalent to:

```
acsch[x]
```

but the first equation takes much longer to evaluate than the second. There are a few cases where there are various ways to state the same equation, such as:

```
[x] * [x]
[x] ^ 2
sqr[x]
```

All three equations above square the variable **x**. All three require only one evaluation each, and therefore require almost exactly the same amount of time to evaluate. The equations below are also completely equivalent:

```
[x] * [x] * [x]
[x] ^ 3
```

In this case, the first equation requires two evaluations ([x] * [x], and then the result of that * [x]), while the second equation requires only one evaluation. The evaluation of the second equation will therefore be approximately 2 times faster.

Avoid causing unnecessary evaluations. For instance, the following equation:

```
(1 / 2) * [base] * [height]
```

is faster if expressed as:

```
[base] * [height] / 2
```

Examples

Absolute Latitude

The following equation takes the variable **abs_latitude** and multiplies it by -1 if the value of **latitude_n_or_s** is S, **abs_latitude** is multiplied by 1 otherwise.

```
[abs_latitude] * (([latitude_n_or_s] != "S") + ~1 *
([latitude_n_or_s] = "S"))
```

Distance Between Points

The following equation computes the distance between the points given by **x1**, **y1**, **x2** and **y2** (assuming all points lie on a plane).

```
sqrt(sqr([x2] - [x1]) + sqr([y2] - [y1]))
```

Quadratic Solution

The following equation computes one of the solutions to the quadratic formula:

```
(~1 * [b] + sqrt([b] ^ 2 - 4 * [a] * [c])) / (2 * [a])
```

Sine of Angle

The following 3 equations are all roughly equivalent, finding the sine of angle **deg** (which is measured in degrees). The first equation is not recommended, because the value used for pi is not as accurate as the value given by the pre-defined constant **:pi**. The second equation is better, but requires more time to be evaluated than the third.

```
sin(3.141592 * [deg] / 180)
sin(:pi * [deg] / 180)
sin(rad[deg])
```

Volume of Sphere

The following equation finds the volume of a sphere:

```
4 * :pi * [radius] ^ 3 / 3
```