GRADS

Table of Contents

General Topics

Starting and Quitting GrADS

GrADS is started by entering the command: `grads`

Before initialising the graphics output environment, GrADS will prompt for landscape or portrait mode. Landscape is 11 x 8.5 inches (usually what you want). Portrait is 8.5 x 11 inches, primarily used for producing vertically oriented hardcopy output. The actual size of the window will not, of course, be 11 x 8.5 inches (or 8.5 x 11 inches), but instead will be whatever size you chose by using your workstation's window manager. But GrADS will treat the window as if it were one of the above sizes, so it is best to size the window with approximately the proper aspect ratio. This can be done using the window manager or from GrADS using the command:

```
set xsize x y
```

which resizes the window to `x,y` pixels.

After answering this prompt, a separate graphics output window will be opened (but not on PCs). You may move or resize this window at any time.

You will enter GrADS commands in the text window from where you started GrADS. Graphics output will appear in the graphics window in response to the commands you enter. You will thus need to make the text window the "active" window; the window that receives keyboard input.

Help

Typing `help` at the GrADS command prompt gives a summary list of operations essential to do anything in GrADS. This is intended to jog memory rather than provide an exhaustive help facility. If the GrADS manual is not available, you can obtain info on most command parameters by typing the command on its own. Alternatively, we are setting up comprehensive documentation intended to be used as a local Web facility.

Diagnostics at startup

When you start GrADS you get platform specific diagnostics, for example:

```
GX Package Initialization: Size = 11 8.5
!!!! 32-bit BIG ENDIAN machine version
ga>
```

The `!!!!` line tells you that this version is `32-bit` (i.e., data are 32-bit) and it was compiled for a `big endian` machine (the Sun in this case). On the Cray you get...

```
!!!! 64-BIT MACHINE VERSION (CRAYS)
```

Startup options

You may specify the following options as arguments to the `grads` command when GrADS is started:

`-b` Run grads in batch mode. No graphics output window is opened.
`-l` Run grads in landscape mode. The Portrait vs. Landscape question is not asked.
`-p` Run grads in portrait mode.
`-c` Execute the supplied command as the 1 st GrADS command after GrADS is started.

An example:

```
grads -c "run profile.gs"
```

These options may be used in combinations. For example:

```
grads -blc "run batch.gs"
```

Would run grads in batch mode, using landscape orientation (thus no questions are asked at startup); and execute the command:

`batch.gs` upon startup.

Leaving GrADS

To leave GrADS, enter the command:

- `quit`
-

Basic Concept of Operation

When you have successfully installed and started GrADS, you'll be confronted with two windows -- a terminal window with a prompt, much like the infamous C:> in MS-DOS, and a resizable window (black background by default) where graphics are displayed.

GrADS commands are entered in the terminal window and the response from GrADS is either graphics in the graphics window or text in the terminal window. The three fundamental GrADS commands:

1. `open` open or make available to GrADS a data file with either gridded or station data
2. `d` display a GrADS "expression" (e.g., a slice of data)
3. `set` manipulate the "what" "where" and "how" of data display

The GrADS "expression," or what you want to look at, can be as simple as a variable in the data file that was opened, e.g., `d slp` or an arithmetic or GrADS function operation on the data, e.g., `d slp/100` or `d mag(u,v)` where mag is a GrADS intrinsic function.

The "where" of data display is called the "dimension environment" and defines which part, chunk or "hyperslab" of the 4-D geophysical space (lon,lat,level,time) is displayed. The dimension environment is manipulated through the set command and is controlled in either grid coordinates (x,y,z,t or indices) or *world* coordinates (`lon, lat,lev, time`).

The "what" and "how" of display is controlled by `set commands` and includes both graphics methods (e.g., contours, streamlines) and data (e.g., `d` to a file).

GrADS graphics can be written to a file (i.e., `enable print` *filename* and `print`) and then converted to postscript for printing and/or conversion to other image formats.

In addition, GrADS includes graphic primitives (e.g., lines and circles) and basic labelling through the `draw command`.

The `q` or `query` command is used to get information from GrADS such as which files are opened and even statistics.

About GrADS Gridded Data Sets

This section describes GrADS gridded data sets -- their structure and format, how to create them, and how to instruct GrADS to interpret them properly. Here are some quick links for skipping through this section:

Introduction

In GrADS, the raw binary data and the meta data (information about the binary data) are stored in separate files. The meta data file contains a complete description of the binary data as well as instructions for GrADS on where to find the data and how to read it. The binary data file is purely data with no space or time identifiers. The meta data file is the one you open in GrADS -- it is called the **data descriptor file**. The data descriptor file has a `.ctl` extension and is therefore also referred to as a **control file**.

ga-> `open` *filename*.ctl

The Data Descriptor File

The data descriptor file contains a complete description of the binary data as well as instructions for GrADS on where to find the data and how to read it. The descriptor file is an ascii file that can be created easily with a text editor. The general contents of a gridded data descriptor file are as follows:

- Filename for the binary data
- Missing or undefined data value
- Mapping between grid coordinates and world coordinates
- Description of variables in the binary data set

The individual components of data descriptor files are discussed in detail in the section Elements of a Data Descriptor File.

The data descriptor file is free format, which means the components of each record (line

of text) are blank delimited. The only exceptions are comment records which must start with an asterisk (*) in the first column and may not appear in the list of variable descriptor records between the VARS and ENDVARS records. Individual records may not be more than 255 characters long. Here is an example of a basic data descriptor file:

```
DSET  ^gridded_data_sample.dat
TITLE Gridded Data Sample
UNDEF -9.99E33
XDEF 180 LINEAR 0.0  2.0
YDEF  90 LINEAR -90  2.0
ZDEF  10 LEVELS 1000 850 700 500 400 300 250 200 150 100
TDEF   4 LINEAR 0Z10apr1991 12hr
VARS     4
slp     0  99  sea level pressure
hgt    10  99  heights
temp   10  99  temperature
shum    6  99  specific humidity
ENDVARS
```

In this example, the binary data set is named gridded_data_sample.dat and is located in the same directory as the descriptor file. This is specified by the caret (^) in front of the data filename. The undefined or missing data value is -9.99e33, there are 180 grid points in the X direction, 90 grid points in the Y direction, 10 vertical levels, 4 time steps, and 4 variables. The variable "slp" is a surface variable -- it has no vertical levels, but is assigned a default vertical coordinate of Z=1. The variables "hgt" and "temp" have 10 vertical levels, and the variable "shum" has 6 vertical levels (the first six listed, 1000 to 300).

Structure of a Gridded Binary Data File

The binary data file is purely data with no space or time identifiers. The data descriptor specifies the data's grid dimensions, but it is up to the user to make sure that the binary data have been written to file in the proper order so GrADS will interpret them correctly.

GrADS views gridded data sets as 5-dimensional arrays varying in longitude, latitude, vertical level, variable, and time. It is helpful to think of a gridded binary data file as a sequence of "building blocks", where each building block is a horizonal grid of data varying in the X and Y dimensions. The first dimension (X) always varies from west to east; the second dimension (Y) varies from south to north (by default). One horizontal grid represents a particular variable at a particular height and time.

Each horizontal grid in a GrADS binary data file must be the same size. If you have two variables with different horizontal grids, you must create two separate data sets.

The structure of a 3-D, 4-D, or 5-D data set is determined by the order in which the horizonal grids are written to file. The building blocks are stacked in a sequence according to dimension. The sequence goes in the following order starting from the fastest varying dimension to the slowest varying dimension: longitude (X), latitude (Y), vertical level (Z), variable (VAR), time (T).

For example, suppose you want to create a 4-D binary data set containing four variables. The horizonal grids would be written to the data set in the following order:

```
Time 1, Variable 1 , Each vertical level from bottom to top
Time 1, Variable 2 , Each vertical level from bottom to top
Time 1, Variable 3 , Each vertical level from bottom to top
Time 1, Variable 4 , Each vertical level from bottom to top

Time 2, Variable 1 , Each vertical level from bottom to top
Time 2, Variable 2 , Each vertical level from bottom to top
Time 2, Variable 3 , Each vertical level from bottom to top
Time 2, Variable 4 , Each vertical level from bottom to top

etc.
```

Binary Formats

GrADS can read binary data that are formatted with or without FORTRAN record length headers. Files containing record length headers are called "sequential" and those without embedded record length information are called "direct access" or "stream" files. Unless otherwise specified, GrADS will assume the data file does not contain the record length headers.

GrADS can also directly read GRIB formatted data -- one of GrADS most powerful and unique features! See the section on Creating Data Descriptor Files for GRIB Data for more information.

A third category of data formats that GrADS can read are "self-describing files" such as NetCDF or HDF-SDS. For more information, see the references pages for sdfopen and xdfopen.

Creating Data Files

The default format for GrADS gridded binary data files is "stream" or "direct access". If you want to read FORTRAN "sequential" unformatted binary data files, you must include the following additional record in the data descriptor file:

```
OPTIONS sequential
```

Following are three examples of how to create gridded binary data files with simple FORTRAN programs.

1. Suppose you have U and V wind components in 4-dimensions (X, Y, Z, and T) and you want to write them out in so they can be viewed in GrADS. The FORTRAN code might look something like this:

```
parameter (ni=144,nj=91,nk=8,nt=4)
dimension u(ni,nj,nk),v(ni,nj,nk),dum(ni,nj)
do n=1,nk
   call load(u,ni,nj,nk,n,dum)
   write(10) dum
end do
do n=1,nk
   call load(v,ni,nj,nk,n,dum)
   write(10) dum
end do

subroutine load(a,ni,nj,nk,n,dum)
dimension a(ni,nj,nk),dum(ni,nj)
do i=1,ni
   do j=1,nj
   dum(i,j)=a(i,j,n)
end do
end do
return
```

   The data descriptor file would look something like:

```
DSET      ^model.dat
TITLE     Sample Model Data
UNDEF    0.10000E+16
XDEF     144 linear   0 2.5
YDEF      91 linear -90 2.0
ZDEF       8 levels 1000 900 800 700 500 300 100 50
TDEF       4 linear 00z01apr85 6hr
VARS      2
   u 8 99 U component
   v 8 99 V component
ENDVARS
```

2. This simple example write out one variable:

```
   REAL  Z(72,46,16)
   ....
   OPEN(8,FILE='grads.dat',FORM='UNFORMATTED',
 & ACCESS='DIRECT',RECL=72*46)
```

```
   ....
   IREC=1
   DO 10 I=1,16
     WRITE (8,REC=IREC) ((Z(J,K,I),J=1,72),K=1,46)
     IREC=IREC+1
10 CONTINUE
```

3. Another simple sample might be:

```
   REAL X(100)
   DO 10 I=1,100
     X(I)=I
10 CONTINUE
   OPEN (8,FILE='samp.dat',FORM='UNFORMATTED',ACCESS='DIRECT',
  &RECL=100)
   WRITE (8,REC=1) X
   STOP
   END
```

   The associated descriptor file:

```
DSET      samp.dat
TITLE     Sample Data Set
UNDEF    -9.99E33
XDEF     100 LINEAR 1 1
YDEF     1 LINEAR 1 1
ZDEF      1 LINEAR 1 1
TDEF      1 LINEAR 1JAN2000 1DY
VARS      1
x  0  99  100 Data Points
ENDVARS
```

   Once created, you can use this data set to experiment with GrADS data functions, such as:

```
display sin(x/50)
```

Components of a GrADS Data Descriptor File

DSET
DTYPE
INDEX
STNMAP
TITLE
UNDEF
UNPACK
FILEHEADER
THEADER
XYHEADER
XVAR
YVAR
ZVAR
STID
TVAR
TOFFVAR
OPTIONS
PDEF
XDEF
YDEF
ZDEF
TDEF
VARS
ENDVARS

DSET *data_filename* back to top

This entry specifies the filename of the data file being described. If the data and the descriptor file are not in the same directory, then *data_filename* must include a full path. If a ^ character is placed in front of *data_filename*, then *data_filename* is assumed to be relative to the path of the descriptor file. If *data_filename* does not include a full path or a ^, then GrADS assumes the data set and its descriptor file are in the same directory. If you are using the ^ character in the DSET entry, then the descriptor file and the data file may be moved to a new directory without changing any entries in the data descriptor file, provided their relative paths remain the same. For example:

If the data descriptor file is:
    /data/wx/grads/sa.ctl
and the binary data file is:
    /data/wx/grads/sa.dat
then the data file name in the data descriptor file can be:
    DSET ^sa.dat
instead of:
    DSET /data/wx/grads/sa.dat

DTYPE *keyword* back to top

The DTYPE entry specifies the type of data being described. There are four options: grib, hdfsds, netcdf, or station. If the data type is none of these, then the DTYPE entry is omitted completely from the descriptor file and GrADS will assume the data type is gridded binary.

bufr	(GrADS version 1.9) Data file is a BUFR station data file. This data type must be accompanied by the following special entries: XVAR, YVAR, TVAR, STID. Optional special entries are: ZVAR, TOFFVAR.
grib	Data file is an indexed GRIB file. This data type requires a secondary entry in the descriptor file: INDEX. The INDEX entry provides the filename (including the full path or a ^) for the grib index file. The index file is created by the gribmap utility. You must run gribmap and create the index file before you can display the grib data in GrADS.
hdfsds	(GrADS version 1.9) Data file is an HDF Scientific Data Set (SDS). Although HDF-SDS files are self-describing and may be read automatically using the sdfopen/xdfopen commands, this DTYPE gives you the option of overriding the file's own metadata and creating a descriptor file for some or all of the variables in the file. This DTYPE may also be used if the metadata in the HDF-SDS file is insufficient or is not coards-compliant. This data type requires a special entry in the *units* field of the variable declaration. The undef and unpack entries contain special options for this dtype.
netcdf	(GrADS version 1.9) Data file is NetCDF. Although NetCDF files are self-describing and may be read automatically using the sdfopen/xdfopen commands, this DTYPE gives you the option of overriding the file's own metadata and creating a descriptor file for some or all of the variables in the file. This DTYPE may also be used if the metadata in the NetCDF file is insufficient or is not coards-compliant. This data type requires a special entry in the *units* field of the variable declaration. The undef and unpack entries contain special options for this dtype.
station	Data file is in GrADS station data format. This data type requires a secondary entry in the descriptor file: STNMAP. The STNMAP entry provides the filename (including the full path or a ^) for the station data map file. The map file is created by the stnmap utility. You must run stnmap and create the map file before you can display the station data in GrADS.

INDEX *filename* back to top

This entry specifies the name of the grib map file. It is required when using the DTYPE grib entry to read grib formatted data. The file is generated by running the external utility gribmap. Filenaming conventions are the same as those described for the DSET entry.

STNMAP *filename* back to top

This entry specifies the name of the station map file. It is required when using the DTYPE station entry to read GrADS-formatted station data. The file is generated by running the external utility stnmap. Filenaming conventions are the same as those described for the DSET entry.

TITLE *string* back to top

This entry gives brief description of the contents of the data set. *String* will be included in the output from a query command and it will appear in the directory listing if you are serving this data file with the GrADS-DODS Server (GDS), so it is helpful to put meaningful information in the title field. For GDS use, do not use double quotation marks (") in the title.

UNDEF *value <undef_attribute_name>* back to top

This entry specifies the undefined or missing data value. UNDEF is a *required entry* even if there are no undefined data. GrADS operations and graphics routines will ignore data with this value from this data set.
(GrADS version 1.9) An optional second argument has been added for data sets of DTYPE netcdf or hdfsds -- it is the name of the attribute that contains the undefined value. This is used when individual variables in the data file have different missing values. Attribute names are case sensitive, and it is assumed that the name is identical for all variables in the NetCDF or HDF-SDS data file. After data I/O, the missing values in the grid are converted from the individual undef to the file-wide undef (the numerical value in the first argument of the undef record). Then it appears to GrADS that all variables have the same undef, even if they don't in the original data file. If the name given does not match any attributes, or if no name is given, the file-wide undef value will be used.
Example: UNDEF 1e+33 _FillValue

UNPACK *scale_factor_attribute_name add_offset_attribute_name* back to top

(GrADS version 1.9) This entry is used with DTYPE netcdf or hdfsds for data variables that are 'packed' -- i.e. non-float data that need to be converted to float by applying the following formula:

y = x * *scale_factor* + *add_offset*

Two arguments are required, the first is the attribute name for the scale factor (e.g. scale_factor, Slope), the second is the attribute nme for the offset (e.g. add_offset, Intercept). Attribute names are case sensitive, and it is assumed that the names are identical for all variables in the NetCDF or HDF-SDS data file. If the names given do not match any attributes, the scale factor will be assigned a value of 1.0 and the offset will be assigned a value of 0.0.
Example: UNPACK scale_factor add_offset

FILEHEADER *length* back to top

This optional entry tells GrADS that your data file has a header record of *length* bytes that precedes the data. GrADS will skip past this header, then treat the remaineder of the file as though it were a normal GrADS binary file after that point. This optional descriptor file entry is only valid for GrADS gridded data sets.

THEADER *length* back to top

This optional entry tells GrADS that the data file has a header record of *length* bytes preceding each time block of binary data. This optional descriptor file entry is only valid for GrADS gridded data sets. See the section on structure of a gridded binary data file for more information.

XYHEADER *length* back to top

This optional entry tells GrADS that the data file has a header record of length bytes preceding each horizontal grid (XY block) of binary data. This optional descriptor file entry is only valid for GrADS gridded data sets. See the section on structure of a gridded binary data file for more information.

XVAR *x,y* back to top

(GrADS version 1.9) This entry provides the x,y pair for the station's longitude. This entry is required for DTYPE bufr.

YVAR *x,y* back to top

(GrADS version 1.9) This entry provides the x,y pair for the station's latitude. This entry is required for DTYPE bufr.

ZVAR *x,y* back to top

(GrADS version 1.9) This entry provides the x,y pair for the station data's vertical coordinate (e.g., pressure). This is an optional entry for DTYPE bufr.

STID *x,y* back to top

(GrADS version 1.9) This entry provides the x,y pair for the station ID. This entry is required for DTYPE bufr.

TVAR yr *x,y* mo *x,y* dy *x,y* hr *x,y* mn *x,y* sc *x,y* back to top

(GrADS version 1.9) This entry provides the x,y pairs for all the **base time** coordinate variables. Each time unit (year=yr, month=mo, day=dy, hour=hr, minute=mn, second=sc) is presented as a 2-letter abbreviation followed by the x,y pair that goes with that time unit. The time for any individual station report is the base time plus the offset time (see TOFFVAR). All six base time units are not required to appear in the TVAR record, only those that are in the data file. This entry is required for DTYPE bufr.

TOFFVAR yr *x,y* mo *x,y* dy *x,y* hr *x,y* mn *x,y* sc *x,y* back to top

(GrADS version 1.9) This entry provides the x,y pairs for all the **offset time** coordinate variables. The syntax is the same as TVAR. The time for any individual station report is the base time plus the offset time. All six offset time units are not required to appear in the TOFFVAR record, only those that are in the data file. This is an optional entry for DTYPE bufr.

OPTIONS *keyword* back to top

This entry controls various aspects of the way GrADS interprets the raw data file. It replaces the old FORMAT record. The *keyword* argument may be one or more of the following:

yrev	Indicates that the Y dimension (latitude) in the data file has been written in the reverse order from what GrADS assumes. An important thing to remember is that GrADS still presents the view that the data goes from south to north. The YDEF statement does not change; it still describes the transformation from a grid space going from south to north. The reversal of the Y axis is done as the data is read from the data file.
zrev	Indicates that the Z dimension (pressure) in the data file has been written from top to bottom, rather than from bottom to top as GrADS assumes. The same considerations as noted above for yrev also apply.
template	Indicates that a template for multiple data files is in use. For more information, see the section on Using Templates.
sequential	Indicates that the file was written in sequential unformatted I/O. This keyword may be used with either station or gridded data. If your gridded data is written in sequential format, then each record must be an X-Y varying grid. If you have only one X and one Y dimension in your file, then each record in the file will be one element long (it may not be a good idea to write the file this way).
365_day_calendar	Indicates the data file was created with perpetual 365-day years, with no leap years. This is used for some types of model ouput.
byteswapped	Indicates the binary data file is in reverse byte order from the normal byte order of your machine. Putting this keyword in the OPTIONS record of the descriptor file tells GrADS to swap the byte order as the data is being read. May be used with gridded or station data.

The best way to ensure hardware independence for gridded data is to specify the data's source platform. This facilitates moving data files and their descriptor files between machines; the data may be used on any type of hardware without having to worry about byte ordering. The following three OPTIONS keywords are used to describe the byte ordering of a gridded or station data file:

big_endian	Indicates the data file contains 32-bit IEEE floats created on a big endian platform (e.g., sun, sgi)
little_endian	Indicates the data file contains 32-bit IEEE floats created on a little endian platform (e.g., iX86, and dec)
cray_32bit_ieee	Indicates the data file contains 32-bit IEEE floats created on a cray.

PDEF

back to top

Display Pre-Projected Data with PDEF

Gridded data that are mapped onto a particular map projection are called 'pre-projected.' An example of pre-projected data is the output from a weather forecast model that is mapped onto a north polar stereographic grid.

In order to display pre-projected data on a map in GrADS, the descriptor file must contain a PDEF entry. A descriptor file that contains a PDEF record describes two different grids. The first grid is described by the PDEF record itself and is the "native" grid for the pre-projected data in the file. The PDEF record describes the size of this native grid, and then describes how to convert from lat/lon to the i/j of the grid (this description takes the form of stating the projection of the grid along with the projection constants or providing the mapping to the native grid in a supplementary data file). The second grid described in the desctiptor file is a rectilinear lat/lon grid, which is defined by the XDEF and YDEF records. The rectilinear grid is used by GrADS internally and can be any size or resolution -- it is completely independent of the pre-projected or native grid. GrADS uses the information about the two grids to interpolate from the PDEF-described native grid to the XDEF/ YDEF-described rectilinear grid. All displays and analyses are done using the interpolated data on the rectilinear grid. The virtue of this approach is that all built in GrADS analytic functions (e.g., aave, hcurl...) will work even though the data were not originally on a lon/lat grid. The downside is that you are looking at interpolated data.

It is possible to view the pre-projected data on its native grid. To do this, you omit the PDEF entry from the descriptor file, and use the XDEF and YDEF entries to describe the shape of the native grid. In this case, your displays must be drawn in i/j space without a map projection (set mpdraw off).

When you do a display of a pre-projected vector field, you must know whether the original vector components are defined relative to the data grid or relative to the Earth. If the data are grid-relative, they must be rotated to Earth-relative coordinates in order for the interpolation to work properly. See the "Notes" under each particular projection type for further information.

PDEF Syntax

PDEF *isize jsize* **NPS** *ipole jpole lonref gridinc*
PDEF *isize jsize* **SPS** *ipole jpole lonref gridinc*

Example:PDEF 53 45 nps 27 49 -105 190.5

Args:	*isize*	The size of the native grid in the x direction
	jsize	The size of the native grid in the y direction
	ipole	the i coordinate of the pole referenced from the lower left corner, assumed to be at (1,1)
	jpole	the j coordinate of the pole referenced from the lower left corner, assumed to be at (1,1)
	lonref	reference longitude
	gridinc	distance between gripoints in km

Notes: Polar stereographic projections (N and S) are defined as at NCEP. Wind rotation has also been added so that vector data will be properly displayed.

PDEF *isize jsize* **LCC** *latref lonref iref jref Struelat Ntruelat slon dx dy*

Example:PDEF 103 69 lcc 30 -88 51.5 34.5 20 40 -88 90000 90000

Args:		
	isize	The size of the native grid in the x direction
	jsize	The size of the native grid in the y direction
	latref	reference latitude
	lonref	reference longitude (in degrees, E is positive, W is negative)
	iref	i of ref point
	jref	j of ref point
	Struelat	S true lat
	Ntruelat	N true lat
	slon	standard longitude
	dx	grid X increment in meters
	dy	grid Y increment in meters

Notes: The Lambert Conformal projection (lcc) was implemented for use with data from the U.S. Navy's limited area model NORAPS.*Wind rotation has not been implemented!!! Use only for scalar fields.*

PDEF *isize jsize* **ETA.U** *lonref latref dlon dlat*

Example:PDEF 181 136 eta.u -97.0 41.0 0.38888888 0.37

Args:		
	isize	The size of the native grid in the x direction
	jsize	The size of the native grid in the y direction
	lonref	reference longitude (in degrees, E is positive, W is negative)
	latref	reference latitude
	dlon	grid longitude increment in degrees
	dlat	grid latitude increment in degrees

Notes: The eta model native grid is awkward to work with because the variables are on staggered *and* non-rectangular grids. NCEP created "unstaggered" eta model fields, in which the variables are placed on a common rectangular grid. Wind rotation has also been added so that vector data will be properly displayed.

PDEF *isize jsize* **PSE** *slat slon ipole jpole dx dy sign*

Example:

Args:	*isize*	The size of the native grid in the x direction
	jsize	The size of the native grid in the y direction
	slat	absolute value of the standard latitude
	slon	absolute value of the standard longitude
	ipole	the i coordinate of the pole referenced from the lower left corner, assumed to be at (0,0)
	jpole	the j coordinate of the pole referenced from the lower left corner, assumed to be at (0,0)
	dx	grid X increment in km
	dy	grid Y increment in km
	sign	1 for NH; -1 for SH

Notes: The polar stereo projection used by the original NMC models is not very precise because it assumes the earth is round (eccentricity = 0). While this approximation was reasonable for coarse resolution NWP models, it is inadequate to work with higher resolution data such as SSM/I. *Wind rotation has not been implemented!!! Use only for scalar fields.*

PDEF *isize jsize* **OPS** *latref lonref xoff yoff iref jref dx dy*

Example:PDEF 26 16 ops 40.0 -100.0 90000.0 90000.0 14.0 9.0 180000.0 180000.0

Args:	*isize*	The size of the native grid in the x direction
	jsize	The size of the native grid in the y direction
	latref	reference latitude
	lonref	reference longitude (in degrees, E is positive, W is negative)
	xoff	lonref offset in meters
	yoff	latref offset in meters
	iref	the i coordinate of the reference point
	jref	the j coordinate of the reference point
	dx	grid X increment in km
	dy	grid Y increment in km
	dy	grid Y increment in km

Notes: The CSU RAMS model uses an oblique polar stereo projection. *Wind rotation has not been implemented!!! Use only for scalar fields.*

PDEF *isize jsize* **BILIN** *format byteorder fname*

Example:PDEF 100 100 BILIN sequential binary-big ^mygrid.interp.values

Args:	*isize*	The size of the native grid in the x direction
	jsize	The size of the native grid in the y direction
	format	Must be either STREAM (direct access) or SEQENTIAL (fortran formatted)
	byteorde r	If set to BINARY, byte odering is assumed to be same as local machine If set to BINARY-BIG, byte ordering is assumed to be big-endian If set to BINARY-LITTLE, byte ordering is assumed to be little-endian
	fname	The name of the supplementary file
Notes:	The supplementary file contains three lat-lon floating-point grids: i values, j values, and wind rotation values. The size of these grids must match the XDEF and YDEF entries in the descriptor file.	

PDEF *size* **1 FILE** *num format byteorder fname*

Example: PDEF 5000 1 file 4 sequential binary-big ^mygrid.interp.values
PDEF 15238 1 file 1 stream binary ^gtd.filepdef

Args:	*size*	size of the native grid, expressed as a vector (eg, all gridpoints in an x-y grid)
	num	number of sets of interpolation grids supplied
	format	Must be either STREAM (direct access) or SEQENTIAL (fortran formatted)
	byteorde r	If set to BINARY, byte odering is assumed to be same as local machine If set to BINARY-BIG, byte ordering is assumed to be big-endian If set to BINARY-LITTLE, byte ordering is assumed to be little-endian
	fname	The name of the supplementary file
Notes:	See below	

How PDEF Grid Interpolation Works

To illustrate how the data is interpolated from the native grid to the rectilinear grid, let's consider an example. Here are a set of relevant records from a descriptor file:

PDEF 100 100 nps ...
XDEF 181 linear -180 1
YDEF 90 linear 0 1

These three entries describe data on a native 100x100 North Polar stereographic projection and a rectilinear lat/lon grid that is 181 by 90 and has an interval of 1 degree in both lat and lon. Consider one point within the rectilinear grid, the point -90,40. GrADS calls an internal routine to calculate the i and j values in the native grid that correspond to this lat/lon point. Let's say we get i,j values of 31.24 and 67.88. To do the interpolation to the lat/lon point -90,40, GrADS uses the data values from the following four native grid points: 31,67 - 31,68 - 32,67 - 32,68. Bi-linear interpolation is used within this grid box to get down to the position 31.24,67.88. The interpolation is linear within the i,j grid. When a descriptor file is opened that contains a PDEF record, GrADS calculates the i/j values in the native grid that correspond to the lat/lon pair for each gridpoint in the rectilinear grid.

How PDEF Wind Rotation Works

There is a third value calculated for every lat/lon point, and that is the wind rotation value. With some "pre-projected" or native grids, the winds are given relative to the i/j space and not the lat/lon space. To insure correct interpolation, the winds must be rotated to lat/lon space. This is done by determining a rotation amount for each lat/lon point. When u or v wind components are displayed, the values are not just interpolated but also rotated.

To do the wind rotation properly, GrADS requires both the u and v components. Even if you are just displaying u, GrADS has to retrieve (internally) both the u and v component in order to do the rotation calculation. GrADS determines how to match u and v variables by checking the *units* field of the variable record in the descriptor file. The u variable must have a *units* value of 33, and the v variable must have a *units* value of 34. (This is the GRIB convention). If there are more than one u/v pairs, secondary *units* values are used. For example:

u	18	33,100	U-Wind Components on Pressure Levels
v	18	34,100	V-Wind Components on Pressure Levels
u10	0	33,105	10 Meter U Wind
v10	0	34,105	10 Meter V Wind

might be some variable records in the descriptor file. If wind rotation is called for, u and v would be paired, and u10 and v10 would be paired (since the secondary values would be checked, ie, the 105,100 values).

The PDEF BILIN Option

When a descriptor file is opened that contains a PDEF record, we have explained that GrADS internally generates three grids, each one the size of the rectilinear lat/lon grid. The first two grids contain the i and j values (respectively) from the native grid that correspond to each grid point in the rectilinear grid; the third grid contains wind rotation values. But this only works for a small set of well-defined native grids. GrADS will generate these three internal grids automatically for polar stereographic, lamber conformal, and some eta grids. If the native grid for your data is not one of the predefined projections, it is still possible for GrADS to handle the data. All you have to do is supply these three grids to GrADS with a supplementary data file and use the bilin option in your PDEF record.

The supplementary file will contains three lat-lon floating-point grids sized according to the XDEF and YDEF records in the descriptor file.The three grids contain: i values, j values, and wind rotation values. A value of -999 in the i-value grid indicates not to interpolate to that lat-lon point (will end up missing on output) and a value of -999 in the wind-rotation grid indicates not to do wind rotation for that point. If the wind-rotation grid is all -999 values, no rotation is ever done and a flag is set not to even attempt rotation.

The PDEF FILE Option

All of the PDEF examples discussed so far involve the same method for grid interpolation: a grid point value in the rectilinear grid is calculated by finding the four

neighboring grid points in the native grid and averaging them, with weights applied bi-linearly according to their proximity to the to rectilinear grid point. The PDEF FILE option generalizes this method so that an arbitrary number of native grid point values and their weights are averaged to generate the interpolated rectilinear grid point values. The index values for the native grid values that are to be used and their weights are specified by the user in a supplementary data file (*fname*).

The *num* argument in the PDEF FILE entry specifies the number of native grid points that will be used to calcuate each interpolated rectilinear grid point value. For each *num,* the supplementary data file will contain two grids -- both will be the size of the rectilinear grid (as defined by XDEF and YDEF). The first grid contains the index or offset values that point to the native grid value that will be used in the interpolation; the second grid contains the weights for those native grid values. The first grid contains integer values, the second grid contains floating-point values. Finally, the supplementary data file must also contain one grid of floating-point wind rotation values. Thus if *num* equals 1, there will be 3 grids in *fname*. If *num* equals 3, there will be 7 grids in *fname* (3 sets of 2 grids plus the wind rotation grid).

To do the grid interpolation, GrADS will first read the data in the native grid (vector) along with the values in the supplementary grids. To calculate the interpolated value for a particular lat-lon point, GrADS will get *num* native grid point values, multiply each by their weight, sum over all the weighted native grid points, and divide by the sum of the weights.

Native grid offset values of -999 indicate not to use an input point for that portion of the calculation (thus you can apply less than the "num" number of interpolation points for some of the points). So in theory this can be used for such things as thinned grids, staggered grids, etc.

An Example:

The original data are set up as a vector of land points only, taken from a 1-degree lat/lon grid. There are 15238 land points in the native grid (vector). We use the PDEF FILE option to repopulate a rectilinear lat/lon grid with the land values, leaving the ocean grid points as missing. In this case, ther eis no interpolation done. The PDEF option is used simply to convert a vector of land points into a 2D grid with ocean points missing. The descriptor file looks like this:

```
DSET ^gswp_vegetation_parameters.nc
DTYPE netcdf
TITLE Monthly Vegetation parameters at 1x1 degree
UNDEF 1.e+20
PDEF 15238 1 file 1 stream binary ^gswp.filepdef
XDEF 360 linear -179.5 1
YDEF 150 linear -59.5 1
ZDEF 1 linear 1 1
TDEF 204 linear 00Z01jan1982 1mo
VARS 1
NDVI=>ndvi 0 t,x Monthly vegetation index
ENDVARS
```

The supplementary file gtd.filepdef contains three grids -- the first contains the index values that associate each location in the lat/lon grid with it's point in the vector. All of the ocean points will have a missing value of -999. The second grid will contain the weights, which will be 1 for land points, 0 for ocean points. The third grid will contain all missing values since wind rotation is not a issue in this example. Here is a descriptor file for the supplementary data file (a useful strategy for making sure you've got everything written out correctly):

```
DSET ^gswp.filepdef
TITLE PDEF file for GSWP Vegetation Parameters
UNDEF -999
XDEF 360 linear -179.5 1
YDEF 150 linear -59.5 1
ZDEF 1 linear 1 1
TDEF 1 linear 00z01jul1982 3hr
VARS 3
i 0 -1,40,4 Index Values
w 0 99 Weights
r 0 99 Wind Rotation Values
ENDVARS
```

XDEF *xnum mapping <additional arguments>* back to top

This entry defines the grid point values for the X dimension, or longitude. The first argument, *xnum*, specifies the number of grid points in the X direction. *xnum* must be an integer >= 1. *mapping* defines the method by which longitudes are assigned to X grid points. There are two options for *mapping*:

LINEAR	Linear mapping
LEVELS	Longitudes specified individually

The LINEAR mapping method requires two additional arguments: *start* and *increment*. *start* is a floating point value that indicates the longitude at grid point X=1. Negative values indicate western longitudes. *increment* is the spacing between grid point values, given as a positive floating point value.

The LEVELS mapping method requires one additional argument, *value-list*, which explicitly specifies the longitude value for each grid point. *value-list* should contain *xnum* floating point values. It may continue into the next record in the descriptor file, but note that records may not have more than 255 characters. There must be at least 2 levels in *value-list*; otherwise use the LINEAR method.

Here are some examples:

```
XDEF  144  LINEAR  0.0 2.5
XDEF   72  LINEAR  0.0 5.0
XDEF   12  LEVELS  0 30 60 90 120 150 180 210 240 270 300 330
XDEF   12  LEVELS  15 45 75 105 135 165 195 225 255 285 315 345
```

YDEF *ynum mapping <additional arguments>* back to top

This entry defines the grid point values for the Y dimension, or latitude. The first argument, *ynum*, specifies the number of grid points in the Y direction. *ynum* must be an integer >= 1. *mapping* defines the method by which latitudes are assigned to Y grid points. There are several options for *mapping*:

LINEAR	Linear mapping
LEVELS	Latitudes specified individually
GAUST62	Gaussian T62 latitudes
GAUSR15	Gaussian R15 latitudes
GAUSR20	Gaussian R20 latitudes
GAUSR30	Gaussian R30 latitudes
GAUSR40	Gaussian R40 latitudes

The LINEAR mapping method requires two additional arguments: *start* and *increment*. *start* is a floating point value that indicates the latitude at grid point Y=1. Negative values indicate southern latitides. *increment* is the spacing between grid point values in the Y direction. It is assumed that the Y dimension values go from south to north, so *increment* is always positive.

The LEVELS mapping method requires one additional argument, *value-list*, which explicitly specifies the latitude for each grid point, from south to north. *value-list* should contain *ynum* floating point values. It may continue into the next record in the descriptor file, but note that records may not have more than 255 characters. There must be at least 2 levels in *value-list*; otherwise use the LINEAR method.

The Gaussian mapping methods require one additional argument: start. This argument indicates the first gaussian grid number. If the data span all latitudes, start would be 1, indicating the southernmost gaussian grid latitude.

Here are some examples:

```
YDEF   73  LINEAR   -90 2.5
YDEF  180  LINEAR   -90 1.0
YDEF   18  LEVELS   -85 -75 -65 -55 -45 -35 -25 -15 -5 5 15 25 35 45 55 65 75
                    85
YDEF   94  GAUST62  1
YDEF   20  GAUSR40  15
```

The NCEP/NCAR Reanalysis surface variables are on the GAUST62 grid.

The final example shows that there are 20 Y dimension values which start at Gaussian Latitude 15 (64.10 south) on the Gaussian R40 grid

ZDEF *znum mapping <additional arguments>* back to top

This entry defines the grid point values for the Z dimension. The first argument, *znum*, specifies the number of pressure levels. *znum* must be an integer >= 1. *mapping* defines the method by which longitudes are assigned to Z grid points. There are two options for *mapping*:

LINEAR	Linear mapping
LEVELS	Pressure levels specified individually

The LINEAR mapping method requires two additional arguments: *start* and *increment*. *start* is a floating point value that indicates the longitude at grid point Z=1. *increment* is the spacing between grid point values in the Z direction, or from lower to higher. *increment* may be a negative value.

The LEVELS mapping method requires one additional argument, *value-list*, which explicitly specifies the pressure level for each grid point in ascending order. *value-list* should contain *znum* floating point values. It may continue into the next record in the descriptor file, but note that records may not have more than 255 characters. There must be at least 2 levels in *value-list*; otherwise use the LINEAR method.

Here are some examples:

```
ZDEF   10  LINEAR  1000 -100
ZDEF    7  LEVELS  1000 850 700 500 300 200 100
ZDEF   17  LEVELS  1000 925 850 700 600 500 400 300 250 200 150 100 70
                   50
```

TDEF *tnum* LINEAR *start increment* back to top

This entry defines the grid point values for the T dimension. The first argument, *tnum*, specifies the number of time steps. *tnum* must be an integer >= 1. The method by which times are assigned to T grid points is always LINEAR.

start indicates the initial time value at grid point T=1. *start* must be specified in the GrADS absolute date/time format:

hh:*mm*Z*dd*mmm*yyyy*

where:

hh = hour (two digit integer)
mm = minute (two digit integer)
dd = day (one or two digit integer)
mm = 3-character month
m
yyyy = year (may be a two or four digit integer; 2 digits implies a year between 1950 and 2049)

If not specified, *hh* defaults to 00, *mm* defaults to 00, and *dd* defaults to 1. The month and year must be specified. No intervening blanks are allowed in the GrADS absolute date/time format.

increment is the spacing between grid point values in the T direction. *increment* must be specified in the GrADS absolute time increment format:

vvkk

where:

vv = an integer number, 1 or 2 digits
kk = mn (minute)
hr (hour)
dy (day)
mo (month)
yr (year)

Here are some examples:

```
TDEF   60 LINEAR 00Z31dec1999 1mn
TDEF   73 LINEAR 3jan1989 5dy
TDEF  730 LINEAR 00z1jan1990 12hr
TDEF   12 LINEAR 1jan2000 1mo
TDEF  365 LINEAR 12Z1jan1959 1dy
TDEF   40 LINEAR 1jan1950 1yr
```

back to top

VARS *varnum*
variable_record_1
variable_record_2
...
variable_record_*varnum*
ENDVARS

This ensemble of entries describes all the variables contained in the data set. *varnum* indicates the number of variables in the data set and is therefore also equal to the number of variable records that are listed between the VARS and ENDVARS entries. ENDVARS must be the final line of the Grads data descriptor file. Any blank lines after the ENDVARS statement may cause open to fail!

The format of the variable records is as follows:

varname levs units description

The syntax of *varname* and *units* is different depending on what kind of data format (DTYPE) you are describing. Details provided below:

varname	This is a 1-15 character "name" or abbreviation for the data variable. *varname* may contain alphabetic and numeric characters but it must start with an alphabetic character (a-z).
varname (DTYPE netcdf or hdfsds)	(GrADS version 1.9) For DTYPE netcdf or hdfsds, *varname* may have a different syntax: SDF_varname=>grads_varname SDF_varname is the name the data variable was given when the SDF file was originally created. For NetCDF files, this name appears in the output from ncdump. It is important that SDF_varname exactly matches the variable name in the data file. SDF_varname may contain uppercase letters and non-alpha-numeric characters. The classic *varname* syntax (i.e., when "SDF_varname =>" is omitted) may be used if SDF_varname meets the criteria for GrADS variable names: it must be less than 15 characters, start with an alphabetic character, and cannot contain any upper case letters or non-alpha-numeric characters.

levs	This is an integer that specifies the number of vertical levels the variable contains. *levs* may not exceed *znum* as specified in the ZDEF statement. If *levs* is 0, the variable does not correspond to any vertical level. Surface variables (e.g. sea level pressure) have a *levs* value of 0. For DTYPE station or bufr, surface variables have a *levs* value of 0 and upper air variables have a *levs* value of 1. (Exception to this rule for bufr data: replicated surface variables are given a levs value of 2).
description	This is text description or long name for the variable, max 40 characters.

The *units* component of the variable record is used for data with DTYPE bufr, grib, netcdf, or hdfsds. It is also used for non-standard binary data files that require special "unpacking" instructions, and special cases of pre-projected wind components. If the data you are describing does not fall into any of these categories, put a value of 99 in the *units* field.

units	For flat binary files containing 4-byte floating-point data that are not pre-projected, this field is ignored but must be included. Put in a value of 99.
units (DTYPE bufr)	(GrADS version 1.9) For DTYPE bufr files, this field contains the x,y pair for the named variable.

units (DTYPE grib)

For DTYPE grib, the *units* field specifies the GRIB parameters of the variable. This information is used by the gribmap utility for mapping the variables listed in the descriptor file to the data records in the GRIB files. This parameter may contain up to four comma-delimited numbers:

VV,LTYPE,LEVEL,RI

where,

VV	=The GRIB parameter number (Required)
LTYPE	=The level type indicator (Required)
LEVEL	=The value of the LTYPE (Optional)
RI	=The "range indicator" (for certain level types) (Optional)

The external utilities gribscan and wgrib are quite useful in determining what the values for the *units* field should be for a GRIB data file. Examples:

u	39	33,100	U Winds [m/s]
t	39	11,100	Temperature [K]
ts	0	11,1	Surface Temperature [K]
tb	0	11,116,60,30	Temperature, 30-60mb above surface [K]
dpt	0	17,100,1000	Dew Point Temperature at 1000 mb [K]

units (DTYPE netcdf or hdfsds)

(GrADS version 1.9) For DTYPE netcdf or hdfsds, the *units* field is a comma-delimited list of the varying dimensions of the variable. Dimensions expressed as x, y, z, or t correspond to the four axes defined by XDEF, YDEF, ZDEF and TDEF. For example, a surface variable such as sea level pressure might look like this:

```
presSFC=>psfc   0   y,x   Surface Pressure
```

A time-varying atmospheric variable such as geopotential height might look like this:

```
Height=>hght   17   t,z,y,x   Geopotential Height (m)
```

The order of the dimensions listed in the *units* field does matter. They must describe the shape of the variable as it was written to the SDF data file. For NetCDf files, this information appears in the output from ncdump next to the variable name.

If your data file contains a variable that also varies in a non-world-coordinate dimension (e.g. histogram interval, spectral band, ensemble number) then you can put a non-negative integer in the list of varying dimensions that will become the array index of the extra dimension. For example:

```
VAR=>hist0   0   0,y,x   First historgram interval for VAR
VAR=>hist1   0   1,y,x   Second historgram interval for VAR
VAR=>hist2   0   2,y,x   Third histogram interval for VAR
```

Another option in this example would be to fill the unused Z axis with the histogram intervals:

```
zdef 3 linear 1 1
...
VAR=>hist   0   z,y,x   VAR Histogram
```

In this case, it would appear to GrADS that variable 'hist' varies in Z, but the user would have to remember that the Z levels correspond to histogram intervals. The latter technique makes it easier to slice through the data, but is not the most accurate representation. And if you don't have an unsued world-coordinate axis available, then you still have a way to access your data.

units (non-standard binary)

For non-standard binary files, the *units* field is used to instruct GrADS how to read binary files that do not conform to the default structure or do not contain 4-byte float data. GrADS assumes the data were written in the following order (starting from the fastest varying dimension to the slowest): longitude (X), latitude (Y), vertical level (Z), variable (VAR), time (T). If your binary data set was created or "packed" according to a different dimension sequence, then you can use the *units* field to tell GrADS exactly how to unpack the data.

For these non-standard binary files, the *units* field is a series of one or more comma-delimited numbers, the first of which is always -1. The syntax is as follows:

-1, *structure* <,arg>

There are four options for *structure*, outlined below. Some of these options have additional attributes which are specified with *arg*.

-1,10,*arg*	This option indicates that "VAR" and "Z" have been transposed in the dimension sequence. The order is: longitude (X), latitude (Y), variable (VAR), vertical level (Z), time(T). Thus, all variables are written out one level at a time. This feature was designed to be used with NASA GCM data in the "phoenix" format. The upper air *prognostic* variables were transposed, but the *diagnostic* variables were not. Thus an *arg* of 1 means the variable has been var-z transposed, and an *arg* of 2 means the variable has not.
-1,20,*arg*	This option indicates that "VAR" and "T" have been transposed in the dimension sequence. The order is: longitude (X), latitude (Y), vertical level (Z), time(T), variable (VAR). Thus, all times for one variable are written out in order followed by all times for the next variable, etc. If your data set is actually a collection of separate files that are aggregated by using a template, then you must use *arg* to tell GrADS how many time steps are contained in each individual file. For example, here are the relevant records from a descriptor file for 10 years of monthly wind and temperature data packaged in 10 separate files (one for each year) with "VAR" and "T" dimensions transposed: DSET ^monthlydata_%y4.dat OPTIONS template

	TDEF 120 linear jan79 1mo VARS 3 u 17 -1,20,12 U-Wind Component v 17 -1,20,12 V-Wind Component t 17 -1,20,12 Temperature ENDVARS
-1,30	This option handles the cruel and unusual case where X and Y dimensions are transposed and the horizontal grids are (lat,lon) as opposed to (lon,lat) data. This option causes GrADS to work very inefficiently. However, it is useful for initial inspection and debugging.
-1,40,*arg*	This option handles non-float data. Data are converted to floats internally after they are read from the binary file. The dimension sequence is assumed to be the default. The secondary *arg* tells GrADS what type of data values are in the binary file: *units* = -1,40,1 = 1-byte unsigned chars (0-255) *units* = -1,40,2 = 2-byte unsigned integers *units* = -1,40,-2 = 2-byte signed integers *units* = -1,40,4 = 4-byte integers

units (pre-projected wind components)	For pre-projected vector component data that require the use of PDEF and rotation, GrADS has to retrieve both the u and v component in order to do the rotation calculation. GrADS determines how to match u and v variables by checking the *units* field of the variable record. The u variable must have a *units* value of 33, and the v variable must have a *units* value of 34. (This is the GRIB convention). If there are more than one u/v pairs, secondary *units* values are used.

Doing GRIB in GrADS

INTRODUCTION

One of the most powerful features of GrADS is its ability to work DIRECTLY with GRIB data. Unfortunately, "doing GRIB in GrADS" is not simple; it requires an understanding of GRIB and how GrADS works this type of data.

This note attempts to provide the required understanding to use GRIB data in GrADS.

WHAT IS GRIB?

GRIB (GRIdded Binary) is an international, public, binary format for the efficient storage of meteorological/oceanographic variables. Typically, GRIB data consists of a sequence of 2-D (`lon,lat`) chunks of a (in most general sense) 4-D variable (e.g., `u comp on the wind = f(lon,lat,level,time)`). The sequence is commonly organized in files containing all variables at a particular time (i.e., `3-D (lon,lat,level) volume`).

THE PROBLEM

The problem for the user is how to "interface" or "display" GRIB file(s) to GrADS. The solution has two components. The first is to see what is in the GRIB file, i.e., variables, grids, times, etc. and the second is to "map" the 2-D GRIB fields to higher dimensional structures available in GrADS.

UP-FRONT LIMITATIONS

There are some limitations on the kinds of GRIB data that can be interfaced/displayed in GrADS:

a) lon,lat grids (NOT lat,lon)
b) simple packing
c) grid point data
d) grids must be contiguous (no blocked or octet grids)

Thus, "thinned" grids (non rectangular) and spectral coefficients are not supported. However, GRIB versions 1 AND 0 are supported (GRIB 0 data must be filtered to GRIB 1 for wgrib (see wgrib) and version 1.7 of GrADS will support such non-rectilinear grids. Further, it IS possible to display "preprojected" GRIB data (e.g., polar stereo fields, see eta.ctl).

THE SOLUTION - PART 1

The first, and relatively straightforward, part of the solution is to find out what is in the GRIB file. I use two utilities:

1. gribscan (comes with GrADS);
2. wgrib

The big virtue of wgrib is that the code is written in ANSI C and runs on all the major platforms from PC's to Cray. Although wgrib was designed to support the NCEP reanalysis project, it has been extended to handle other sources of GRIB data (e.g., ECMWF) and is my tool of choice. If I can't read the data in wgrib then I change the code and feed the changes to Wesley Ebisuzaki, NCEP.

Once you have wgrib running,

```
wgrib ncep.reanl.mo.7901.grb
```

yields,

```
1:0:d=79010100:UGRD:kpds5=33:kpds6=100:kpds7=850:
TR=113:P1=0:P2=6:TimeU=1:850
mb:anl:ave@6hr:NAve=124
2:15852:d=79010100:UGRD:kpds5=33:kpds6=100:kpds7=5
00: TR=113:P1=0:P2=6:TimeU=1:500
mb:anl:ave@6hr:NAve=124
```

```
3:33018:d=79010100:UGRD:kpds5=33:kpds6=100:kpds7=2
00: TR=113:P1=0:P2=6:TimeU=1:200
mb:anl:ave@6hr:NAve=124
4:51498:d=79010100:VGRD:kpds5=34:kpds6=100:kpds7=8
50: TR=113:P1=0:P2=6:TimeU=1:850
mb:anl:ave@6hr:NAve=124
5:66036:d=79010100:VGRD:kpds5=34:kpds6=100:kpds7=5
00: TR=113:P1=0:P2=6:TimeU=1:500
mb:anl:ave@6hr:NAve=124
6:81888:d=79010100:VGRD:kpds5=34:kpds6=100:kpds7=2
00: TR=113:P1=0:P2=6:TimeU=1:200
mb:anl:ave@6hr:NAve=124
7:99054:d=79010100:PRES:kpds5=1:kpds6=102:kpds7=0:
TR=113:P1=0:P2=6:TimeU=1:MSL:anl:ave@6hr:NAve=124
```

So we see 7 fields in the file valid at `00Z1jan1979` (`d=79010100`).

for,

```
wgrib ncep.reanl.mo.7902.grb
```

we find,

```
1:0:d=79020100:UGRD:kpds5=33:kpds6=100:kpds7=850:
TR=113:P1=0:P2=6:TimeU=1:850
mb:anl:ave@6hr:NAve=112
2:15852:d=79020100:UGRD:kpds5=33:kpds6=100:kpds7=5
00: TR=113:P1=0:P2=6:TimeU=1:500
mb:anl:ave@6hr:NAve=112
3:33018:d=79020100:UGRD:kpds5=33:kpds6=100:kpds7=2
00: TR=113:P1=0:P2=6:TimeU=1:200
mb:anl:ave@6hr:NAve=112
4:50184:d=79020100:VGRD:kpds5=34:kpds6=100:kpds7=8
50: TR=113:P1=0:P2=6:TimeU=1:850
mb:anl:ave@6hr:NAve=112
5:64722:d=79020100:VGRD:kpds5=34:kpds6=100:kpds7=5
00: TR=113:P1=0:P2=6:TimeU=1:500
mb:anl:ave@6hr:NAve=112
6:80574:d=79020100:VGRD:kpds5=34:kpds6=100:kpds7=2
00: TR=113:P1=0:P2=6:TimeU=1:200
mb:anl:ave@6hr:NAve=112
7:96426:d=79020100:PRES:kpds5=1:kpds6=102:kpds7=0:
TR=113:P1=0:P2=6:TimeU=1:MSL:anl:ave@6hr:NAve=112
```

or the same fields as before, except they are valid at 00z1feb1979.

To find out about the data grid use,

```
wgrib -V ncep.reanl.mo.7901.grb
```

and for the first record you will find:

```
rec 1:pos 0:date 79020100 UGRD kpds5=33 kpds6=100 kpds7=850
levels=(3,82) grid=2 850 mb anl:ave@6hr:
  timerange 113 P1 0 P2 6  nx 144 ny 73 GDS grid 0
num_in_ave 112 missing 0
  center 7 subcenter 0 process 80
  latlon: lat  90.000000 to -90.000000 by 2.500000
          long 0.000000 to -2.500000 by 2.500000, (144 x
73) scan 0 bdsgrid 1
  min/max data -12.29 16.86  num bits 12  BDS_Ref -1229
DecScale 2 BinScale 0
```

The grid is the NCEP #2 grid (see the Gospel According to John) which is a global 2.5 deg grid. The grid has 144 points in x (lon) and 73 points in y (lat). The (1,1) point is located at 90N and 0E.

THE SOLUTION - PART 2

This is the hard part -- creating a relationship between the sequence of 2-D (lon,lat) GRIB fields in a file(s) and the GrADS, 4-D, external-to-the-data, spatio-temporal data volume (lon,lat,level,time). In GrADS, the GRIB-to-4-D volume relationship is defined by the data descriptor or .ctl file. The actual relationship is created using the GrADS utility gribmap which generates an "index" or "map" between GrADS variables in the .ctl file and the GRIB data.

Before describing the details of gribmap, first consider the unix shell script, grb2ctl.sh, originally written by Wesley which I have adapted to help me out. This script uses wgrib to first create a listing of fields in the GRIB data file and then parses the listing for times, variables and levels for the .ctl file.

See,

http://wesley.wwb.noaa.gov/grib2ctl.html

In our example,

```
grb2ctl.sh ncep.reanl.mo.7901.grb
```

yields to standard output,

```
dset ^ncep.reanl.mo.7901.grb
dtype grib
options yrev
index ^ncep.reanl.mo.7901.grb.gmp
undef -9.99E+33
title ncep.reanl.mo.7901.grb
xdef 144 linear 0 2.5
ydef 73 linear -90 2.5
zdef 3 levels
850 500 200
tdef 1 linear 00Z01jan79 1mo
vars 3
     PRES 0 1 ,102,0 Pressure [Pa]
     UGRD 3 33 ,100 u wind [m/s]
     VGRD 3 34 ,100 v wind [m/s]

endvars
```

How did `grb2ctl.sh` do this? First, only ONE grid was found in the GRIB data file (defined by `dset`) and the script had the grid geometry built in. Second, variables with multiply levels (3-D or `lon,lat,level`) where DEFINED to have a "level indicator" of 100 (more below). Third, there was only ONE time in the file and the script SET the time increment to 1mo.

These conditions will often not be present. Thus, the output from `grb2ctl.sh` will likely have to be "tweaked" by the good 'ol trial and error method. In some cases the `.ctl` file may have to built "by hand" using the output from `wgrib`, so you'll probably have to know a lot about GRIB and how gribmap works in many situations.

The key ingredients in the `.ctl` file are:

- grid geometry (xdef,ydef)
- starting time and time increment (tdef)
- variables and "units" parameter
- variable type - "level" or 3-D (zdef) or "surface" (2-D)

The units parameter specifies the GRIB parameters of the variable in the `.ctl` to

be used by gribmap for match GrADS variables to the fields in the GRIB files. This parameter consists of up to four, comma-delimited numbers:

```
VV,LTYPE,(LEVEL),(TRI)
```

where,

VV - (Required) The GRIB parameter number (33 = u comp of the wind, table 2 in John:section 1 page 27, i.e., GRIB Edition 1 (FM 92))

LTYPE - (Required) The level type indicator (100 = pressure level, in Table 3 in John:section 1 Page 33)

LEVEL - (optional) The value of the LTYPE (LTYPE 102 is mean sea level so LEVEL is 0 for where level is located (1,102,0, 0 is AT mean sea level)

TRI - (optional) The "time range indicator" for special applications (Table 5 in John:section 1 Page 37).

Coming up with the units parameter, the grid geometry and the times is the trick.

Fortunately, gribmap can tell us how well the .ctl mapped the GRIB data to the higher dimensional, GrADS data view. And, more importantly, the gribmap process does NOT depend on how the data are actually ordered in the GRIB file, in either level or variable.

Let's redirect output from grb2ctl.sh to a .ctl file, e.g.,

```
grb2ctl.sh on ncep.reanl.mo.7901.grb >
ncep.reanl.mo.ctl
```

and then run gribmap. To reiterate, the gribmap utility compares each field in the GRIB file to each variable, at each level and for all times in the .ctl file and creates an index file telling GrADS WHERE the fields are (or are not) located in the GRIB data.

With the verbose option on,

```
gribmap -v -i ncep.reanl.mo.ctl
```

we get,

```
Scanning binary GRIB file(s):
 ncep.reanl.mo.7901.grb
!!!!!MATCH: 1  15852  2  1  0  33 100 850 btim:
1979010100:00 tau: 0 dtim: 1979010100:00
!!!!!MATCH: 2  33018  2  1  0  33 100 500 btim:
1979010100:00 tau: 0 dtim: 1979010100:00
!!!!!MATCH: 3  51498  2  1  0  33 100 200 btim:
1979010100:00 tau: 0 dtim: 1979010100:00
!!!!!MATCH: 4  66036  2  1  0  34 100 850 btim:
1979010100:00 tau: 0 dtim: 1979010100:00
!!!!!MATCH: 5  81888  2  1  0  34 100 500 btim:
1979010100:00 tau: 0 dtim: 1979010100:00
!!!!!MATCH: 6  99054  2  1  0  34 100 200 btim:
1979010100:00 tau: 0 dtim: 1979010100:00
!!!!!MATCH: 7 116220  2  1  0   1 102 0   btim:
1979010100:00 tau: 0 dtim: 1979010100:00
Reached EOF
```

We have succeeded!!! Each 2-D in the GRIB file has been mapped to a variable in `ncep.reanl.mo.ctl`. In the case of `UGRD`, a 3-D (`lon,lat,level`) variable which we can be sliced in the vertical with GrADS. However, failure to match will NOT stop GrADS from "working." If the data was NOT there, GrADS will return a grid with "undefined" values on display and this state can actually be tested...

The "tweaking" is done by adjusting the `.ctl` file until we get a `!!!!! MATCH` for each GRIB field in the data file(s). I have added a number of options that finely control the mapping process in gribmap for NCEP. See the GrADS document for details (ftp://sprite.llnl.gov/grads/doc/gadoc151.*).

Finally, let's adjust the `ncep.reanl.mo.ctl` file to take advantage of the file naming convention:

```
dset ^ncep.reanl.mo.%y2%m2.grb
dtype grib
options yrev template
index ^ncep.reanl.mo.gmp
undef -9.99E+33
title ncep.reanl.mo.7901.grb
xdef 144 linear 0 2.5
ydef 73 linear -90 2.5
zdef 3 levels 850 500 200
tdef 2 linear 00Z01jan79 1mo
vars 3
     PRES 0 1 ,102,0 Pressure [Pa]
     UGRD 3 33 ,100 u wind [m/s]
```

```
        VGRD 3 34 ,100 v wind [m/s]
    endvars
```

I have changed the number of times to two and have used the template option. This tells GrADS to locate data in TIME by the file name (`dset ^ncep.reanl.mo.%y2%m2.grb`). Thus, I have two (or more) data files, but only ONE `.ctl` file and I can now work with the 2-D GRIB data as if they were 4-D (`lon,lat,lev,time`) in GrADS. I also changed the name of the index file to be reflective of the now 4-D data structure.

To summarize the process:

- use `wgrib` (or gribscan) to see if the data can be worked in GrADS;
- use `grb2ctl.sh` or the output from `wgrib` to construct a `.ctl` file;
- run gribmap in verbose mode (`-v`) to relate the GRIB data to the 4-D structure in the `.ctl` file, and to see how well the map worked; and
- repeat steps 2) and 3) until you get `!!!! MATCH` from gribmap.

CONCLUSIONS

There's no doubt about it, "Doing GRIB in GrADS" is not very straightforward, but the benefits are, in my opinion, immense for two reasons. First, we have avoided conversion of the GRIB to a form which supports higher dimensional data (e.g., `netCDF`). We've saved disk space and have minimized potential technical errors (every time you touch the data you have an opportunity to screw it up). Second, from a GrADS performance standpoint, GRIB is nearly as fast as other binary formats -- the cost in decompression on the fly is compensated by reduced I/O.

In the end, GRIB-to-GrADS interface gives us the advantages of GRIB (efficient storage, self description and an open, international format) while overcoming the disadvantages of GRIB (2-D data and no means to organize to a higher dimension) via the GrADS 4-D data model. We get the best of both worlds, but only if we can make the `.ctl` file. Hopefully this document will help you do this.

sdfopen

`sdfopen` *filename <template #timesteps>*

Opens a NetCDF or HDF-SDS format file that conforms to the COARDS conventions. The arguments and options are as follows:

filename
: The name of the COARDS-compliant NetCDF or HDF-SDS file.

template
: This optional argument is used when you want to aggregate multiple data files and handle them as if they were one individual file. The individual data files must be identical in all dimensions except time. *template* has a similar structure to the substitution template in a GrADS data descriptor file. See Using Templates for details.

#timesteps
: This argument must be included whenever *template* is invoked. The *#timesteps* is the sum of the timestep counts in all the files to be examined, not the count in any one file.

Usage Notes

1. Any particular data file in the group described by the *template* is automatically accessed as the "time" or "t" settings in GrADS are altered.

2. Use `xdfopen` to open a non-COARDS-compliant self-describing file.

3. For additional information, see The Self-Describing Files (SDF) Interface.

Examples

1. If you had daily U-Wind data in two files, `uwnd.1989.nc` and `uwnd.1990.nc`, you could access them both as one GrADS data set by entering:

   ```
   sdfopen /data/uwnd.1989.nc uwnd.%y4.nc 730
   ```

xdfopen

xdfopen *filename*

GrADS requires a certain amount of metadata in order to understand how to read a NetCDF/HDF-SDS data file, also called a self-describing file (SDF). The sdfopen command assumes all the metadata is internal to the self-describing file, whereas the xdfopen command allows the user to supplement or replace any internal metadata via a data descriptor file. In this way, xdfopen provides access to some self-describing files that do not comply with the COARDS conventions.

filename is the name of the data descriptor file that contains the supplemental metadata. It has a syntax very similar to the regular data descriptor files that are used with the open command. The few differences are noted below:

1. DSET *SDF_filename*

   This is the only required entry. *SDF_filename* may be either the name of a netCDF or HDF-SDS file or a substitution template for a collection of netCDF or HDF-SDS files.

   Other than DSET, the only other data descriptor file entries that are supported are UNDEF, TITLE, XDEF, YDEF, ZDEF, TDEF, OPTIONS, VARS, and ENDVARS. Valid arguments for the OPTIONS entry are: yrev, zrev, template, and 365_day_calendar.

2. XDEF, YDEF, ZDEF, and TDEF:

   Each of these entries requires an additional argument, *SDF_dimension_name*, which comes before all the other arguments. The *SDF_dimension_name* is used to achieve dimension order independence, so it must be a real dimension in the SDF. The *SDF_dimension_name* string may be mixed case and should appear exactly as it is listed in the output from ncdump.

   If the coordinate variables in the SDF file exist and have the required metadata, then *SDF_dimension_name* is the only argument needed for the corresonding variable definition entry (XDEF, YDEF, ZDEF, and TDEF) in the data descriptor file.

3. The first argument ("*varname*") of the variable definition lines that appear between VARS and ENDVARS has a slightly different syntax:

SDF_varname=>*grads_varname*

SDF_varname is the name of the variable as it appears in the output from ncdump. It may be of mixed case and include blanks.

If everything up to and including the "=>" is omitted, then *grads_varname* must be identical to *SDF_varname*. This syntax (when "*SDF_varname*=>" is omitted) will only work properly in GrADS if *SDF_varname* is less than 15 characters and does not contain any upper case letters.

As it was with the coordinate variables, if the data variables in the SDF file have the required metadata, then *SDF_varname*=>*grads_varname* is the only argument needed for the corresonding variable definition entry in the data descriptor file.

4. The order of the variable definition lines between VARS and ENDVARS is not important. Unlike in data descriptor files for the open command, the variables that do not vary in Z do not need to be listed first.

Usage Notes

1. If *filename* contains only the DSET entry, then xdfopen devolves into working just like sdfopen.

2. *filename* does not need to be a full data descriptor file, it only needs to contain whatever metadata the SDF file lacks. Anything not specified in *filename* will be looked for in the file's internal metadata.

3. The *SDF_dimension_name* parameter in the XDEF, YDEF, TDEF, and ZDEF entries and the first parameter of the VARIABLE definition lines are the only parts of the data descriptor file that aren't converted to lower case before they are interpreted.

4. For further information on the COARDS conventions, check out Conventions for the standardization of NetCDF files.

Examples

1. This example shows the data descriptor file that would be required in order to open a self-describing file that is missing much of the required metadata. Below is the sample data descriptor file for the NetCDF file moisture.nc. Follow this link to see output from ncdump for this file.

```
DSET ^moisture.nc
TITLE This is a sample
UNDEF 99999.0
XDEF dimension1 144 LINEAR 0.0 2.5
YDEF dimension2  73 LINEAR 0.0 2.5
TDEF dimension3 365 LINEAR 0Z01JAN1979 1DY
VARS 1
Moisture=>moisture 1 99 Moisture
ENDVARS
```

2. This second example comes from a real-world HDF-SDS file from the Data Assimilation Office at NASA Goddard Space Flight Center. The data descriptor file is shown below, and this link shows the output from running the HDF version of ncdump on DAOE054A.hdf. (Note that the output has been annotated with explanatory comments -- they are preceded with "//")

```
DSET ^DAOE054A.hdf
TITLE This is only a test
OPTIONS YREV
UNDEF 1.0E15
XDEF XDim:DAOgrid 144 LINEAR -180.0 2.5
YDEF YDim:DAOgrid  91 LINEAR  -90.0 2.0
ZDEF HGHT18DIMS:DAOgrid 18 LEVELS 1000 850 700 500 400 300 250
200 150 100 70 50 30 10 5 2 1 0.4
TDEF TIME4DIMS:DAOgrid 4 LINEAR 0Z31JUL1993 6HR
VARS 3
GEOPOTENTIAL_HEIGHT=>hgt 18 99 geopotential height
SPECIFICHUMIDITY=>shum 18 99 specific humidity
TEMPERATURE=>temp 18 99 temperature
ENDVARS
```

Reinitialization of GrADS

Two commands are available for resetting or reinitializing the state of GrADS:

Reinit

The `reinit` command returns GrADS to its initial state. `reinit` closes all open files, releases all defined variables, and resets all graphics settings to their defaults.

Reset

The `reset` command returns GrADS to its initial state with the following exceptions:

1. No files are closed
2. No defined variables are released
3. The `set display` settings are not modified

If files are open, the default file is set to 1, and the dimension environment is set to X,Y varying and Z and T set to 1 (as though file 1 were just opened).

The `reset` command may be qualified so that only certain aspects of GrADS are returned to their initial state. The qualifiers are as follows:

`reset events`	resets the events buffer (e.g., mouse clicks)
`reset graphics`	resets the graphics, but not the widgets
`reset hbuff`	resets the display buffer when in double buffer mode
`reset norset`	resets the X events only

Command line editing and history

If the `readline` library compiles on your system then the default prompt will be `ga->` as opposed to `ga>`. This indicates that command line editing is active. The library defaults to `emacs` mode but can be set up to run using `vi` syntax.

Here's a list of the commands which may typically be used:

`ctrl-a`	go to beginning of line
`ctrl-e`	go to end of line
`ctrl-f`	go forward one char
`ctrl-b`	go backward one char
`ctrl-d`	delete the char
`ctrl-p`	recall previous line
`ctrl-n`	recall next line
`ctrl-r`	reverse search

You also get `file name completion` using the `tab` key. If there is more than one option, then `double tab` will list the available completions.

For example, suppose you are running grads on div40-2 at FNMOC and want to start looking for files to open...

Type `open /h` and get,

`ga-> open /h` and hit two `tabs` and get:

```
h home home1 home2
```

then type `ome1` and `tab tab` and get,

```
ga-> open /home1/
GCC bogus603 gnu iqpops nmcobs roesserd
GRIB cstrey grads lost+found pacek tsai
Mosaic dh hamilton mendhall picardr witt
NEWDBS dolan hout nicholso qcops
```

then type `GR`, `tab` to go to GRIB dir, followed by `d`, `tab` to go to the dat dir and then `n`, `tab tab` gives,

```
ga-> open /home1/GRIB/dat/nogaps.25.
nogaps.25.95021600.grb    nogaps.25.95021912.grb
nogaps.25.95021600.gribmap
```

```
nogaps.25.95021912.gribmap
nogaps.25.95021612.anal.grb    nogaps.25.anal.ctl
nogaps.25.95021612.ctl    nogaps.25.anal.gribmap
nogaps.25.95021612.grb    nogaps.25.ls.mask.ctl
nogaps.25.95021612.gribmap    nogaps.25.ls.mask.dat
nogaps.25.95021700.anal.grb    nogaps.25.95021700.ctl
```

and type `950217` to get

```
ga-> open /home1/GRIB/dat/nogaps.25.950217
nogaps.25.95021700.anal.grb    nogaps.25.95021712.ctl
nogaps.25.95021700.ctl    nogaps.25.95021712.grb
nogaps.25.95021700.grb    nogaps.25.95021712.gribmap
nogaps.25.95021700.gribmap
nogaps.25.95021712.anal.grb
```

and finally open the `12Z` data with 12.c, `tab`, return to open the file

```
nogaps.25.95021712.ctl
```

WARNING

There is no guarantee that these `readline` routines will always work, so the `-h` option has been added to the invocation of GrADS to turn them off.

External Utilities

gribmap

The GrADS data descriptor file defines a grid structure into which the data will fit -- it gives "shape" to the data by identifying its spatial dimensions, the number of time steps, and the number of variables.

If the data is in binary format, its structure has already been prescribed. If the data is in GRIB format, no consistent relationship exists between the data and the grid structure defined in the data descriptor file. Hence, the need for the `gribmap` utility which "maps" between the GRIB data and the GrADS data description.

As `gribmap` reads each field in the GRIB data file, the parameters for that field (e.g. variable name, vertical level, time) are compared to the information in the data descriptor file until a match is found. The process continues until all the GRIB elements have been "mapped" to their location within the GrADS grid structure.

The syntax for the `gribmap` command is as follows:

```
gribmap [-i fname] [-v] [-t0] [-0] [-fhr] [-sxxx] [-hxxx]
```

The options are as follows:

fname
: The name of the data descriptor file. If not specified, `gribmap` will prompt the user.

`-v`
: Produces verbose output to make it easier to verify what is being mapped.

`-t0`

> `gribmap` will only match those grib records whose base time is the same as the initial time in the data descriptor file. This is used to pull out a forecast sequence (0, 12, 24, ... , 72 hours) starting a specific time.

`-0`

> `gribmap` will ignore the forecast time when setting up a match. This is useful for reanalysis data sets in which some diagnostic fields are "valid" at slightly different forecast time even though the share the same starting time.

`-f`*`hr`*

> `gribmap` will only match those grib records whose forecast time is *`hr`* hours. This is used to isolate a sequence of forecasts. For example, if you wanted to sample all the 120-hour forecasts from the MRF ensemble runs, you would use `gribmap -f120`.

`-s`*`xxx`*

> `gribmap` will scan over no more than *`xxx`* bytes when searching for the character string "GRIB" in each header. The default is 1000.

`-h`*`xxx`*

> `gribmap` will skip over *`xxx`* bytes before starting the scan process. If reading a special NMC format of GRIB data, use `-hnmc`.

Another feature was added to map by the GRIB "time-range-indicator" as specified in the `.ctl` file. This was put in for handling NMC reanalysis data where the time-range-indicator distinguishes between monthly mean variances and means.

gribscan

The gribscan utility is used for extracting grid info from GRIB data files. Its features include grid/product information, gridded output in ASCII, binary, and/or grib format, plus automatic "scanning" for GRIB records so that you don't have to know the physical layout of the data to scan it.

The command sytax is:

gribscan [-i *ifname*] [-o *ofname*] [-*file options*] [-*processing options*] [-*display options*]

Where:

ifname

This is the input grib file name. If -i *ifname* is omitted, gribscan will prompt the user for a file name.

ofname

This is the output file name WITHOUT an extension. If -o *ofname* is omitted, a default file name of zy0x1w2.*type* is created where *type* is:
asc - ascii
grb - GRIB
dat - a stream of floats (GrADS format)

File Options:

-og

gribscan will return output in GRIB format

-oa

gribscan will return output in ASCII format (%8g in C-language syntax)

-of

gribscan will return output as a stream of floats. This is machine dependent and is 64-bit on Crays and 32-bit elsewhere.

Processing Options:

`-s`*NNN*

Specifies the max number *NNN* of bytes between GRIB messages in the file. The default is 500 and it is assumed that you want to ignore junk (e.g., comm stuff) between data.

`-sp`*NNN*

Selects parameter # *NNN* (e.g.,`-sp11` for temperature)

`-sl`*NNN*

Selects level # *NNN* (e.g., `-sp500` to get 500 mb fields)

`-st`*NNN*

Selects tau # *NNN* (e.g., `-st12` to get t=12 forecasts)

-hNNN

Specifies a fixed file header of *NNN* bytes. If omitted, the default is to seek the first GRIB message automatically, but if you know *NNN*, it is more efficient to specify it.

Special note to NMC users: The once "standard" 81-byte header in an NMC GRIB file contained the string "GRIB". Unfortunately, the same string is part of the GRIB indicator section itself! Thus, an automatic scan for GRIB to demark the start of the data will fail if the 81-byte header is present! When in doubt (or failure) try using the `-h81` option.

Note: These processing options can be used simultaneously to output a very narrow set of fields.

Display options:

`-q`	Quick output to extract stuff GrADS gribmap cares about
`-q1`	One-line quick output
`-d`	Comma delimited mode
`-v`	Verbose mode for diagnostics
`-bd`	Binary data section info
`-gv`	Uses the NMC GRIB variable table to output mnemonic, title, and units

-gd	Output info from the grid defn sec
-S	Silent mode; NO standard output

Examples

1. A "quick" scan to get the info GrADS cares about:

```
gribscan -q -i eta.T12Z.PGrbF48 | grep 184
```

Gives the result:

```
184,F,135,108,100,0,100,0,1e+09,T,1994,8,29,12,0,1,
48,0,G,104,BDTG,94082912
```

Where:

184	field # in the file
F	field data
135	param #
108	level indicator
100	level
0	l1 byte 1 of level
100	l2 byte 2 of level
0	time range indicator
1e+09	decimal scale factor
T	time data follows
1994	year
8	month
29	day
12	hour
0	min
1	forecast time unit (hour)
48	t=48 h forecast
G	grid param follows
104	NMC grid #104
BDTG	Base date-time-group (yymmddhh) follows

2. Comma delimited output:

```
gribscan -d -i eta.T12Z.PGrbF48 | grep 184
```

Gives the same results as the previous example but arranged differently:

```
PDS,184,104,135,108,100,0,100,1994,8,29,12,0,1,48,0
,0,1e+09
```

3. A full listing:

```
gribscan -d -gv -bd -gd -i eta.T12Z.PGrbF48 | grep
184
```

Gives the following results:

```
PDS,184,104,135,108,100,0,100,1994,8,29,12,0,1,48,
0,0,1e+09,mconv,Horizontal moisture
divergence,[kg/kg/s],GDS,5,147,110,-
139.475,90.755,0.354,-0.268,-
105.000,33536.000,0,1,0,BDS,12,-
646.844,16170,4825059,26366
```

Where:

104	grid id
param #135	mconv,Horizontal moisture divergence,[kg/kg/s]
BDS	binary data section
646.844	ref value 16170 - # of points
4825059	starting byte of the data
26366	length of the grib message

Note that eliminating the `-d` option would result in a fixed-column type output.

4. Output a selected few fields in GRIB:

```
gribscan -og -sp135 -q -i eta.T12Z.PGrbF48 -o
eta.135
```

Writes out all GRIB message containing the 135 parameter to the file `eta.135.grb`. A subsequent execution of `gribscan` on `eta.135.grb` would return:

```
1, F ,135,108,100,0,100,0,1e+09,
T,1994,8,29,12,0,1,48,0, G ,104, BDTG, 94082912 2,
F,135,108,21860,85,100,0,1e+09, T
,1994,8,29,12,0,1,48,0, G ,104, BDTG, 94082912
```

gxeps

`gxeps [ -acdklrsv -i <infile> -o <outfile> ] [<infile>]`

Converts the GrADS metacode format file to a PostScript file. Where:

`-i` *fname*	identifies input GrADS metacode file
`-o` *fname*	identifies output postscript file
`-c`	prints color plot
`-r`	prints on a black background
`-d`	appends CTRL-D to the file, useful if printing on a HP1200C/PS color printer
`-1`	use PostScript Level 1 (default)
`-2`	use PostScript Level 2
`-a`	Page size A4
`-l`	Page size US-Letter
`-L`	Ask for a label to be printed on the plot
`-n`	Ask for a note to include in PostScript file header
`-s`	Add a file and time stamp on the plot.
`-v`	Verbose mode.

Usage Notes

1. The default behaviour of `gxeps`is to create a grayscale plot on a white background. The GrADS default rainbow colors (color numbers 2 to 14) are converted into appropriate grey shades. User-defined colors (numbers above 15) are translated to greyscale intensity based on their *green* content only.
2. For more information, see the section in the User's Guide on Producing Image Output from GrADS.

Examples

1. `gxeps -a`
   Prompts user for the input and output file names and convert to greyscale on white background with page size A4.
2. `gxeps -rc -i mytest.mf -o mytest.ps`
   Converts GrADS metacode format file *`mytest.mf`* to a color plot on black background and outputs the result to PostScript file *`mytest.ps`*.

gxps

`gxps [ -crd -i <infile> -o <outfile> ] ]`

`gxps` is a UNIX utility that converts the GrADS metacode format file to a PostScript file. Command line arguments and switches are:

`-i` *`fname`*	identifies input GrADS metacode file
`-o` *`fname`*	identifies output postscript file
`-c`	prints color plot
`-r`	prints on a black background
`-d`	appends CTRL-D to the file, useful if printing on a HP1200C/PS color printer

Usage Notes

1. The default behaviour of `gxps`is to create a grayscale plot on a white background. The GrADS default rainbow colors (color numbers 2 to 14) are converted into appropriate grey shades. User-defined colors (numbers above 15) are translated to greyscale intensity based on their *green* content only.
2. For more information, see the section in the User's Guide on Producing Image Output from GrADS.

Example

```
gxps -cr -i mytest.mf -o mytest.ps
```

Convert GrADS metacode format file `mytest.mf` to a color plot on a black background and outputs the result to PostScript file `mytest.ps`.

gxtran

`gxtran [ -airg <infile> ]`

This utility is used to display GrADS meta files containing single frames or animations (multi frames).

The options are:

`-i fname`	identifies input GrADS metacode file
`-a`	animates the frames without user hitting return in the command window
`-r`	reverses background/foreground colors
`-g geom`	sets the geometry of the X window as in any other X application.
	geom has the form WWWxHHH+X+Y, giving a display window
	WWW pixels wide and HHH pixels tall starting at point X,Y on the screen.

Usage Notes

Without any options, `gxtran` prompts the user for the name of the GrADS meta file. To quit gxtran, hit return in the command window.

Examples

```
gxtran -a -g 800x600+0 -i mytest.mf
```

This command would open a window of size 800x600 starting at the upper left corner of the screen and animate all frames (plots) in the file `mytest.mf`.

ncdump

ncdump [-c] [-h] [-v *var1,...*] [-b *lang*] [-f *lang*] [-l *len*] [-n *name*] [-d *f_digits*[,*d_digits*]] *file*

Where:

-c

Show the values of *coordinate* variables (variables that are also dimensions) as well as the declarations of all dimensions, variables, and attribute values. Data values of non-coordinate variables are not included in the output. This is the most suitable option to use for a brief look at the structure and contents of a netCDF file.

-h

Show only the *header* information in the output, that is the declarations of dimensions, variables, and attributes but no data values for any variables. The output is identical to using the -c option except that the values of coordinate variables are not included. (At most one of -c or -h options may be present.)

-v *var1,...,varn*

The output will include data values for the specified variables, in addition to the declarations of all dimensions, variables, and attributes. One or more variables must be specified by name in the comma-delimited list following this option. The list must be a single argument to the command, hence cannot contain blanks or other white space characters. The named variables must be valid netCDF variables in the input-file. The default, without this option and in the absence of the -c or -h options, is to include data values for *all* variables in the output.

-b *lang*

A brief annotation in the form of a CDL comment (text beginning with the characters ``//'') will be included in the data section of the output for each `row' of data, to help identify data values for multidimensional variables. If *lang* begins with C or c, then C language conventions will be used (zero-based indices, last dimension varying fastest). If *lang* begins with F or f, then Fortran language conventions will be used (one-based indices, first dimension varying fastest). In either case, the data will be presented in the

same order; only the annotations will differ. This option is useful for browsing through large volumes of multidimensional data.

`-f lang`

Full annotations in the form of trailing CDL comments (text beginning with the characters ``//'') for every data value (except individual characters in character arrays) will be included in the data section. If *lang* begins with `C` or `c`, then C language conventions will be used (zero-based indices, last dimension varying fastest). If *lang* begins with `F` or `f`, then Fortran language conventions will be used (one-based indices, first dimension varying fastest). In either case, the data will be presented in the same order; only the annotations will differ. This option may be useful for piping data into other filters, since each data value appears on a separate line, fully identified.

`-l len`

Changes the default maximum line length (80) used in formatting lists of non-character data values.

`-n name`

CDL requires a name for a netCDF data set, for use by `ncgen -b` in generating a default netCDF file name. By default, `ncdump` constructs this name from the last component of the pathname of the input netCDF file by stripping off any extension it has. Use the `-n` option to specify a different name. Although the output file name used by `ncgen -b` can be specified, it may be wise to have `ncdump` change the default name to avoid inadvertantly overwriting a valuable netCDF file when using `ncdump`, editing the resulting CDL file, and using `ncgen -b` to generate a new netCDF file from the edited CDL file.

`-d float_digits[,double_digits]`

Specifies default number of significant digits to use in displaying floating-point or double precision data values for variables that don't have a `C_format' attribute. Floating-point data will be displayed with *float_digits* significant digits. If *double_digits* is also specified, double-precision values will be displayed with that many significant digits. If a variable has a `C_format' attribute, that overrides any specified floating-point default. In the absence of any `-d`

specifications, floating-point and double- precision data are displayed with 7 and 15 significant digits respectively. CDL files can be made smaller if less precision is required. If both floating-point and double-presision precisions are specified, the two values must appear separated by a comma (no blanks) as a single argument to the command. If you really want every last bit of precision from the netCDF file represented in the CDL file for all possible floating- point values, you will have to specify this with `-d 9,17`.

Usage Notes

`ncdump` generates an ASCII representation of a specified netCDF file on standard output. The ASCII representation is in a form called `CDL` (``network Common Data form Language") that can be viewed, edited, or serve as input to `ncgen`. `ncgen` is a companion program that can generate a binary netCDF file from a `CDL` file. Hence `ncgen` and `ncdump` can be used as inverses to transform the data representation between binary and ASCII representations. See `ncgen` for a description of CDL and netCDF representations.

`ncdump` defines a default format used for each type of netCDF data, but this can be changed if a `C_format' attribute is defined for a netCDF variable. In this case, `ncdump` will use the `C_format' attribute to format each value. For example, if floating-point data for the netCDF variable `Z` is known to be accurate to only three significant digits, it would be appropriate to use the variable attribute

```
Z:C_format = "%.3g"
```

`ncdump` may also be used as a simple browser for netCDF data files, to display the dimension names and sizes; variable names, types, and shapes; attribute names and values; and optionally, the values of data for all variables or selected variables in a netCDF file.

Examples

Look at the structure of the data in the netCDF file `foo.nc`:

```
ncdump -c foo.nc
```

1. Produce an annotated CDL version of the structure and data in the netCDF file `foo.nc`, using C-style indexing for the annotations:

```
ncdump -b c foo.nc > foo.cdl
```

2. Output data for only the variables `uwind` and `vwind` from the netCDF file `foo.nc`, and show the floating-point data with only three significant digits of precision:

```
ncdump -v uwind,vwind -d 3 foo.nc
```

3. Produce a fully-annotated (one data value per line) listing of the data for the variable `omega`, using Fortran conventions for indices, and changing the netCDF dataset name in the resulting CDL file to `omega`:

```
ncdump -v omega -f fortran -n omega foo.nc > Z.cdl
```

ncgen

`ncgen [-b] [-c] [-f] [-n] [-o` *`output_file`*`] `*`input_file`*

Where:

`-b`

Create a (binary) netCDF file. If the `-o` option is absent, a default file name will be constructed from the netCDF name (specified after the netcdf keyword in the input) by appending the `.nc` extension. If a file already exists with the specified name, it will be overwritten.

`-c`

Generate C source code that will create a netCDF file matching the netCDF specification. The C source code is written to standard output.

`-f`

Generate Fortran source code that will create a netCDF file matching the netCDF specification. The Fortran source code is written to standard output.

`-o` *`outputfile`*

Name for the netCDF file created. If this option is specified, it implies the `-b` option. (This option is necessary because netCDF files cannot be written directly to standard output, since standard output is not seekable.)

`-n`

Like `-b` option, except creates netCDF file with the obsolete `.cdf` extension instead of the `.nc` extension, in the absence of an output filename specified by the `-O` option. This option is only supported for backward compatibility.

Examples

1. Check the syntax of the CDL file `foo.cdl`:

```
ncgen foo.cdl
```

2. From the CDL file `foo.cdl`, generate an equivalent binary netCDF file named `x.nc`:

```
ncgen -o x.nc foo.cdl
```

3. From the CDL file `foo.cdl`, generate a C program containing the netCDF function invocations necessary to create an equivalent binary netCDF file named `x.nc`:

```
ncgen -c -o x.nc foo.cdl
```

stnmap

`stnmap [-i fname] [-0]`

`stnmap` is an external GrADS utility that writes out a hash table and/or link list information for station data that allows GrADS to access the data more efficiently. After a station data set has been written and the accompanying data descriptor file has been created, you must run the `stnmap` utility before you can look at the data in GrADS.

The `stnmap` options are as follows:

fname
: The name of the station data descriptor file. If not specified, `stnmap` will prompt the user.

`-0`
: Allows processing of certain templated station data sets without actually reading the data.

The output from `stnmap` goes into a file that is named in the `STNMAP` record of the data descriptor file. (See Usage Note #2).

Usage Notes

1. If you change the data file (perhaps by appending another time group), you will also have to change the descriptor file to reflect the changes and then rerun the `stnmap` utility.

2. Note the difference between required records in a station descriptor file and a grid descriptor file:

   `DTYPE`
   : specifies a data type of: station

   `STNMAP`
   : gives the file name of the station mapping file

   `XDEF, YDEF, ZDEF`
   : these records are not specified in a station data control file

   `TDEF`
   : describes the time grouping interval and the number of the time groups in the file

VARS
: surface variables are listed first, and show a "0" for the number-of-levels field. Level-dependent variables are listed after the surface variables, and show a "1" in the number-of-levels field.

Examples

Here's a sample descriptor file `stat.ctl` for station data set `ua.reps`:

```
DSET   ^ua.reps
DTYPE  station
STNMAP ^ua.map
UNDEF  -999.0
TITLE  Real Time Upper air obs
TDEF   10 linear 12z18jan1992 12hr
VARS   8
  slp 0 99 SLP
  ts  0 99 Temps
  us  0 99 U Winds
  vs  0 99 V Winds
  z   1 99 Heightsa
  t   1 99 Temps
  u   1 99 U Winds
  v   1 99 V WInds
ENDVARS
```

Run the `stnmap` utility:

```
stnmap -i stat.ctl
```

The station map file `ua.map` is a binary file, which includes the hash table and/or link list information.

Analysis Topics

Dimension Environment

The data set is always viewed by GrADS as a generalized 4-D (5-D if you include variables) array located in physical space (lon, lat, lev, time), even if it is in reality a subset of a 4-D space.

The current dimension environment describes what part of the data set you want to work with. Expressions are evaluated with respect to the dimension environment (which allows for simplicity in the expression syntax), and the final display will be determined by the dimension environment. Thus, the dimension environment is a GrADS concept that is important to understand.

The dimension environment is manipulated by the user by entering one of the following set commands:

`set lat|lon|lev|time` *val1 <val2>*

This set command sets one dimension of the dimension environment using world coordinates.

Alternatively:

`set x|y|z|t` *val1 <val2>*

This sets one dimension of the dimension environment using grid coordinates. You may use whatever coordinates are convenient to you. Issuing `set lon` is equivalent to issuing `set x`, both set the x dimension. The difference is only the units you wish to enter the command in.

When you enter just one value, that dimension is said to be "fixed". When you enter two values, that dimension is said to be "varying". The combination of fixed and varying dimensions defines the dimension environment.

Examples:

```
set lon -180 0    sets longitude to vary from 180W to 0
set lat 0 90      sets latitude to vary from the equator to 90N
set lev 500       sets the level to 500mb - a fixed dimension
set t 1           sets time to the first time in the data set--using grid
```

coordinates in this case. Time is now a fixed dimension

When **all** dimensions are fixed, you are referring to a **single** data point.

When **one** dimension is varying, you are referring to a **1-D "slice"** through the data set.

When **two** dimensions are varying, you are referring to a **2-D "slice"** through the data set.

When **three** dimension vary, GrADS interprets this as a **sequence of 2-D slices**.

An important note: When you enter dimensions in grid coordinates, they are always converted to world coordinates. This conversion requires some knowledge of what scaling is in use for grid to world conversions. The scaling that is used in all cases (except one) is the scaling of the *default file*. The exception is when you supply a dimension expression within a variable specification, which will be covered later.

Grads Variables

Variable names

The complete specification for a variable name is:

`abbrev.file#(dimexpr,dimexpr,...)` where:

`abbrev` is the abbreviation for the variable as specified in the data descriptor file

`file#` is the file number that contains this variable. The default initially is 1. (`set dfile` changes the default).

`dimexpr` is a dimension expression that locally modifies the current dimension environment.

A dimension expression is used to locally modify the dimension environment for that variable only. Only fixed dimensions can be thus modified.

An absolute dimension expression is:

```
X|Y|Z|T|LON|LAT|LEV|TIME = value
```

A relative dimension expression (relative to the current dimension environment):

```
X|Y|Z|T|LON|LAT|LEV|TIME +/- offset
```

Examples of variable specifications are:

```
z.3(lev=500)        File 3, absolute dimension expression
tv.1(time-12hr)     Relative dimension expression
rh                  Default file number is used
q.2(t-1,lev=850)    Two dimension expressions
z(t+0)              This does have uses....
```

An important note: When you enter a dimension in grid units, GrADS always converts

it to world coordinates. This conversion is done using the scaling of the *default file*. However, when a grid coordinate (x,y,z,t) is supplied within a dimension expression as part of a variable specification, the scaling for that file (ie, the file that variable is to be taken from) is used.

GrADS has a few "**predefined**" variable names. You can think of these as being variables implicitly contained within any opened gridded file. The variable names are:

```
lat
lon
lev
```

When used, they will contain the `lat`, `lon`, and `lev` at the respective grid points, using the scaling of the appropriate file. You can specify: `lat.2` for example, to get latitudes on the grid of the 2nd opened data set.

Defining new variables

The `define` command allows you to interactively create a new variable. The syntax is:

```
define varname = expression
```

The new variable can then be used in subsequent `define` and/or `display` commands. The new variable is stored in memory, not on disk, so avoid defining variables over large dimension ranges.

Defined variables cover the dimension ranges in effect at the time the command is issued. You may define a variable that has from 0 to 4 varying dimensions. The `define` command is the only case within GrADS where four varying dimensions is valid.

When `Z` and/or `T` are varying dimensions, the `define` command evaluates the expression by stepping through `Z` and `T`. In other words, the expression is evaluated within a dimension environment that has fixed `Z` and `T`. This will affect how you compose the expression.

When you use a defined variable, data is taken from the variable in a way similar to data taken from a GrADS data file. For example, say you define a four dimensional variable:

```
set lon -180 0
set lat 0 90
set lev 1000 100
set t 1 10
```

```
define temp = rh
```

After issuing the define command, remember to change the dimension environment so less than 4 dimensions are varying!

```
set t 5
set lev 500
d temp
```

The display of the defined variable will display a 2-D slice taken at time 5 and level 500.

If you define a variable that has fixed dimensions, and then later access this variable, the fixed dimensions are treated as "wild cards". The best way to show this is with an example:

```
set lon -180 0
set lat 0 90
set lev 500
set t 10
define zave = ave(z,t=1,t=30)
```

The defined variable has two varying dimensions. If we now display this variable (or use it in an expression), the fixed dimensions of the defined variable, namely Z and T, will match ANY Z and T dimension setting:

```
set t 1
set lev 200
d zave
```

In the above display, the variable zave would be displayed as it was defined, ie you would get a time average of 500mb heights, even though the level is set to 850.

When the defined variable has varying dimensions, and you have a dimension environment where that dimension is fixed, the proper dimension will be retrieved from the variable:

```
set lon -180 0
set lat 0 90
set lev 500
set t 10
define temp = z
set lat 40
d temp
```

In the above example, the defined variable has a varying Y dimension. We then fix the Y dimension to be 40N, and display a 1-D slice. The data from 40N in the defined grid will be accessed. If you then did:

```
set lat -40
d temp
```

The data from 40S would be accessed from the defined variable. Since this is beyond the dimensions originally used when the variable was defined, the data would be set to missing.

You can also locally override the dimension environment:

```
d temp(lat=50)
```

If that dimension is a varying dimension within the defined variable. If the dimension is a fixed dimension for that variable, the local override will be ignored:

```
d temp(t=15)
```

In the above command, the defined variable temp has fixed T, so the t=15 would be ignored.

N.B.: The `define` command currently supports only grids.

Once you have defined a grid variables, you may tell GrADS that the new variable is climatological, ie that you wish to treat the time dimension of the new variable in a wild card sense.

The command is:

`modify` varname <*seasonal/diurnal*>

where `varname` is the name of a defined variable. If the grid is described as `seasonal`, then it is assumed that the defined variable contains monthly (or multi month) means. Daily or multi-day means are not yet supported. If `diurnal` is specified, it is assumed the defined variable contains means over some time period less than a day.

After describing the defined variable as climatological, then the date/times are treated appropriately when data is accessed from the defined variable.

In the following example, the data set contains 10 years of monthly means:

```
set lon -180 180
set lat -90 90
set lev 500
set t 1 12
define zave = ave(z,t+0,t=120,1yr)
```

This define will set up a variable called zave which contains 12 times, each time being the 10 year mean for that month. We are making use here of the fact that the define command loops through a varying time dimension when evaluating the expression, and within the ave function we are making use of the variable time offset of t+0, which uses a start time that is whatever time the define command is using as it loops.

```
modify zave seasonal
set t 120
d z - zave
```

The final display will remove the 10 year monthly mean for December from the last December in the data set.

Undefining variables

Each variable defined using the define command reserves some system resources. If you no longer need a defined variable it is sensible to free these resources for other use. This is accomplished with the undefine command. For example:

```
undefine p
```

would free the resources used by the defined variable p. Of course, the variable p would no longer be available for GrADS processing.

Expressions

A GrADS expression consists of operators, operands, and parentheses. Parentheses are used the same as in FORTRAN to control the order of operation.

Operators are:

- `+` Addition
- `-` Subtraction
- `*` Multiplication
- `/` Division

Operands are:

`variable specifications, functions, and constants`.

Operations are done on equivalent grid points in each grid. Missing data values in either grid give a result of a missing data value at that grid point. Dividing by zero gives a result of a missing data value at that grid point.

Operations cannot be done between grids that have different scaling in their varying dimensions -- i.e., grids that have different rules for converting the varying dimensions from grid space to world coordinate space. This can only be encountered when you are attempting operations between grids from different files that have different scaling rules.

If one grid has more varying dimensions than the other, the grid with fewer varying dimensions is 'expanded' and the operation is performed.

Some examples of expressions:

`z - z(t-1)` (Height change over time)
`t(lev=500)-t(lev=850)` (Temp change between 500 and 850)
`ave(z,t=1,t=5)` (Average of z over first 5 times in file)
`z-ave(z,lon=0,lon=360,-b)` (Remove zonal mean)
`tloop(aave(p,x=1,x=72,y=1,y=46))` (Time series of globally averaged precip, on a 72x46 grid)

Using Templates

Grads allows you use a single data descriptor file to aggregate multiple data files and handle them as if they were one individual file. The individual data files must be identical in all dimensions except time and in a format GrADS can read. The time range of each individual file must be indicated it its filename.

An example might be a timeseries spanning a single month, where each day's worth of hourly data is contained in individual files:

```
1may92.dat
2may92.dat
...
31may92.dat
```

In order to tell GrADS that there are multiple files in this time series, three records are modified in the data descriptor (.ctl) file:

```
DSET %d1%mc%y2.dat
OPTIONS template
TDEF 744 linear 0z1may1992 1hr
```

First, the `DSET` entry has a substitution template instead of a filename. See below for a description of all the possible components of the template. Second, the `OPTIONS` entry contains the `template` keyword. Third, the `TDEF` entry describes the time range for the entire set of data files.

Templating works on the following GrADS data types: gridded binary, GRIB, and station data. If you specify any additional `options` keywords in the data descriptor file, make sure the options apply equally to each file included in the template.

Valid components of the substitution template are as follows:

`%y2`	2 digit year
`%y4`	4 digit year
`%m1`	1 or 2 digit month
`%m2`	2 digit month (leading zero if needed)
`%mc`	3 character month abbreviation
`%d1`	1 or 2 digit day
`%d2`	2 digit day (leading zero if needed)
`%h1`	1 or 2 digit hour
`%h2`	2 digit hour
`%h3`	3 digit hour (e.g., 120 or 012)

%f2	2 or 3 digit forecast hour
%f3	3 digit forecast hour
%n2	2 digit minute (leading zero if needed)

When specifying the initial time (e.g., NWP model output from NMC and FNMOC), use these substitutions:

%iy2	initial 2 digit year
%iy4	initial 4 digit year
%im1	initial 1 or 2 digit month
%im2	initial 2 digit month (leading zero if needed)
%in2	initial 2 minute (leading zero if needed)
%imc	initial 3 character month abbreviation
%id1	initial 1 or 2 digit day (leading zero if needed)
%id2	initial 2 digit day
%ih1	initial 1 or 2 digit hour
%ih2	initial 2 digit hour
%ih3	initial 3 digit hour

About GrADS Station Data

This section describes the structure of station data files, how to create them, and how to instruct GrADS to interpret them properly. Please refer to the companion section on Using Station Data for information about the GrADS commands and functions that are available for analyzing and displaying station data. Here are some quick links for skipping through this section:

Structure of a Station Data File

Station data are written to a binary file one report at a time. Groups of station reports are ordered within the file according to the time interval. The time interval for a set of upper air soundings might be 12 hours, and the time interval for a set of surface observations might be 1 hour.

Variables within each report are split into two categories: surface and level-dependent. Surface variables are reported at most once per report, and level-dependent variable are reported at multiple pressure levels within each report.

Each station report in a binary station data file has the following structure:

- A header which provides information about the location of the station
- Surface variables, if any
- Level dependent variables, if any

The header is described by the following C language data structure:

```
struct reportheader {
  char  id[8];    /* Station ID */
  float lat;      /* Latitude of Station */
  float lon;      /* Longitude of Station */
  float t;        /* Time in grid-relative units */
  int   nlev;     /* Number of data groups following the header
*/
  int   flag;     /* Level independent var set flag */
};
```

A detailed description of each header entry follows:

`id`

The station ID uniquely identifies the station. It can be 1 to 7 characters long and may be assigned arbitrarily; ie. the stations could be numbered in some arbitrary order.

`lat, lon`

The location of the station, given in world coordinates (latitude and longitude).

`t`

The time of this report, in grid-relative units. This refers to the way the stations are grouped in time. For example, if you are working with surface airways reports, you would probably have a time grouping interval of one hour. If you wanted to treat the report times of each report as being exactly on the hour, you would set t to 0.0. If the report was for 12:15pm, and you were writing the time group for 12pm, you would set t to be 0.25. Thus, t would typically have the range of - 0.5 to 0.5.

`nlev`

Number of data groups following the header. This is the count of the one surface group, if present, plus the number of level dependent groups. Is set to zero to mark the end of a time group in the file.

`flag`

If set to `0`, there are no surface variables following the header. If set to `1`, then there are surface variables following the header.

The surface variable data (if present) are written to file following the header. Surface variables are written out as floating point numbers in the order they are listed in the data descriptor file. Each of the surface variables must be written -- missing variables should contain the missing data value specified in the data descriptor file. The group of surface variable data must be the same size for each report in the file.

The level-dependent variables are written to file following the surface variables as follows:

`level` -- a floating point value giving the Z dimension in world

coordinates for this level.
`variables` -- The level-dependent variables for this level.

Each level dependent group must have all the level dependent variables present, even if they are filled with the missing data value. The group of level dependent variable data must be the same size for all levels and all reports in the file.

After all the reports for one time grouping have been written, a special header (with no data groups -- `nlev` set to zero) is written to indicate the end of the time group. The next time group may then start immediately after. A time group with no reports would still contain the time group terminator header record (ie, two terminators in a row).

Creating a Station Data File

GrADS station data files must be written out in the structure outlined in the previous section. Examples of C and FORTRAN programs to create station data sets are provided below.

Let's say you have a data set with monthly rainfall:

```
Year   Month  Stid    Lat    Lon    Rainfall
1980     1     QQQ    34.3  -85.5    123.3
1980     1     RRR    44.2  -84.5     87.1
1980     1     SSS    22.4  -83.5    412.8
1980     1     TTT    33.4  -82.5     23.3
1980     2     QQQ    34.3  -85.5    145.1
1980     2     RRR    44.2  -84.5    871.4
1980     2     SSS    22.4  -83.5    223.1
1980     2     TTT    33.4  -82.5     45.5
```

A sample DEC FORTRAN program to write this data set in GrADS format is given below. Note that the OPEN statement is set to write a stream data set. This option may not not available with every compiler. If your program writes out data in sequential format, you must add an `"OPTIONS sequential"` entry to your GrADS data descriptor file.

```
       CHARACTER*8 STID
       OPEN (8,NAME='rain.ch')
       OPEN (10,NAME='rain.dat',FORM='UNFORMATTED',RECORDTYPE='STREAM')
       IFLAG = 0
C  Read and Write
10     READ (8,9000,END=90) IYEAR,IMONTH,STID,RLAT,RLON,RVAL
9000   FORMAT (I4,3X,I2,2X,A8,3F8.1)
       IF (IFLAG.EQ.0) THEN
          IFLAG = 1
```

```
            IYROLD = IYEAR
            IMNOLD = IMONTH
         ENDIF
C  If new time group, write time group terminator.
C  Assuming no empty time groups.
         IF (IYROLD.NE.IYEAR.OR.IMNOLD.NE.IMONTH) THEN
            NLEV = 0
            WRITE (10) STID,RLAT,RLON,TIM,NLEV,NFLAG
            ENDIF
            IYROLD = IYEAR
            IMNOLD = IMONTH
C  Write this report
         TIM = 0.0
         NLEV = 1
         NFLAG = 1
         WRITE (10) STID,RLAT,RLON,TIM,NLEV,NFLAG
         WRITE (10) RVAL
         GO TO 10
C  On end of file write last time group terminator.
90       CONTINUE
         NLEV = 0
         WRITE (10) STID,RLAT,RLON,TIM,NLEV,NFLAG
         STOP
         END
```

An equivalent C program might be:

```
#include <stdio.h>
/* Structure that describes a report header in a stn file */
struct rpthdr {
  char  id[8];    /* Station ID */
  float lat;      /* Latitude of Station */
  float lon;      /* Longitude of Station */
  float t;        /* Time in grid-relative units */
  int   nlev;     /* Number of levels following */
  int   flag;     /* Level independent var set flag */
} hdr;

main ()
{
  FILE  *ifile, *ofile;
  char  rec[80];
  int   flag,year,month,yrsav,mnsav,i;
  float val;

/* Open files */
  ifile = fopen ("rain.ch","r");
  ofile = fopen ("rain.dat","wb");
  if (ifile==NULL || ofile==NULL) {
    printf("Error opening files\n");
    return;
```

```
  }

/* Read, write loop */
  flag = 1;
  while (fgets(rec,79,ifile)!=NULL) {
    /* Format conversion */
    sscanf (rec,"%i %i",&year,&month);
    sscanf (rec+20," %g %g %g",&hdr.lat,&hdr.lon,&val);
    for (i=0; i<8; i++) hdr.id[i] = rec[i+11];
    /* Time group terminator if needed */
    if (flag) {
      yrsav = year;
      mnsav = month;
      flag = 0;
    }
    if (yrsav!=year || mnsav!=month) {
      hdr.nlev = 0;
      fwrite(&hdr,sizeof(struct rpthdr), 1, ofile);
    }
    yrsav = year;
    mnsav = month;
    /* Write this report */
    hdr.nlev = 1;
    hdr.flag = 1;
    hdr.t = 0.0;
    fwrite (&hdr,sizeof(struct rpthdr), 1, ofile);
    fwrite (&val,sizeof(float), 1, ofile);
  }
  hdr.nlev = 0;
  fwrite (&hdr,sizeof(struct rpthdr), 1, ofile);
}
```

Station Data Descriptor File

After creating a binary file containing your station data, you must write a station data descriptor file so GrADS knows how to interpret the binary data file. The format for the data descriptor file for station data is similar to the format for a gridded data set, but there are a few differences as well as additional entries that are unique to station data descriptor files. These differences are outlined below. For further information on all the entries of a descriptor file, consult the secion of the User's Guide on Elements of a GrADS Data Descriptor File.

Here is an example of a station data descriptor file. Remember that the variables must be listed in the same order as they were written to the binary file.

```
DSET   ^station_data_sample.dat
DTYPE  station
STNMAP station_data_sample.map
```

```
UNDEF  -999.0
TITLE  Station Data Sample
TDEF   10 linear 12z18jan1992 12hr
VARS 11
ps    0  99  Surface Pressure
ts    0  99  Surface Temperature
ds    0  99  Surface Dewpoint Temperature
us    0  99  Surface U-Wind
vs    0  99  Surface V-Wind
elev  0  99  Elevation of Station
z     1  99  Height
t     1  99  Temperature
d     1  99  Dewpoint Temperature
u     1  99  U-Wind
v     1  99  V-Wind
ENDVARS
```

Note the following differences between this descriptor file and a gridded data descriptor file:

DTYPE station

This entry identifies the data file as station data.

STNMAP *filename*

This entry identifies the file name of the station map file created by the stnmap utility.

XDEF, YDEF, and ZDEF

These entries are not included in a station data control file.

TDEF

This entry gives the number of the time groups in the file, the time stamp for the first group, and the interval between time groups.

VARS

The surface variables are listed first, and contain a "0" in the *levs* field. Level-dependent variables are listed after the surface variables, and contain a "1" in the *levs* field.

STNMAP Utility

Once the station data set has been created and the descriptor file has been written, the final step is to create the station map file by running the `stnmap` utility. This utility is executed externally from the command line, not from within GrADs. `stnmap` writes out information that allows GrADS to access the station report data more efficiently. The output from `stnmap` is called the station map file and its name is identified in the `STNMAP` entry of the data descriptor file. `stnmap` will prompt the user for the name of the data descriptor file, or it can be specified as an input argument on the command line. Station map files must be created on the machine where they are to be used. Consult the reference page for more information.

If you change the station data file, perhaps by appending another time group, then you will also have to change the descriptor file to reflect the changes and then rerun the `stnmap` utility.

Using GrADS Station Data

This section describes some of the GrADS commands and functions that are available for analyzing and displaying station data. Please refer to the companion section About Station Data for information on the structure of station data files, how to create them, and how to instruct GrADS to interpret them properly.

Here are some quick links for skipping through this section:

Operating on Station Data

Currently, station data operations and display are supported for three distinct dimension environments:

- X, Y varying (horizontal X, Y plot)
- Z varying (vertical profile)
- T varying (time series)

Operations may be done on station data as with gridded data. Operations between grids and station data are not supported.

Operations between station data are defined as being the operation performed on data points that have exactly the same varying dimension values.

For example, if T is the only varying dimension, the expression:

```
display ts-ds
```

would result in a time series of station data reports being retrieved for two separate variables. Then, for station reports having exactly the same time, the operation is performed. Note that duplicates are ignored, with the operation being performed between the first occurrences encountered.

When both X and Y are both fixed dimensions, the variable specification may include a station identifier, which specifies a local override for both lat and lon.

The syntax for this would be:

```
varname(stid=ident)
```

The station identifiers are case insensitive.

Some functions do not support station data types. These are:

hdivg hcurl vint maskout ave aave tloop

When X and Y are varying, station data values are displayed as numbers centred at their locations. If two expressions are supplied on the display command (ie, display ts;ds) then two values are displayed, above and below the station location. The display is controlled by the following set commands:

set ccolor *color*
set dignum *digits*
set digsiz *size*
set stid *on|off*

The set stid command controls whether the station identifier is displayed with each value.

Plotting Station Models

GrADs will plot station models from station data. This is enabled by:

set gxout model

The appropriate display command is:

```
display u;v;t;d;slp;delta;cld;wx;vis
```

where:

u and v are the wind components. A wind barb will be drawn using these values. If either is missing, the station model will not be plotted at all.

t, d, slp, and delta are plotted numerically around the station model

cld is the value of the symbol desired at the center of the station model. Values 1 to 9 are assumed to be the marker types (ie, circle, square, crosshair, etc). Values

20 to 25 are assumed to be cloudiness values:

20 -clear
21 -scattered 22 -broken
23 -overcast
24 -obscured
25 -missing (M plotted)

wx is the value of the wx symbol (see draw wxsym) to be plotted in the wx location.

vis is the visibility as a real number. It will be plotted as a whole number and a fraction.

When any of these items are missing (other than u and v), the model is plotted without that element. To represent a globally missing value, enter a constant in the display command. For example, if the delta were always missing, use:

```
display u;v;t;d;slp;0.0;cld
```

The station models respond to the usual set commands such as set digsiz, set dignum, set cthick, set ccolor.

In addition, there is:

```
set stnopts
```

which will cause the model to plot the number in the slp location as a three digit number, with only the *last* three digits of the whole number plotted. This allows the standard 3 digit sea level pressure to be plotted by enabling dig3 and plotting slp*10.

Drawing Arbitrary Cross Sections

Drawing arbitrary vertical cross sections based on a collection of station data profiles involves transforming station data (scattered observations) into gridded data so as to take advantage of the GrADS grid display and analysis features.

The first step is to form a collection of 1-D data (Z or T varying). The collect command saves station data profiles or time series in memory as a set. The 1-D data may be either real station data or gridded data converted to station data using gr2stn.

The second stop is to convert the collection of station data into a grid for display or analysis purposes. This is accomplished by the new function coll2gr.

coll2gr does not yet support time slices; currently, it will only work when the collection of stations is a collection of vertical profiles.

coll2gr produces an output grid that varies in X and Z; the dimension environment used when coll2gr is invoked must also be X and Z varying. The X axis of the output grid will contain the equally spaced station profiles and will span the range of the current X dimension environment. The Z axis of the output grid will span the range of the current Z dimension environment and will have either the specified number of levels or a union of the levels. Data points outside of the range of levels will be used for interpolating to within the range if appropriate.

The X axis of the output grid from coll2gr is artificial in terms of the world coordinates -- it doesn't really represent longitudes. A way to completely control the labelling of the display output is provided:

```
set xlabs lab1 | lab2 | lab3 ...
set ylabs lab1 | lab2 | lab3 ...
```

Each label string may include blanks. The labels will be plotted equally spaced along the indicated axis. Spacing can be modified by adding blank strings:

```
set xlabs | | | | lab1 | ...
```

Here is a sample script written by M. Fiorino that uses these features:

```
************************************************************************
* The following lines will display an arbitrary X section
* from one specified point to another.
*
* lon1 is the westernmost longitude point
* lon2 is the easternmost longitude point
* lat1 is the latitude that corresponds to lon1
* lat2 is the latitude that corresponds to lon2
*
* The loop is used to interpolate between points in
* the arbitrary cross section. This code will plot
* any cross section as long as you specify the points.
* My code plots cross sections of PV after I calculated
* PV on 11 pressure surfaces. I have another script
* that plots cross sections of potential temperature, and
```

```
* the code is very similar to this, except theta is substituted
* for PV.
*
* Many thanks to Brian Doty at COLA for his help with this code.
*
*********************************************************************

'open pv.ctl'
'set grads off'
'set zlog on'
'set x 1'
'set y 1'
'set lev 1000 100'
lon1 = -95.0
lon2 = -90.0
lat1 = 55.0
lat2 = 15.0
lon = lon1
'collect 1 free'
while (lon <= lon2)
  lat = lat1 + (lat2-lat1)*(lon-lon1) / (lon2-lon1)
  'collect 1 gr2stn(pv,'lon','lat')'
  lon = lon + 1
endwhile

'set x 14 16'
'set xaxis 'lon1' 'lon2
'set clab on'
'set gxout shaded'
'set clevs 0 .5 15'
'set ccols 0 0 7 0'
'd coll2gr(1,-u)'
'set gxout contour'
'set cint .5'
'd coll2gr(1,-u)'
```

User Defined Functions (UDFs)

Users may write their own GrADS functions in the computer language of their choice, and have them available from the GrADS expression facility (via the `display` command). Some possible user defined functions might be:

- filtering functions
- grid interpolation functions
- thermodynamic functions

You may write a function that can be invoked via the GrADs expression facility. This function may be written in any computer language, and may perform any desired I/O, calculations, etc. Please read the following documentation carefully to understand the restrictions to this capability.

Overview of User Defined Functions

The steps that GrADS uses to invoke a user defined function are:

1. When GrADS is first started, it reads a file that describes the user defined functions. This file is called the 'user defined function table'.

2. When a user function is invoked via the display command expression, GrADS parses the arguments to the functions, obtains the results of any expressions, and writes the resultant data to a 'function data transfer file'.

   Please note that in a user-defined function adding the double quote ("") around a `char` argument passes the string directly *without* the usual conversion to lower case and removal of blanks, e.g.,

   ```
   d grhilo(slp,F8.2,"This is the Label",0.25)
   ```

   Here `F8.2` is passed as `f8.2`, but the second character string would not be

converted to `thisisthelabel`.

3. A user written program is then invoked. This program may read the function data transfer file, do any desired processing, then write the result into a function result file.

4. GrADS will read the function result file and generate the internal objects necessary for this result to participate in the remainder of the expression evaluation.

The user defined function table

The user defined function table (UDFT) is a simple text file that contains information about each user defined function. There are five records for each defined function, and the file may contains descriptions for any number of functions. The 5 records are:

Record 1: This record contains several blank delimited fields:

Field 1: The name of the function, 1-8 characters, beginning with a letter. The name should be in lower case. Note that function names are not case dependent, and that GrADS converts all expression to lower case before evaluation.

Field 2: An integer value, specifying the minimum number of arguments that the function may have.

Field 3: An integer value, specifying the maximum number of arguments that the function may have. This may not be more than 8.

Field 4 to N: A keyword describing the data type of each argument:

`expr:` The argument is an expression.
`value:` The argument is a data value.
`char:` The argument is a character string.

Record 2: This record contains blank delimited option keywords. Current keywords are:

`sequential` - GrADS will write data to the function data transfer file in FORTRAN sequential unformatted records. This is typically appropriate if the function is written in FORTRAN.

`direct` - GrADS will write data to the function data transfer file without

any record descriptor words. This is typically appropriate if the function is written in C.

Record 3: This record contains the file name of the function executable routine. This routine will be invoked as its own separate process via the `system` call. Do a `man system` if you would like more information on the rules governing this system feature.

Record 4: This record contains the file name of the function data transfer file. This is the file that GrADS will write data to before invoking the user function executable, and is typically the file the function will read to obtain the data to be operated upon.

Record 5: This record contains the file name of the function result file. The function writes the result of its operations into this file in a specified format, and GrADS reads this file to obtain the result of the function calculation.

The user function definition table itself is pointed to by the environment variable GAUDFT. If this variable is not set, the function table will not be read. An example of setting this variable is:

```
setenv GAUDFT /usr/local/grads/udft
```

User defined functions have precedence over GrADS intrinsic functions, thus a user defined function can be set up to replace a GrADS function. Be sure you do not do this inadvertently by choosing a function name already in use by GrADS.

Format of the function data transfer file

The function data transfer file contains a header record plus one or more records representing each argument to the function. The user function routine will know what data types to expect (since they will be specified in the UDFT), and can read the file in a predictable way.

Header record: The header record always contains 20 floating point numbers. The record will always be the same size. Values defined in this record are:

1st value: Number of arguments used when invoking the function.

2nd value: Set to zero, to indicate this particular transfer file format. The function should test this value, and return an error if non-zero, in order to be compatible with future enhancements to this file format.

Values 3 to 20: Reserved for future use.

Argument records: The argument records are written out in the order that the arguments are presented. The contents of the argument records depends on the data type of the argument: value, character string, or expression. Each of these data types will result in a different argument record being written out:

- `value`: If the argument data type is a value, then the argument record will contain a single floating point value.

- `char`: If the argument data type is a character string, then the argument record will an 80-byte character array that contains the argument string. If the argument string is longer than 80 bytes, the trailing bytes will be lost. If the argument is shorter, it will be padded with blanks. Note that the argument will already be processed by the GrADS expression parser, which will convert all characters to lower case and remove any blanks.

- `expr`: If the argument data type is a gridded expression, then GrADS will evaluate the expression and write a series of records to the transfer file. Listed below are the records that will be written to the transfer file for each argument that is a gridded expression:

  1st record: This record contains 20 values, all floating point, that make up the header for the gridded expression. Note that some of the values are essentially integer, but for convenience they are written as a floating point array. Appropriate care should be taken in the function program when converting these values back to integer.

  1 -- Undefined value for the grid

  2 -- An index to identify the i dimension (idim). Options for the index are:

-1	None
0	X dimension (lon)
1	Y dimension (lat)
2	Z dimension (lev)
3	T dimension (time)

  3 -- An index to identify the j dimension (jdim). Options are the same as for idim. If both idim and jdim are -1, the grid is a single value.

4 -- number of elements in the i direction (isiz).

5 -- number of elements in the j direction (jsiz).

6 -- i dimension linear flag. If 0, the dimension has non-linear scaling.

7 -- j dimension linear flag. If 0, the dimension has non-linear scaling.

8 -- istrt. This is the world coordinate value of the first idim element, ONLY if idim has linear scaling and idim is not time.

9 -- iincr. This is the increment of the world coordinate values for idim, ONLY if idim has linear scaling.

10 -- jstrt. This is the world coordinate value of the first jdim element, ONLY if jdim has linear scaling and jdim is not time.

11 -- jincr. This is the increment of the world coordinate values for jdim, ONLY if jdim has linear scaling.

12 -- If one of the dimensions is time, values 12 to 16 define the start time:

12: start year
13: start month
14: start day
15: start hour
16: start minute

17 -- If one of the dimensions is time, values 17 and 18 define the time increment:

17: time increment in minutes
18: time increment in months

(GrADS handles all increments in terms of minutes and months.)

19,20 -- reserved for future use

2nd record: This record contains the actual grid of data. It contains isiz*jsiz floating point elements.

3rd record: This record contains the world coordinate values for each grid element in the i dimension. Thus, the record will contain isiz floating point elements.

4th record: This record contains the world coordinate values for each grid element in the j dimension. Thus, the record will contain jsiz floating point elements.

Format of the function result file

The function result file returns the result of the user defined function to GrADS. It is the responsibility of the function program to write this file in the proper format. A file written out in an improper format may cause GrADS to crash, or to produce incorrect results.

The result of a function is always a grid. Thus, the format of the function result file is as follows:

Header record: The header record should always contain 20 floating point numbers. The record will always be the same size. Values defined in this record are:

1st value: This value contains the return code. Any non-zero return code causes GrADS to assume the function detected an error, and GrADS does not read any further output.

2nd value: Set to zero, to indicate this particular transfer file format. The function should test this value, and return an error if non-zero, in order to be compatible with future enhancements to this file format.

Values 3 to 20: Reserved for future use.

Grid records: The grid records should be written in the same order and format as the `expr` argument record in the data transfer file, with one important exception: the 3rd and 4th records containing the world coordinate values for each grid element in the i and j dimensions are written out to the function result file only if the scaling is non-linear. Thus the transfer file and the result file are not symmetric: GrADS writes a transfer file with record #3 and #4 always included, but it does NOT like to see record #3 and #4 in the result file if the dimensions are linear.

The linear/non-linear scaling of the grid dimensions is determined by examining the grid header contents -- values 6 and 7 contain the idim and jdim linear flags. Note that the time dimension is always linear.

Example: Linear Regression Function

This is a simple example of what a user defined function might look like in FORTRAN. This is a simple linear regression function, which only handles a 1-D grid and takes one argument, and expression.

First, the user defined function table (UDFT):

```
      linreg 1 1 expr
      sequential
      /mnt/grads/linreg
      /mnt/grads/linreg.out
      /mnt/grads/linreg.in
```

The source code for the FORTRAN program linreg is:

```
      real vals(20),ovals(20)
      real x(10000),y(10000)
c
      open (8,file='/mnt/grads/linreg.out',form='unformatted')
      open (10,file='/mnt/grads/linreg.in',form='unformatted')
c
      read (8)
      read (8) vals
      idim = vals(2)
      jdim = vals(3)
c
c  If this is not a 1-D grid, write error message and exit
      if (idim.eq.-1 .or. jdim.ne.-1) then
        write (6,*) 'Error: Invalid dimension environment'
        vals(1) = 1
        write (10) vals
        stop
      endif
c
c  If the grid is too big, write error message and exit
      isiz = vals(4)
      if (isiz.gt.10000) then
        write (6,*) 'Error from linreg: Grid too big'
        vals(1) = 1
        write (10) vals
        stop
      endif
c
c  Read the data
      read (8) (y(i),i=1,isiz)
c
c  Read non-linear scaling if necessary
      ilin = vals(6)
      if (ilin.eq.0) then
```

```
        read (8) (x(i),i=1,isiz)
      else
        do 100 i=1,isiz
          x(i) = i
100     continue
      endif
c
c  Do linear regression
      call fit (x,y,isiz,a,b)
c
c  Fill in data values
      do 110 i=1,isiz
        y(i) = a+x(i)*b
110   continue
c
c  Write out return info.
c  The header and the non-linear scaling
c  info will be the same as what GrADs gave us.
      ovals(1) = 0.0
      write (10) ovals
      write (10) vals
      write (10) (y(i),i=1,isiz)
      if (ilin.eq.0) write(10) (x(i),i=1,isiz)
c
      stop
      end
c
c-------------------------------------------------
      SUBROUTINE FIT(X,Y,NDATA,A,B)
c
c A is the intercept
c B is the slope
c
      REAL X(NDATA), Y(NDATA)
c
      SX = 0.
      SY = 0.
      ST2 = 0.
      B = 0.
      DO 12 I = 1, NDATA
        SX = SX + X(I)
        SY = SY + Y(I)
12    CONTINUE
      SS = FLOAT(NDATA)
      SXOSS = SX/SS
      DO 14 I = 1, NDATA
        T = X(I) - SXOSS
        ST2 = ST2 + T * T
        B = B + T * Y(I)
14    CONTINUE
      B = B/ST2
      A = (SY - SX * B)/SS
```

```
RETURN
END
```

Using Map Projections in GrADS

It is important to understand the distinction between the two uses of map projections when creating GrADS displays of your data:

- projection of the data (preprojected grids);
- projection of the display.

GrADS supports two types of data grids:

- `lon/lat` grids (and not necessarily regular, e.g., gaussian);
- preprojected grids.

Using Preprojected Grids

Preprojected data are data **already** on a map projection. GrADS supports four types of preprojected data:

1. N polar stereo (NMC model projection);
2. S polar stereo (NMC model projection) ;
3. Lambert Conformal (originally for Navy NORAPS model);
4. NMC eta model (unstaggered).
5. More precise N and S polar stereo (hi res SSM/I data)
6. Colorado State University RAMS model (oblique polar stereo; beta)

When preprojected grids are opened in GrADS, bilinear interpolation constants are calculated and all date are displayed on an internal GrADS lat/lon grid defined by the xdef and ydef card in the data description or .ctl file (that's why it takes longer to "open" a preprojected grid data set).

It is very important to point out that the internal GrADS grid can be any grid as it is completely independent of the preprojected data grid. Thus, there is nothing stopping you displaying preprojected data on a very high res lon/lat grid (again, defined in the .ctl by xdef and ydef). In fact, you could create and open multiple .ctl files with different resolutions and/or regions which pointed to the same preprojected data file.

When you do a display (i.e., get a grid of data), the preprojected data are bilinearly interpolated to the GrADS internal lat/lon grid. For preprojected scalar fields (e.g., 500 mb heights), the display is adequate and the precision of the interpolation can be controlled by xdef and ydef to define a higher spatial resolution grid.

The big virtue of this approach is that all built in GrADS analytic functions (e.g., aave, hcurl...) continue to work even though the data were not originally on a lon/lat grid. The downside is that you are not looking directly at your data on a geographic map. However, one could always define a .ctl file which simply opened the data file as i,j data and displayed without the map (set mpdraw off). So, in my opinion, this compromise is not that limiting even if as a modeller you wanted to look at the grid--you just don't get the map background.

Preprojected vector fields are a little trickier, depending on whether the vector is defined relative to the data grid or relative to the Earth. For example, NMC polar stereo grids use winds relative to the data grid and thus must be rotated to the internal GrADS lat/lon grid (again defined in the .ctl file by the xdef and ydef cards).

The only potential problem with working with preprojected data (e.g., Lambert Conformal model data) is defining the projection for GrADS. This is accomplished using a pdef card in the data descriptor .ctl file.

Polar Stereo Preprojected Data (coarse accuracy for NMC Models)

Preprojected data on a polar stereo projection (N and S) is defined as at NMC. For the NCEP model GRIB data distributed via anon ftp from ftp.ncep.noaa.gov, the

`pdef` card is:

```
pdef isize jsize projtype ipole jpole lonref
gridinc
pdef 53 45 nps 27 49 -105 190.5
```

where,

`ipole` and `jpole` are the (i,j) of the pole referenced from the lower left corner at (1,1) and `gridinc` is the dx in km.

The relevant GrADS source is:

```
void w3fb04 (float alat, float along, float xmeshl, float orient, float *xi, float *xj)
{
/*
C
C SUBPROGRAM: W3FB04 LATITUDE, LONGITUDE TO GRID
COORDINATES
C AUTHOR: MCDONELL,J. ORG: W345 DATE: 90-06-04
C
C ABSTRACT: CONVERTS THE COORDINATES OF A LOCATION ON
EARTH FROM THE
C NATURAL COORDINATE SYSTEM OF LATITUDE/LONGITUDE TO
THE GRID (I,J)
C COORDINATE SYSTEM OVERLAID ON A POLAR STEREOGRAPHIC
MAP PRO
C JECTION TRUE AT 60 DEGREES N OR S LATITUDE. W3FB04 IS THE
REVERSE
C OF W3FB05.
C
C PROGRAM HISTORY LOG:
C 77-05-01 J. MCDONELL
C 89-01-10 R.E.JONES CONVERT TO MICROSOFT FORTRAN 4.1
C 90-06-04 R.E.JONES CONVERT TO SUN FORTRAN 1.3
C 93-01-26 B. Doty converted to
C
C
C USAGE: CALL W3FB04 (ALAT, ALONG, XMESHL, ORIENT, XI, XJ)
C
C INPUT VARIABLES:
C NAMES INTERFACE DESCRIPTION OF VARIABLES AND TYPES
C ------ --------- ----------------------------------------------
```

```
C ALAT ARG LIST LATITUDE IN DEGREES (<0 IF SH)
C ALONG ARG LIST WEST LONGITUDE IN DEGREES
C XMESHL ARG LIST MESH LENGTH OF GRID IN KM AT 60 DEG
LAT(<0 IF SH)
C (190.5 LFM GRID, 381.0 NH PE GRID,-381.0 SH PE GRID)
C ORIENT ARG LIST ORIENTATION WEST LONGITUDE OF THE GRID
C (105.0 LFM GRID, 80.0 NH PE GRID, 260.0 SH PE GRID)
C
C OUTPUT VARIABLES:
C NAMES INTERFACE DESCRIPTION OF VARIABLES AND TYPES
C ------ --------- ------------------------------------------------
C XI ARG LIST I OF THE POINT RELATIVE TO NORTH OR SOUTH POLE
C XJ ARG LIST J OF THE POINT RELATIVE TO NORTH OR SOUTH POLE
C
C SUBPROGRAMS CALLED:
C NAMES LIBRARY
C ------------------------------------------------------ --------
C COS SIN SYSLIB
C
C REMARKS: ALL PARAMETERS IN THE CALLING STATEMENT MUST
BE
C REAL. THE RANGE OF ALLOWABLE LATITUDES IS FROM A POLE TO

C 30 DEGREES INTO THE OPPOSITE HEMISPHERE.
C THE GRID USED IN THIS SUBROUTINE HAS ITS ORIGIN (I=0,J=0)
C AT THE POLE IN EITHER HEMISPHERE, SO IF THE USER'S GRID HAS
ITS
C ORIGIN AT A POINT OTHER THAN THE POLE, A TRANSLATION IS
NEEDED
C TO GET I AND J. THE GRIDLINES OF I=CONSTANT ARE PARALLEL
TO A
C LONGITUDE DESIGNATED BY THE USER. THE EARTH'S RADIUS IS
TAKEN C TO BE 6371.2 KM.
C
C ATTRIBUTES:
C LANGUAGE: SUN FORTRAN 1.4 C MACHINE: SUN SPARCSTATION 1+
C*/
static float radpd = 0.01745329;
static float earthr = 6371.2;

float re,xlat,wlong,r;
  re = (earthr * 1.86603) / xmeshl;
  xlat = alat * radpd;
```

```
  if (xmeshl>0.0) {
    wlong = (along + 180.0 - orient) * radpd;
      r = (re * cos(xlat)) / (1.0 + sin(xlat));
      *xi = r * sin(wlong);
      *xj = r * cos(wlong);

  } else {
    re = -re;
    xlat = -xlat;
    wlong = (along - orient) * radpd;
      r = (re * cos(xlat)) / (1.0+ sin(xlat));
      *xi = r * sin(wlong);
      *xj = -r * cos(wlong);
  }
}
```

Lambert Conformal Preprojected Data

The Lambert Conformal projection (lcc) was implemented for the U.S. Navy's limited area model NORAPS. Thus, to work with your lcc data you must express your grid in the context of the Navy lcc grid. NMC has been able to do this for their AIWIPS grids and the Navy definition should be general enough for others.

A typical NORAPS Lambert-Conformal grid is described below, including the C code which sets up the internal interpolation.

The `.ctl` file is:

```
dset ^temp.grd
title NORAPS DATA TEST
undef 1e20
pdef 103 69 lcc 30 -88 51.5 34.5 20 40 -88 90000
90000
xdef 180 linear -180 1.0
ydef 100 linear -10 1.0
zdef 16 levels 1000 925 850 700 500 400 300 250
200 150 100 70 50 30 20 10
tdef 1 linear 00z1jan94 12hr
vars 1
   t 16 0 temp
endvars
```

where,

103	= #pts in x
69	= #pts in y
lcc	= Lambert-Conformal
30	= lat of ref point
88	= lon of ref point (E is positive, W is negative)
51.5	= i of ref point
34.5	= j of ref point
20	= S true lat
40	= N true lat
88	= standard lon
90000	= dx in M
90000	= dy in M

Otherwise, it is the same as other GrADS files.

Note - the xdef/ydef apply to the lon/lat grid GrADS internally interpolates to and can be anything...

The GrADS source which maps lon/lat of the GrADS internal lon/lat grid to i,j of the preprojected grid is:

```
/* Lambert Conformal conversion */
void ll2lc (float *vals, float grdlat, float grdlon,
float *grdi, float *grdj) {
/*  Subroutine to convert from lat-lon to Lambert Conformal i,j.
Provided by NRL Monterey; converted to C 6/15/94.
c                SUBROUTINE: ll2lc
c
c                PURPOSE: To compute i- and j-coordinates of a
specified
c                         grid given the latitude and longitude
points.
c                         All latitudes in this routine start
c                         with -90.0 at the south pole and
increase
c                         northward to +90.0 at the north pole.
The
c                         longitudes start with 0.0 at the
Greenwich
c                         meridian and increase to the east, so
that
c                         90.0 refers to 90.0E, 180.0 is the
inter-
c                         national dateline and 270.0 is 90.0W.
c
```

```
c                INPUT VARIABLES:
c
c   vals+0         reflat: latitude at reference point
(iref,jref)
c
c   vals+1         reflon: longitude at reference point
(iref,jref)
c
c   vals+2         iref:   i-coordinate value of reference point
c
c   vals+3         jref:   j-coordinate value of reference point
c
c   vals+4         stdlt1: standard latitude of grid
c
c   vals+5         stdlt2: second standard latitude of grid (only
required
c                  if igrid = 2, lambert conformal)
c
c   vals+6         stdlon: standard longitude of grid (longitude
that
c                          points to the north)
c
c   vals+7         delx:   grid spacing of grid in x-direction
c                          for igrid = 1,2,3 or 4, delx must be
in meters
c                          for igrid = 5, delx must be in degrees
c
c   vals+8         dely:   grid spacing (in meters) of grid in y-
direction
c                          for igrid = 1,2,3 or 4, delx must be
in meters
c                          for igrid = 5, dely must be in degrees
c
c                  grdlat: latitude of point (grdi,grdj)
c
c                  grdlon: longitude of point (grdi,grdj)
c
c                  grdi:   i-co ordinate(s) that this routine
will generate
c                          information for
c
c                  grdj:   j-coordinate(s) that this routine will
generate
c                          information for
c
*/

   float pi, pi2, pi4, d2r, r2d, radius, omega4;
   float
gcon,ogcon,ahem,deg,cn1,cn2,cn3,cn4,rih,xih,yih,rrih,check;
   float alnfix,alon,x,y;
   pi = 4.0*atan(1.0);
```

```
   pi2 = pi/2.0;
   pi4 = pi/4.0;
   d2r = pi/180.0;
   r2d = 180.0/pi;
   radius = 6371229.0;
   omega4 = 4.0*pi/86400.0;

/*mf --------------- mf*/
/*case where standard lats are the same */
  if(*(vals+4) == *(vals+5)) {
    gcon = sin(*(vals+4)*d2r);
  } else {
    gcon = (log(sin((90.0-*(vals+4))*d2r))
    log(sin((90.0-*(vals+5))*d2r)))
    /(log(tan((90.0-*(vals+4))*0.5*d2r))
    log(tan((90.0-*(vals+5))*0.5*d2r)));
  }
/*mf --------------- mf*/
  ogcon = 1.0/gcon;
  ahem = fabs(*(vals+4))/(*(vals+4));
  deg = (90.0-fabs(*(vals+4)))*d2r;
  cn1 = sin(deg);
  cn2 = radius*cn1*ogcon;
  deg = deg*0.5;
  cn3 = tan(deg);
  deg = (90.0-fabs(*vals))*0.5*d2r;
  cn4 = tan(deg);
  rih = cn2*pow((cn4/cn3),gcon);
  deg = (*(vals+1)-*(vals+6))*d2r*gcon;
  xih = rih*sin(deg);
  yih = -rih*cos(deg)*ahem;
  deg = (90.0-grdlat*ahem)*0.5*d2r;
  cn4 = tan(deg);
  rrih = cn2*pow((cn4/cn3),gcon);
  check = 180.0-*(vals+6);
  alnfix = *(vals+6)+check;
  alon = grdlon+check;
  while (alon<0.0) alon = alon+360.0;
  while (alon>360.0) alon = alon-360.0;
  deg = (alon-alnfix)*gcon*d2r;
  x = rrih*sin(deg);
  y = -rrih*cos(deg)*ahem;
  *grdi = *(vals+2)+(x-xih)/(*(vals+7));
  *grdj = *(vals+3)+(y-yih)/(*(vals+8));
}
```

NMC Eta model (unstaggered grids)

The NMC eta model "native" grid is awkward to work with because the variables

are on staggered (e.g., the grid for winds is not the same as the grid for mass points) *and* non rectangular (number of points in i is *not* constant with j) grids. Because any contouring of irregularly gridded data involves interpolation at some point, NMC creates "unstaggered" eta model fields for practical application programs such as GrADS. In the unstaggered grids all variables are placed on a common *and* rectangular grid (the mass points).

Wind rotation has also been added so that vector data will be properly displayed.

The pdef card for a typical eta model grid is:

```
pdef 181 136 eta.u -97.0 41.0 0.38888888
0.37037037
```

181 = #pts in x
136 = #pts in y
eta.u = eta grid, unstaggered
-97.0 = lon of ref point (E is positive in GrADS, W is negative) [deg]
41.0 = lat of ref point [deg]
0.3888 = dlon [deg]
0.37037 = dlat [deg]

The source code in GrADS for the lon,lat -> i,j mapping is:

```
void ll2eg (int im, int jm, float *vals,  float grdlon, float
grdlat,
      float *grdi, float *grdj, float *alpha) {

/*  Subroutine to convert from lat-lon to NMC eta i,j.

    Provided by Eric Rogers NMC; converted to C 3/29/95 by Mike
Fiorino.

c                SUBROUTINE: ll2eg
c
c                PURPOSE: To compute i- and j-coordinates of a
specified
c                         grid given the latitude and longitude
points.
c                         All latitudes in this routine start
c                         with -90.0 at the south pole and
increase
c                         northward to +90.0 at the north pole.
The
c                         longitudes start with 0.0 at the
Greenwich
c                         meridian and increase to the east, so
```

```
that
c                        90.0 refers to 90.0E, 180.0 is the
inter-
c                        national dateline and 270.0 is 90.0W.
c
c                INPUT VARIABLES:
c
c   vals+0         tlm0d: longitude of the reference center point
c
c   vals+1         tph0d: latitude of the reference center point
c
c   vals+2         dlam:  dlon grid increment in deg
c
c   vals+3         dphi:  dlat grid increment in deg
c
c
c                  grdlat: latitude of point (grdi,grdj)
c
c                  grdlon: longitude of point (grdi,grdj)
c
c                  grdi:   i-coordinate(s) that this routine will
generate
c                          information for
c
c                  grdj:   j-coordinate(s) that this routine will
generate
c                          information for
c

*/

   float pi,d2r,r2d, earthr;   float tlm0d,tph0d,dlam,dphi;
   float
    phi,lam,lame,lam0,phi0,lam0e,cosphi,sinphi,sinphi0,cosphi0,si
nlam r,cos
    lamr;
   float x1,x,y,z,bigphi,biglam,cc,num,den,tlm,tph;

   int idim,jdim;

   pi=3.141592654;

   d2r=pi/180.0;
   r2d=1.0/d2r;
   earthr=6371.2;

   tlm0d=-*(vals+0); /* convert + W to + E, the grads standard
for
    longitude */
   tph0d=*(vals+1);
   dlam=(*(vals+2))*0.5;
   dphi=(*(vals+3))*0.5;
```

```
   /* grid point and center of eta grid trig */

   /* convert to radians */

   phi    = grdlat*d2r;
   lam    = -grdlon*d2r; /* convert + W to + E, the grads
standard for
    longitude */
   lame   = (grdlon)*d2r;

   phi0   = tph0d*d2r;
   lam0   = tlm0d*d2r;
   lam0e  = ( 360.0 + *(vals+0) )*d2r;

   /* cos and sin */

   cosphi = cos(phi);
   sinphi = sin(phi);

   sinphi0 = sin(phi0);
   cosphi0 = cos(phi0);

   sinlamr=sin(lame-lam0e);
   coslamr=cos(lame-lam0e);

   x1     = cosphi*cos(lam-lam0);
   x      = cosphi0*x1+sinphi0*sinphi;
   y      = -cosphi*sin(lam-lam0);
   z      = -sinphi0*x1+cosphi0*sinphi;

   /* params for wind rotation alpha */

   cc=cosphi*coslamr;
   num=cosphi*sinlamr;
   den=cosphi0*cc+sinphi0*sinphi;

   tlm=atan2(num,den);

   /* parms for lat/lon -> i,j */

   bigphi = atan(z/(sqrt(x*x+y*y)))*r2d;
   biglam = atan(y/x)*r2d;

   idim = im*2-1;
   jdim = jm*2-1 ;

   *grdi  = (biglam/dlam)+(idim+1)*0.5;
   *grdj  = (bigphi/dphi)+(jdim+1)*0.5;
   *grdi  = (*grdi+1)*0.5-1;
   *grdj  = (*grdj+1)*0.5-1;
```

```
   *alpha = asin( ( sinphi0*sin(tlm)) / cosphi ) ;

/*   printf("qqq %6.2f %6.2f %6.2f %6.2f %g %g %g %g\n",
       grdlon,grdlat,*grdi,*grdj,*alpha,tlm*r2d,cosphi,sinphi0);
*/

}
```

NMC high accuracy polar stereo for SSM/I data

The polar stereo projection used by the original NMC models is not very precise because it assumes the earth is round (eccentricity = 0). While this approximation was reasonable for coarse resolution NWP models, it is inadequate to work with higher resolution data such as SSM/I.

Wind rotation has not been implemented!!! Use only for scalar fields.

```
pdef ni nj pse slat slon polei polej dx dy sgn
```

`ni` = # points in x
`nj` = # points in y
`slat` = absolute value of the standard latitude
`slon` = absolute value of the standard longitude
`pse` = polar stereo, "eccentric"
`polei` = x index position of the pole (where (0,0) is the index of the first point vice the more typical (1,1))
`polej` = y index position of the pole (where (0,0) is the index of the first point vice the more typical (1,1))
`dx` = delta x in km
`dy` = delta y in km
`sgn` = 1 for N polar stereo and -1 for S polar stereo

Source code in GrADS for the lon,lat -> i,j mapping:

```
void ll2pse (int im, int jm, float *vals, float lon, float lat,
       float *grdi, float *grdj) {

   /* Convert from geodetic latitude and longitude to polar
stereographic
      grid coordinates.  Follows mapll by V. J. Troisi.
*/
   /* Conventions include that slat and lat must be absolute
values */
   /* The hemispheres are controlled by the sgn parameter */
```

```
/* Bob Grumbine 15 April 1994. */

const rearth = 6738.273e3;
const eccen2 = 0.006693883;
const float pi = 3.141592654;

float cdr, alat, along, e, e2;
float t, x, y, rho, sl, tc, mc;
float slat,slon,xorig,yorig,sgn,polei,polej,dx,dy;

slat=*(vals+0);
slon=*(vals+1);
polei=*(vals+2);
polej=*(vals+3);
dx=*(vals+4)*1000;
dy=*(vals+5)*1000;
sgn=*(vals+6);

xorig = -polei*dx;
yorig = -polej*dy;

/*printf("ppp %g %g %g %g %g %g
%g\n",slat,slon,polei,polej,dx,dy,sgn);*/

cdr   = 180./pi;
alat  = lat/cdr;
along = lon/cdr;
e2    = eccen2;
e     = sqrt(eccen2);

if ( fabs(lat) > 90.)  {
  *grdi = -1;
  *grdj = -1;
  return;
}
else {
  t = tan(pi/4. - alat/2.) /
    pow( (1.-e*sin(alat))/(1.+e*sin(alat)) , e/2.);

  if ( fabs(90. - slat) < 1.E-3) {
    rho = 2.*rearth*t/
  pow( pow(1.+e,1.+e) * pow(1.-e,1.-e) , e/2.);
    }
    else {
      sl = slat/cdr;
      tc = tan(pi/4.-sl/2.) /
      pow( (1.-e*sin(sl))/(1.+e*sin(sl)), (e/2.) );
      mc = cos(sl)/ sqrt(1.-e2*sin(sl)*sin(sl) );
      rho = rearth * mc*t/tc;
    }

    x = rho*sgn*cos(sgn*(along+slon/cdr));
```

```
        y = rho*sgn*sin(sgn*(along+slon/cdr));

        *grdi = (x - xorig)/dx+1;
        *grdj = (y - yorig)/dy+1;

        /*printf("ppp (%g %g) (%g %g %g) %g
      %g\n",lat,lon,x,y,rho,*grdi,*grdj);*/

        return;
    }

}
```

CSU RAMS Oblique Polar Stereo Grids

The CSU RAMS model uses an oblique polar stereo projection. This projection is still being tested...

```
pdef 26 16 ops 40.0 -100.0 90000.0 90000.0 14.0
9.0 180000.0 180000.0
```

26 = #pts in x
16 = #pts in y
ops = oblique polar stereo
40.0 = lat of ref point (14.0, 9.0)
-100.0 = lon of ref point (14.0, 9.0 (E is positive in GrADS, W is negative)
90000.0 = xref offset [m]
90000.0 = yref offset [m]
14.0 = i of ref point
9.0 = j of ref point
180000.0 = dx [m]
180000.0 = dy [m]

Wind rotation has not been implemented!!! Use only for scalar fields.

Source code in GrADS for the lon,lat -> i,j mapping:

```
void ll2ops(float *vals, float lni, float lti, float *grdi, float
*grdj)
   {

   const float radius = 6371229.0 ;
```

```
   const float pi = 3.141592654;

   float stdlat, stdlon, xref, yref, xiref, yjref, delx , dely;

   float plt,pln;
   double pi180,c1,c2,c3,c4,c5,c6,arg2a,bb,plt1,alpha,
   pln1,plt90,argu1,argu2;

   double hsign,glor,rstdlon,glolim,facpla,x,y;

   stdlat = *(vals+0);
   stdlon = *(vals+1);
   xref = *(vals+2);
   yref = *(vals+3);
   xiref = *(vals+4);
   yjref = *(vals+5);
   delx = *(vals+6);
   dely = *(vals+7);

   c1=1.0 ;
   pi180 = asin(c1)/90.0;

/*
c
c     set flag for n/s hemisphere and convert longitude to <0 ;
360>
   interval
c
*/
   if(stdlat >= 0.0) {
     hsign= 1.0 ;
   } else {
     hsign=-1.0 ;
   }
/*
c
c     set flag for n/s hemisphere and convert longitude to <0 ;
360>
   interval
c
*/
   glor=lni ;
   if(glor <= 0.0) glor=360.0+glor ;
   rstdlon=stdlon;
   if(rstdlon < 0.0) rstdlon=360.0+stdlon;

/*
c
c     test for a n/s pole case
c
*/
   if(stdlat == 90.0) {
```

```
     plt=lti ;
     pln=fmod(glor+270.0,360.0) ;
     goto l2000;
   }

  if(stdlat == -90.0) {
     plt=-lti ;
     pln=fmod(glor+270.0,360.0) ;
     goto l2000;
   }

/*
c
c     test for longitude on 'greenwich or date line'
c
*/
   if(glor == rstdlon) {
     if(lti > stdlat) {
       plt=90.0-lti+stdlat;
       pln=90.0;
     } else {
       plt=90.0-stdlat+lti;
       pln=270.0;;
     }
     goto l2000;
   }

   if(fmod(glor+180.0,360.0) == rstdlon) {
     plt=stdlat-90.0+lti;
     if(plt < -90.0) {
       plt=-180.0-plt;
       pln=270.0;
     } else {
       pln= 90.0;
     }
     goto l2000;
   }

/*
c
c     determine longitude distance relative to rstdlon so it
belongs to
c     the absolute interval 0 - 180
c
*/
   argu1 = glor-rstdlon;
   if(argu1 > 180.0) argu1 = argu1-360.0;
   if(argu1 < -180.0) argu1 = argu1+360.0;

/*
c
c     1. get the help circle bb and angle alpha (legalize
```

```
arguments)
c
*/

   c2=lti*pi180 ;
   c3=argu1*pi180 ;
   arg2a = cos(c2)*cos(c3) ;
   if( -c1 > arg2a ) arg2a = -c1 ; /* arg2a = max1(arg2a,-c1)  */
   if(  c1 < arg2a ) arg2a = c1 ; /* min1(arg2a, c1)         */
   bb = acos(arg2a) ;

   c4=hsign*lti*pi180 ;
   arg2a = sin(c4)/sin(bb) ;
   if( -c1 > arg2a ) arg2a = -c1 ; /* arg2a = dmax1(arg2a,-c1) */
   if(  c1 < arg2a ) arg2a = c1  ; /* arg2a = dmin1(arg2a, c1) */
   alpha = asin(arg2a) ;
/*
c
c     2. get plt and pln (still legalizing arguments)
c
*/
   c5=stdlat*pi180 ;
   c6=hsign*stdlat*pi180 ;
   arg2a = cos(c5)*cos(bb) + sin(c6)*sin(c4) ;
   if( -c1 > arg2a ) arg2a = -c1 ; /* arg2a = dmax1(arg2a,-c1) */
   if(  c1 < arg2a ) arg2a = c1  ; /* arg2a = dmin1(arg2a, c1) */
   plt1   = asin(arg2a) ;

   arg2a = sin(bb)*cos(alpha)/cos(plt1) ;

   if( -c1 > arg2a ) arg2a = -c1 ; /* arg2a = dmax1(arg2a,-c1) */
   if(  c1 < arg2a ) arg2a =  c1 ; /* arg2a = dmin1(arg2a, c1) */
   pln1   = asin(arg2a) ;

/*
c
c    test for passage of the 90 degree longitude (duallity in
pln)
c         get plt for which pln=90 when lti is the latitude
c
*/
   arg2a = sin(c4)/sin(c6);
   if( -c1 > arg2a ) arg2a = -c1 ; /* arg2a = dmax1(arg2a,-c1) */
   if(  c1 < arg2a ) arg2a =  c1 ; /* arg2a = dmin1(arg2a, c1) */
   plt90 = asin(arg2a) ;

/*
c
c         get help arc bb and angle alpha
c
*/
   arg2a = cos(c5)*sin(plt90) ;
```

```
   if( -c1 > arg2a ) arg2a = -c1 ; /* arg2a = dmax1(arg2a,-c1) */
   if(  c1 < arg2a ) arg2a =  c1 ; /* arg2a = dmin1(arg2a, c1) */
   bb    = acos(arg2a) ;

   arg2a = sin(c4)/sin(bb) ;
   if( -c1 > arg2a ) arg2a = -c1 ; /* arg2a = dmax1(arg2a,-c1) */
   if(  c1 < arg2a ) arg2a =  c1 ; /* arg2a = dmin1(arg2a, c1) */
   alpha = asin(arg2a) ;

/*
c
c         get glolim - it is nesc. to test for the existence of
solution
c
*/
   argu2  = cos(c2)*cos(bb) / (1.-sin(c4)*sin(bb)*sin(alpha)) ;
   if( fabs(argu2) > c1 ) {
     glolim = 999.0;
   } else {
     glolim = acos(argu2)/pi180;
   }

/*
c
c
     modify (if nesc.) the pln solution
c
*/
   if( ( fabs(argu1) > glolim && lti <= stdlat ) || ( lti >
stdlat ) ) {
     pln1 = pi180*180.0 - pln1;
   }
/*
c
c     the solution is symmetric so the direction must be if'ed
c
*/
   if(argu1 < 0.0) {
     pln1 = -pln1;
   }
/*
c
c     convert the radians to degrees
c
*/
   plt = plt1/pi180 ;
   pln = pln1/pi180 ;

/*
c
c     to obtain a rotated value (ie so x-axis in pol.ste. points
east)
```

```
c     add 270 to longitude
c
*/
   pln=fmod(pln+270.0,360.0) ;

 l2000:

/*
c
c     this program convert polar stereographic coordinates to x,y
ditto
c     longitude:   0 - 360  ; positive to the east
c     latitude : -90 -  90  ; positive for northern hemisphere
c     it is assumed that the x-axis point towards the east and
c     corresponds to longitude = 0
c
c     tsp 20/06-89
c
c     constants and functions
c
*/
   facpla = radius*2.0/(1.0+sin(plt*pi180))*cos(plt*pi180);
   x = facpla*cos(pln*pi180) ;
   y = facpla*sin(pln*pi180)  ;

   *grdi=(x-xref)/delx + xiref;
   *grdj=(y-yref)/dely + yjref;

   return;

}
```

Pitfalls when using preprojected data

There are a few *gotchas* with using preprojected data:

1. the units in the variable definition for the `u` and `v` components **must** be `33` and `34K` (the GRIB standard) respectively, e.g.,

   ```
   u 15 33   u component of the wind at 15 pressure levels
   v 15 34   v component of the wind at 15 pressure levels
   ```

2. wind rotation is handled for polar stereo (N and S) preprojected data, but *not* for Lambert Conformal, as the Navy rotates the winds relative to earth. This will have to be added later.....
3. the `eta.u` **and** `ops` projection are still experimental...

GrADS Display Projections

Now that you hopefully understand GrADS data grids, it is time to discuss display projections. Graphics in GrADS are calculated relative to the internal GrADS data grid `i,j` space, transformed to the display device coordinates (e.g., the screen) and then displayed. That is, the i,j of the graphic element is converted to `lat/lon` and then to `x,y` on the screen via a map projection.

GrADS currently supports four `display projections`:

- lat/lon (or spherical);
- N polar stereo (`set mproj` nps);
- S polar stereo (`set mproj` sps);
- the Robinson projection (set lon -180 180, set lat -90 90, set mproj robinson).

As you can probably appreciate, the i,j-to-lon/lat-to-screen x,y for `lon/lat` displays is very simple and is considerably more complicated for N and S `polar stereo` projections.

In principle, a Lambert Conformal display projection could be implemented. It just takes work and a simple user interface for setting up that display projection. Actually, the user interface (i.e., "set" calls) is the most difficult problem...

Summary and Plans

GrADS handles map projections in two different ways. The first is preprojected data where the fields are *already* on a projection (e.g., Lambert Conformal). It is fairly straightforward to implement other preprojected data projections and we will be fully implementing the NMC eta grid both staggered and unstaggered, "thinned" gaussian grids and the CSU RAMS oblique polar stereo projection. The second is in how i,j graphics (calculated in "grid" space) are displayed on a map background. Currently, only a few basic projections (lon/lat, polar stereo and robinson) are supported, but perhaps the development group will tackle this problem.

Variable Formats and Binary Data File Structure

This section describes how to refine the variable declarations in the data descriptor file to accurately reflect the structure and format of each variable in a binary file. Before continuing, it is recommended that you review the material in these other sections:

- About GrADS Gridded Data Sets
- Elements of a GrADS Data Descriptor File

In a GrADS data descriptor file each variable declaration record has the following syntax:

```
varname levs units description
```

The VARS section of Elements of a GrADS Data Descriptor File explains the general syntax of the variable declaration record. This section goes into further detail on the use of the *units* keyword to invoke some special features that allow GrADS to read binary files that do not conform to the default structure.

The structure of a 3-D or 4-D data set is determined by the order in which the horizonal grids are written to file. The default sequence goes in the following order starting from the fastest varying dimension to the slowest varying dimension: longitude (X), latitude (Y), vertical level (Z), variable (VAR), time (T).

If your binary data set was created or "packed" according to a different dimension sequence, then you can use the *units* keyword to tell GrADS exactly how to unpack the data. The *units* keyword is actually a series of one or more comma-delimited numbers. If *units* is set to 99 then all the features for unpacking special data formats are ignored. If *units* is set to -1, then the features are invoked via additional parameters that follow the -1 and are separated by commas:

```
units = -1, structure <,arg>
```

There are four options for *structure*, outlined below. Some of these options have additional attributes which are specified with *arg*.

1. *units* = -1,10,1

   This option indicates that "VAR" and "Z" have been transposed in the dimension sequence. The order is: longitude (X), latitude (Y), variable (VAR), vertical level (Z), time(T). Thus, all variables are written out one level at a time.

   This feature was designed to be used with NASA GCM data in the "phoenix"

format. The upper air *prognostic* variables were transposed, but the *diagnostic* variables were not. Thus an *arg* of 1 means the variable has been var-z transposed, and an *arg* of 2 means the variable has not.

2. *units* = -1,20

   This option indicates that "VAR" and "T" have been transposed in the dimension sequence. The order is: longitude (X), latitude (Y), vertical level (Z), time(T), variable (VAR). Thus, all times for one variable are written out in order followed by all times for the next variable, etc.

   Suppose your data set is actually a collection of separate files that are aggregated by using a template. Then you must use an additional argument to tell GrADS how many time steps are contained in each individual file. Use *arg* to tell GrADS the size of the time dimension in each individual file. For example, here are the relevant records from a descriptor file for 10 years of monthly wind component and temperature data packaged with "VAR" and "T" dimensions transposed:

```
DSET ^monthlydata_%y4.dat
OPTIONS template
...
TDEF 120 linear jan79 1mo
VARS 3

  u 18 -1,20,12 u component
  v 18 -1,20,12 v component
  t 18 -1,20,12 temperature
ENDVARS
```

3. *units* = -1,30

   This option handles the cruel and unusual case where X and Y dimensions are transposed and the horizontal grids are (lat,lon) as opposed to (lon,lat) data. This option causes GrADS to work very inefficiently because it wasn't worth it to make a big change to GrADS internal I/O to handle this type of pathological data. However, it is useful for initial inspection and debugging and that's basically what it is designed for.

4. *units* = -1,40

   This option handles non-float data. Data are converted to floats internally after they are read from the binary file. The dimension sequence is assumed to be the default. The secondary *arg* tells GrADS what type of data values are in the binary file:

units = -1,40,1	= 1-byte unsigned chars (0-255)
units = -1,40,2	= 2-byte unsigned integers
units = -1,40,-2	= 2-byte signed integers
units = -1,40,4	= 4-byte integers

Display Topics

Drawing Data Plots

The `display` command is how you actually display data (output expressions) plots via the graphics output window. The command is:

`display` *expression*

or

`d` *expression*

The simplest *expression* is a variable abbreviation.

If you display when **all** dimensions are fixed, you get a **single** value which is typed out.

If you display when **one** dimension varies, you get a **1-D line graph** by default.

If you display when **two** dimensions are varying, you get a **2-D contour plot** by default.

A variety of plot types are available in addition to the above defaults.

Clearing the Display

GrADS will overlay the output from each display command. To clear the display, enter:

`clear` (or just `c`)

Issued without parameters, the `clear` command does pretty heavy duty clearing of many of the GrADS internal settings. Parameters can be added to limit what is cleared when using more advanced features, for example:

`c events` flushes the events buffer (e.g., mouse clicks)
`c graphics` clears the graphics, but **not** the widgets
`c hbuff` clears the display buffer when in double buffer mode

WARNING: If you make any error in the syntax of clear then GrADS does the full clear...

Graphics Output Types

Before you can display a graph of your data, you will need to set the type of plot you want and, probably, some other graphics parameters as well.

By default, when one dimension varies, you get a line graph, and when two dimensions vary, you get a contour plot. These defaults can be changed by the command:

`set gxout` *graphics_type*

Some examples of *graphics_type* are `contour, shaded, grid, bar, vector,` or `streamline`. For a complete list, see the reference page.

There are many options that can be set to control how the data will be displayed for each *graphics_type*.

For the graphics output types `vector, stream,` and `barb`, the plotting routines need two result grids, where the first result grid is treated as the U component, and the second result grid is treated as the V component. These two result grids are provided to the display command by entering two expressions separated by a semicolon:

```
display u ; v
display ave(u,t=1,t=10) ; ave(v,t=1,t=10)
```

For the graphics output types `vector` and `stream`, you can specify a third result grid that will be used to colorize the vectors or streamlines:

```
display u ; v ; mag(u,v)
display u ; v ; hcurl(u,v)
```

For a graphics output type `wxsym`, each value at a station location is assumed to be a wx symbol code number. To see a chart of all available wx symbols and their corresponding code numbers, run the sample script `wxsym.gs`.

Animation

There are two different ways to animate images within GrADS.

1. Set the dimension environment to have three varying dimensions and then `display` a variable. GrADS will return an **animation sequence**. By default, the animation dimension is time, but you may specify a different dimension to animate by using the following command:

   `set loopdim` x|y|z|t

   If you wish to animate a variable with fewer than three varying dimensions (i.e., animate a line graph), you can control animation by entering:

   `set looping` on|off

   Remember to `set looping off` when you are done animating, or you will get a surprise when you display your next expression!

2. Use double buffering, which means you have two display windows, one of which is always in the background. Double buffering is invoked with the following command:

   `set dbuff` on

   When you issue a display command after turning on double buffering, the image is drawn to the backgound buffer. Then you issue the `swap` command, and GrADS swaps the background and foreground buffers so you can see what you've displayed. `swap` works like `clear` in that it resets many graphics options. Here is a sample script demonstrating how to use double buffering:

```
'open model.ctl'
'set dbuff on'
t = 1
'set gxout shaded'
while (t <= 5)
  'set t 't
  'draw title Temperature'
  'd t'
  'cbarn'
  'swap'
  t = t + 1
endwhile
```

You may also control the speed of the animation by inserting a `q pos` following the `swap` command -- then each click of the mouse would move to the next time step.

Page Control in GrADS

Real and virtual pages

The "real" page is an 8.5x11 page in the landscape or portrait orientation. The orientation is specified when you first startup grads. The graphics output window is a representation of the real page on your computer screen. The graphics window can be any size at all. You can set the dimensions explicitly (in pixel coordinates) using the set xsize command, or you can simply resize the graphics window using your mouse. When it comes time to print the contents of the graphics window to a real page, the screen coordinates (in pixels) will be scaled so the graphics will fit in the real page in the same way they fit in the graphics window.

The "virtual" page is a page-within-a-page that fits within the limits of the real page. By default, the virtual page is the same as the real page, so that real page coordinates are exactly the same as virtual page coordinates. All graphics are drawn on the virtual page. The limits of the virtual page may be changed by using the following command:

```
set vpage xmin xmax ymin ymax
```

After entering a `set vpage` command, GrADS will return the size of the virtual page in inches. Any arguments to graphics commands that require page coordinates in inches are to be given in virtual page coordinates. For example, to draw a plot in the lower left quadrant of the real page, use the following command:

```
set vpage 0 5.5 0 4.25
```

GrADS will return the following virtual page dimensions:

```
Virtual page size = 11 8.5
```

If the virtual page has the same aspect ratio as the real page, Grads will give it the same dimensions as the real page -- in this case the virtual page is a mini version of an 11"x8.5" page. Here's another example where the virtual page is a centered square:

```
set vpage 4 7 2.75 5.75
```

GrADS will return the following virtual page dimensions:

```
Virtual page size = 8.5 8.5
```

On the *real page* the plot will be within a 3" square, but on the *virtual page* in Grads the plot will be within an 8.5" square. Remember that any arguments to graphics commands that require page coordinates in inches are to be given in virtual page coordinates.

To return to the default state where the virtual page equals the real page, enter:

```
set vpage off
```

Controlling the plot area

It is possible to control the area within the virtual page where GrADS draws contour plots, maps, or line graphs. The command is:

```
set parea xmin xmax ymin ymax
```

This area does not include axis labels, titles, color bars, etc., so be sure to provide for adequate margins in order to see these features. Note that the plot area is specified in terms of virtual page units.

GrADS chooses an appropriate default plotting area depending on the type of graphics output. To return to this default, enter:

```
set parea off
```

Line graphs and contour plots that don't contain a map will be scaled to fill the entire plot area. Any plot that contains a map projection will be scaled to fit within the plotting area while maintaining a correct lat/lon aspect ratio. Thus, the map may not fill the entire plotting area except under certain lat/lon ranges. This feature may be turned off by setting the map projection to "scaled". See the reference page for set mproj for additional map projection options.

Drawing Multi-Panel Plots

For drawing multi-panel plots, use `set vpage` to define several virtual pages that fit within the limits of the real page. Virtual pages may overlap. The sample script called `panels_demo.gs` demonstrates how to set up virtual page coordinates for a multi-panel plot with a specified number of rows and columns. It uses a GrADS script function called `panels.gsf`.

If you want to place a label or some other graphic element in each panel, the position is

given in virtual page coordinates. These coordinates will be the same no matter which panel you're in. This makes it easy to shift the labels in one direction or another to accomodate the graphics.

Do not use `set parea` to draw multiple plots on one page. That is not what `parea` was designed for. It is far better (and easier!) to use the `set vpage` command as described above.

Controlling Colors in GrADS

The GrADS Default Colors

GrADS is built with 16 default colors that are used in a variety of applications. Every color in GrADS has a unique *color number* that is used as an index to identify it in GrADS commands. Complete specifications of the default colors numbered 0 to 15 are given below:

Col#	Description	Sample	R	G	B	
0	background		0	0	0	(black by default)
1	foreground		255	255	255	(white by default)
2	red		250	60	60	
3	green		0	220	0	
4	dark blue		30	60	255	
5	light blue		0	200	200	
6	magenta		240	0	130	
7	yellow		230	220	50	
8	orange		240	130	40	
9	purple		160	0	200	
10	yellow/green		160	230	50	
11	medium blue		0	160	255	
12	dark yellow		230	175	45	
13	aqua		0	210	140	

```
14    dark purple      130   0 220
15    gray             170 170 170
```

Disclaimer: The color samples may not be displayed properly.

The GrADS Default Rainbow Sequence

GrADS creates a default rainbow palette using the following sequence of 13 built-in colors:

When drawing contour plots, the default behaviour of GrADS is to color code the contours and select an appropriate contour interval so that each contour is a different color and the colors span the range of the default rainbow sequence. The same principle is behind the selection of default contour intervals for filled contours and shaded grid plots.

The scripts "cbar.gs" and "cbarn.gs" will draw a color key alongside a plot of filled contours or shaded grid cells; the script uses the query shades command to get information about the contour levels and their color shades.

Defining new colors

For some types of displays, the 16 GrADS default colors may not be suitable or adequate. It is possible for the user to define new colors using the set rgb command:

```
set rgb color# R G B
```

For example, let's create a palette of colors for plotting anomalies. We need to define new colors that will be shades of blue and red that range in intensity from fully saturated to very light. White will be the color in the center of the new anomaly palette.

```
* These are the BLUE shades
set rgb 16   0   0 255
set rgb 17  55  55 255
set rgb 18 110 110 255
set rgb 19 165 165 255
set rgb 20 220 220 255
* These are the RED shades
set rgb 21 255 220 220
set rgb 22 255 165 165
set rgb 23 255 110 110
set rgb 24 255  55  55
```

```
set rgb 25 255   0   0
```

Overriding the Defaults

Now that we have a set of newly defined colors (numbered 16-25), we can override the defaults and specify our anomaly palette with exact contour levels and the colors that go with them. This is accomplished by using the following commands:

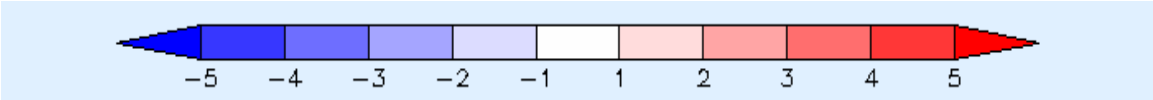
```
set clevs lev1 lev2 lev3 ... levN
set ccols col1 col2 col3 ... colN
```

Contour levels and the colors that go with them are reset with every execution of clear or display. Thus, it may be easier to use these commands in a script to avoid typing them over and over again.

Filled Contours or Shaded Grids: If you are specifying the levels and colors for filled contours (set gxout shaded) or shaded grid cells (set gxout grfill), then the number of colors specified with set ccols must be one larger than the number of contour levels specified with set clevs. Continuing with our example of creating an anomaly palette, the commands would have the following syntax:

```
set gxout shaded
set clevs -5 -4 -3 -2 -1 1 2 3 4 5
set ccols 16 17 18 19 20 1 21 22 23 24 25
```

Note the "0" contour level has been omitted, but color number "1" is still in the palette. Drawing a plot with these specified clevs and ccols and then running the "cbarn.gs" script will result in the following color key:



Here is example using 6 colors and 5 contour levels that shows how the filled contours (or shaded grids) relate to the data values:

```
col1:          values <= lev1
col2:  lev1 < values <= lev2
col3:  lev2 < values <= lev3
col4:  lev3 < values <= lev4
col5:  lev4 < values <= lev5
col6:  lev5 < values
```

Line Contours: If you are specifying the levels and colors for line contours (`set gxout contour`), then the number of arguments to set clevs and set ccols should be equal -- one color for each contour.

Plotting Contours of Constant Color

It is sometimes preferable to plot line contours without the rainbow coloring. An example might be a plot with sea level pressure contours in one color (red) and 500 mb height contours overlaid in another color (blue). For drawing all the contours in the same color, use the `set ccolor` command:

```
set gxout contour
set ccolor 2
d slp
set ccolor 4
d z(lev=500)
```

Omitting Colors

The default behavior of GrADS when plotting filled contours or shaded grid cells is to colorize all areas. To omit a particular color (or contour level) from the plot, simply assign the background color. For example:

```
set gxout shaded
set clevs -5 -4 -3 -2 -1 1 2 3 4 5
set ccols 0 17 18 19 20 0 21 22 23 24 0
```

This example is similar to the one given above, but notice where some of the `ccols` have been set to "0" (the background color). The first, last, and middle colors have been omitted. These commands set up a plot that will only shade areas where the anomalies are between 1 and 5 and -1 and -5. The remaining areas will be black.

Plotting Non-Continuous Index Grids

Plotting grids with index values or non-continuous data (e.g. surface type classification) is simplified by using the graphics output type `fgrid` and the `set fgvals` command.

```
set gxout fgrid
set fgvals 1 15 2 5 3 1
d sfctype
```

In this example, the variable `"sfctype"` has three values: `1` represents land, `2`

represents oceans, and 3 represents sea ice. These commands would draw a plot with land grid cells filled with color number 15 (gray), ocean grid cells filled with color number 5 (light blue), and sea ice grid cells filled with color number 1 (white). If the first two arguments to set fgvals were omitted, then the land grid cells would not be omitted and only ocean and sea ice grid cells would be colored.

Font File Format

GrADS currently supports fonts numbered 0 to 9. Fonts 0 to 4 are provided with the GrADS distribution. The files are named `font0.dat`, `font1.dat`, etc. If you create a `font5.dat` file (or `6` through `9`) and put it in the same place as the other font files, you can use that font immediately, by `set font` 5, or via font escape sequences.

The font files are in an ASCII format. Each file is 95 records in length. Each record represents a character in the font, and the records are ordered in the order of ASCII code values, starting with the blank (code value 32 decimal). So the last record represents the tilde, the 17th record the zero, etc. So when you are using the font the file represents, and enter a zero character, the character in the 17th record is plotted, whatever it may be.

Each record starts with a 3 character number. This is the number of code pairs in the record. This number is followed by the indicated number of code pairs. Each code pair is two characters. Each character represents a value, which is determined as an offset from the Capital R (decimal 82). So, if the character is Capital L, it represents a value of -6. If the character is a lower case k, it represents a value of +25.

The first code pair represents the horizontal extent of the character. So the first character of the code pair would be the minimum X, and the 2nd character the maximum X. If I remember correctly, this extent should include white space. This is followed by code pairs that represent drawing positions in X and Y, where the first character is X, and the 2nd Y. A "pen up" is indicated by the code pair " R" (blank, followed by capital R).

You can look at the existing font files for examples. If you look at `font0.dat`, the first record represents the blank. It thus has one code pair, which just represents the width of the blank in the font, thus allocating white space when a blank is encountered. If you look at record `57` (which represents Cap X), you see: `6H\KFY[ RYFK[` Decoding this, you see there are 6 code pairs. The first is the width extent, `H\`, which is -10 to 10. The next two pairs, `KFY[`, are points -7,-12 and 7,9. So a line would be drawn between those two points (appropriate scaled). The next code pair indicates pen up, followed by `YFK[`, which are 7,-12 and -7,9.

You can see the horizontal extent does not match too well with the actual character. I am not quite sure why this is, nor why the character is not centered. This is the way the fonts came, so I assume there are some font design issues involved.

If you want to design your own font, you will need to review the code GraDS uses to actually plot these fonts, which is `gxchpl.c`. I determined scale factors and centering issues by trial and error, and these values are contained in the code.

Producing Hardcopy and Image Output from GrADS

Generating Image Files from GrADS

There are two different GrADS commands that will convert the contents of the graphics window into an image file. The differences between them are the image formats they support and the way they are implemented in GrADS. `printim` is newer and will only work with GrADS version 1.8.

wi

The `wi` command dumps an exact copy of the GrADS graphics window directly to an image file. It supports a large variety of image formats: GIF, BMP, CGM, EPX, FAX, ICO, JPEG, PCS, HDF, and many others. `wi` requires an active connection to an X-server -- it will not work in batch mode. Consult the reference page for details.

printim

The `printim` command produces a PNG or GIF formatted image file based on the current contents of the GrADS metabuffer, which is the stuff displayed in the graphics window, minus any widgets. `printim` will work in batch mode. Consult the reference page for details.

Generating GrADS metafiles

1. Set-up the GrADS metafile

The first step in creating hardcopy image output is to enter the command:

`enable print` *`filename`*

This opens the output file *`filename`* and enables GrADS to direct image information to that file. Any existing contents of *`filename`* will be lost.

2. Display the image

The next step is to display the graphic that you want to print. When you have finished, enter the command:

`print`

GrADS copies the vector instructions used to create the current display into the output file in a GrADS metacode format.

3. Close the GrADS metafile

There are three way to close the output file:

```
disable print
reinit
quit
```

Converting GrADS Metafiles to Postscript

GrADS metacode files may be translated into postscript using the GrADS external utilities `gxps` and `gxeps`. Both utilities will prompt for input and output filenames, unless they are provided on the command line. The input filename should be the file created by the `enable print` command. The output filename can be anything, but a ".ps" extension is conventional. Any existing file with this name will be overwritten. Once the output file is created, you may print it using UNIX print commands. Please consult the references pages for `gxps` and `gxeps` to see all the command line arguments and options.

`gxps` and `gxeps` are not GrADS commands. They must be executed from the UNIX command line, or preceded by a `!` and executed as a shell command from the GrADS command line.

Displaying GrADS Metafiles

GrADS metacode files may be displayed using the GrADS external utility `gxtran`. The input filename should be the file created by the `enable print` command. If the GrADS metafile contains more than one image, `gxtran` will animate them. The animation can be automatic or controlled by the user with carriage returns. Please consult the gxtran reference page to see all the command line arguments and options.

Displaying GrADS Metafiles with Windows 95/NT

The GrADS metafile Viewer (GV) allows you to view and manipulate GrADS graphics output files using Windows 95/NT. There are two files to download:

- gv32.exe
- gv32.hlp

To open the metafile simply double click on a file listed in the File Manager or Explorer, drag and drop the file onto GV, or use the standard *Open* dialog box. GV assumes that default extension of GRADS metafiles is GMF. If your file includes more than one picture you can browse through pages using the keyboard keys (PageDown and PageUp) or the toolbar buttons.

Use the *View* commands and the *View/Options* dialog box to customize the image -- display it as black-and-white or color, change the line thickness, or clip and enlarge any part of the image. Use the right mouse button to access the most commonly used features.

There are two ways to save separate pages of a GRADS metafile as Windows Metafile (WMF): 1) use the *File/Save Page As* command, or 2) use the *Edit/Copy* command to copy the current page to the Windows Clipboard and then *Edit/Paste* it in your favorite Windows application that handles Windows Metafiles.

Use *File/Print* command to print a current document **to any printer** (you do not need a Postscript printer). Use *File/Print Preview* to display the active metafile as it would appear when printed.

GrADS Scripting Language

Introduction to GrADS Scripts

Scripts offer users the facility to program GrADS operations. Although it is relatively easy for users to produce sophisticated GrADS graphics without ever writing a script, there are occasions where the programming capability makes things even easier. This section explains the general capabilities of scripts, how to run them, and suggests a strategy for users who may wish to write their own.

What scripts can do

The GrADS scripting language, used via the GrADS `run` command, provides a similar capability to the `exec` command, except that scripts also have flow control, defined variables, and access to GrADS command output. Scripts may be written to perform a variety of functions, such as allowing a user to interact with Grads via point and click interface, animating any desired quantities, and annotating plots with information obtained from GrADS query commands.

The scripting language is similar to REXX in implementation. All variables are of type STRING. Mathematical operations are supported on script variables. Flow control is achieved via `if/else/endif` and `while/endwhile` constructs. Loop flow may be modified by the `continue` or `break` commands. Strings contained in variables or generated via an expression may be issued to GrADS as commands. The output from those commands (i.e., the text that GrADS would have output to the terminal) is put into a variable and made available to the script. The language includes support for functions.

Before writing your own scripts, it is recommended that you read the rest of this section and then try to run some of the scripts in the library. Study these example scripts, referring to this page for information on syntax etc., and you will soon be equipped to write scripts of your own.

Running scripts

The command to execute a script is the `run` command:

> `run` *filename &ltarguments>*

This command runs the script contained in the named file, which generally has a `".gs"` tag at the end. Optional *arguments* are passed to the script as a string variable. You may issue any GrADS command from a script, including the `run` command. When calling scripts recursively, be sure that you can back out of the recursion and return to your main script.

Automatic script execution

You may have a simple script automatically executed before every `display` command:

> `set imprun` *script-name*

This script would typically be used to set an option that by default gets reset after each `display` command, for example:

> `set grads` off

You can issue any GrADS command from this script, but the interactions are not always clear. For example, if you issued a `display` command from this script, you could easily enter an infinite recursion loop.

The argument to the script is the expression from the `display` command.

Storing GrADS scripts

It is convenient to put all your GrADS "utility" scripts in one directory (e.g., `/usr/local/grads/lib/scripts`).

To simplify running these scripts, GrADS first looks in the current directory for the script and then, if it can't find it, appends the "`.gs`" extension and tries again. For example, suppose you are working on a test script called `t.gs`. You would run it in GrADS by,

```
run t
```

If after the first two tries, the script still can't be located, then GrADS looks in the directory defined by the environment variable `GASCRP`. In the `t(csh)`, for example,

```
setenv GASCRP /home1/grads/lib
```

or in `ksh`,

```
export GASCRP=/home1/grads/lib
```

Note the if the `/` is not added to the end of the directory name, it is automatically added by UNIX. However, it'll still work if you type

```
setenv GASCRP /home1/grads/lib/
```

If the script cannot be found, then `.gs` is appended and GrADS tries yet again. Thus,

```
d slp
run /home1/grads/lib/cbarn.gs
```

simplifies to,

```
d slp
run cbarn
```

Elements of the Language

A script file is made up of records. The end of a script record is determined by either a newline character or a semicolon (where the semicolon is not contained within a constant string).

Each script record may be one of the following types:

- comment
- statement
- assignment
- say / prompt / pull
- if / else / endif
- while / endwhile / break / continue
- function header / return

Many of the above record types will contain expressions. Script expressions are composed of operands and operators. Operands are strings constants, variables, or function calls; operators are mathematical, logical, or concatenation operations. Further discussion of these record types and the expressions they may contain is given below.

Comment

Comments in GrADS scripts must contain an asterisk (*) in the first column.

Statement

The statement record consists only of an expression:

```
expression
```

The expression is evaluated, and the resulting string is then submitted to GrADS as a command for execution. The script variable `rc` will contain the return code from the GrADS command (this will always be an integer value). In addition, any text output from GrADS in response to the command is put in the variable `result` for examination by the script. A GrADS error resulting from an invalid command WILL NOT terminate execution of the script.

The simplest type of expression is a string constant, which is just a character string enclosed in single or double quotes. Here's an example of simple script containing a comment plus statements comprised of string constants:

```
* this is a sample script
'open my_sst_dataset.ctl'
'set lat -30 30'
'set lon 90 300'
'display sst'
```

Assignment

Assignment records are used to define variables and assign them values. The format of the assignment record is:

```
variable = expression
```

The expression is evaluated, and the result is assigned to be the value of the indicated variable. The same example from above can be rewritten to include assignment statements. Note the use of explicit and implied concatenation:

```
'open my_sst_dataset.ctl'
minlat = -30
maxlat = minlat + 60
minlon = 90
maxlon = 300
'set lat 'minlat%' '%maxlat
'set lon 'minlon' 'maxlon
'display sst'
```

say / prompt

To present information or questions to the GrADS user via the terminal (standard output), use the `say` or `prompt` commands:

```
say expression
prompt expression
```

The result of the *expression* is written to the terminal. The `prompt` command works the same way as the `say` command but does not append a carriage return. It is possible to combine variables and string constants when writing to standard output:

For example:

```
line = "Peter Pan, the flying one"
say line
say `She said it is `line
```

gives:

```
Peter Pan, the flying one
She said it is Peter Pan, the flying one
```

pull

To retrieve information provided by the GrADS user via the terminal (standard input), use the `pull` command:

```
pull variable
```

The script will pause for user input from the keyboard (ending with the carriage return), and then the string entered by the user is assigned to the indicated variable name. To elaborate on a previous example:

```
'open my_sst_dataset.ctl'
prompt 'Enter min and max latitudes: '
pull minlat maxlat
prompt 'Enter min and max longitudes: '
pull minlon maxlon
'set lat 'minlat%' '%maxlat
'set lon 'minlon' 'maxlon
'display sst'
```

if / else / endif

One way to control the flow of script execution is via the `if/else/endif` construct. The format is as follows:

```
if expression
   script record
   script record
   .
   .
else
   script record
   .
   .
endif
```

The `else` block is optional, but the `endif` record must be present. The script records following `if expression` are executed if the expression evaluates to a string containing the character 1. If the expression evaluates to 0, then the script records in the `if` block are not executed and the script continues with the `else` block (if it is present) or the record following `endif`. The `if expression` record must be separated from the script records that follow it. For example, the following script record would be invalid:

```
if (i = 10) j = 20
```

The correct syntax requires three separate script records. This is achieved by putting each record on one line:

```
if (i = 10)
   j = 20
endif
```

Alternatively, the three records could be on the same line separated by a semicolon:

```
if (i = 10) ; j = 20 ; endif
```

N.B. There is no `elseif` construct in GrADS.

while / endwhile

Another method for controlling the flow of script execution is the `while/endwhile` construct. The format is as follows:

```
while expression
   script record
   script record
   .
   .
endwhile
```

The script records following `while expression` are executed if the expression evaluates to a string containing the character 1. If the expression evaluates to 0, then the script records in the `while` block are not executed and the script continues with the record following `endwhile`. The `while expression` record must be separated from the script records that follow it.

Two additional script commands may be used to modify the `while` loop execution: **`break`** and **`continue`**. Inserting the `break` statement will immediately end execution of the loop and the script will move on to the records following `endwhile`. The `continue` statement will immediately end execution of the loop, but the script will then branch immediately back to the top of the loop, and the expression will be re-evaluated.

While loops are often used as counters. For example:

```
count = 1
while (t < 10)
   'set t 'count
   'display z'
```

```
   if (rc != 0) ; break ; endif
   count = count + 1
endwhile
```

Variables

The contents of a script variable is always a character string. However, if the contents of a variable represent a number in the correct format, certain operators may perform numeric operations on that variable, giving a string result which will also be a number.

Variable names can have from 1 to 8 characters, beginning with an alphabetic character and containing letters or numbers only. The name is case sensitive. If a variable has not yet been assigned, its value is its name.

String variables or string constants are enclosed in either single or double quotes. An example of an assignment statement that defines a string variable is as follows:

```
name = `Peter Pan'
name = "Peter Pan"
```

Numeric variables may be entered without quotes, but are still considered strings.

```
number = -99.99
```

Predefined script variables

Some variable names are predefined; it is a good idea to avoid assigning values to these variables. The following are predefined script variables -- their values will change with every execution of a GrADS command from the script:

```
rc
result
```

`lat, lon, lev,` and `time` are also used as predefined variables in GrADS. Although using them within a script is okay, in order to avoid confusion it is not recommended.

Global string variables

String variables are usually local to the functions they are contained in. Global string variables are also available. They are specified via the variable name. Any variable name starting with an underscore (_) will be assumed to be a global variable, and will keep its value throughout an entire script file. An example of an assignment statement that defines

a global string variable is as follows:

```
_var1 = "global variable 1"
```

N.B. Global variables cannot be used in function headers. For example:

```
function dostuff(_var)
```

wouldn't make sense, since `_var` is a global variable, and would be invalid if it were the only argument.

Compound string variables

Compound variables are used to construct arrays in scripts. A compound variable has a variable name with segments separated by periods. For example:

```
varname.i.j
```

In this case, when the variable contents are accessed, `i` and `j` will be looked up to see if they are also variables (non-compound). If they are, the `i` and `j` will be replaced by the string values of `i` and `j`. For example:

```
i = 10
j = 3
varname.i.j = 343
```

In the above example, the assignment is equivalent to:

```
varname.10.3 = 343
```

Note that the string values of `i` and `j` may be anything, but the variable name specification in the script must follow the rules for variable names: letters or numbers, with a leading letter. The variable name after substitution may be any string:

```
i = 'a#$xx'
varname.i = 343
```

The above is valid. However, we cannot refer to this variable name directly:

```
varname.a#$xx = 343
```

would be invalid.

Variable names may *not* be longer than 16 characters, either before or after substitution.

Note that the GrADS scripting language is not particularly efficient in handling large numbers of variables. Thus compound variables should not be used to create large arrays:

```
i = 1
while (i < 10000)
   var.i = i
   i = i + 1
endwhile
```

The above loop will create 10000 distinct variable names. Such a large number of variables in the variable chain will slow the script down a lot.

Operators

The following operators are implemented in the scripting language:

`|`	logical OR
`&`	logical AND
`!`	unary NOT
`-`	unary minus
`=`	equal
`!=`	not equal
`>`	greater than
`>=`	greater than or equal
`<`	less than
`<=`	less than or equal
`%`	concatenation
`+`	addition
`-`	subtraction
`*`	multiplication
`/`	division

The following operators will perform a numeric operation if the operands are numeric:

```
=, !=, >, >=, <, <=, +, -, *, /
```

If any of the following operations are attempted with non-numeric operands, an error will result:

```
+, -, *, /
```

Arithmetical operations are done in floating point. If the result is integral, the result string will be an integer. Logical operations will give a character 0 (zero) if the result is FALSE, and a character 1 (one) if the result is TRUE.

Expressions

Script expressions consist of any combination of operands, operators, and parentheses. Operands may be string constants, variables, or function calls. The precedence of the operators is:

```
-, ! (Unary)
/, *
+, -
%
=, !=, >, >=, <, <=
&
|
```

Within the same precedence level, operations are performed left to right. Parentheses modify the order of operation according to standard convention.

All script expressions, including all function calls, etc. are evaluated and the resulting string is what gets executed as a command. For example:

```
var1 = -1 ; var2 = 10
if (var1*var2 < 10 & var1 > 0)
   say 'both statements are true'
else
   say 'it is not the case that both statements are
true'
endif
```

For the expression following `if`, both sides of the logical operation must be evaluated before the entire expression can be simplified into a true or false result. In this case, the subexpression on the left is true, but the subexpression on the left is not, so the whole expressions resolves to 0 (zero) and the script will print:

```
it is not the case that both statements are true
```

Concatenation

In some espressions, the concatenation operator may be implied. The % operator may be omitted whenever the two operands are a string constant and a variable name. With

implied concatentation, intervening blanks will be ignored.

For example, the following expressions have the same effect:

```
'set lat 'minlat%' '%maxlat        uses the concatenation operator
%
'set lat 'minlat' 'maxlat          concatenation is implied
```

Assuming two previous statements, `minlat = -30` and `maxlat = 30`, the resulting expression would be:

```
'set lat -30 30'
```

Keep in mind the order of precedence when using the concatenation operator.

Functions

Function calls take the form of:

```
name(arg,arg,arg,...)
```

where the function name follows the same rules as for variable names, and the arguments may be any expression. Functions may either be contained within the script file itself, or the may be intrinsic functions. Functions contained within other script files are not supported as yet (other script files may be executed via the GrADS run command).

In either case, functions are invoked as a script expression is being evaluated. Script functions always have a single string result, but may have one or more string arguments. Functions are invoked by:

```
name(arg,arg,arg...)
```

If the function has no arguments, you must still provide the parentheses:

```
name()
```

You may provide your own functions from within your script file by using the `function` definition record:

```
function name(variable, variable, ...)
```

To return from a function, use the `return` command:

`return` *expression*

The *expression* is optional; if not provided, a NULL string will be returned. (A null string is: '') The result of the function is the result of the expression specified on the return command.

When a function is invoked, the arguments are evaluated, then flow of control is transferred to the function. The variables contained in the list within the function definition record are initialized to the values of the passed arguments. If too few arguments where passed for the variables specified, the trailing variables are uninitialized. If too many arguments are passed, the extra arguments are discarded.

You may modify the variables from the function definition record without modifying the variables from the calling routine.

Scope of variables is normally local to the function, but can be global.

When a script file is first invoked (via the `run` command), execution starts at the beginning of the file. A function definition record may optionally be provided at the beginning. If it is, it should specify one variable name. This variable will be initialized to any `run` command options. If no options were given, the variable will be initialized to NULL.

Intrinsic Functions

sublin (*string, n*)
This function gets a single line from a string containing several lines. The result is the *nth* line of *string*. If the string has too few lines, the result is NULL. *n* must be an integer.

subwrd (*string, n*)
This functions gets a single word from a string. The result is the *nth* word of *string*. If the string is too short, the result is NULL. *n* must be an integer.

substr (*string, start, length*)
This function gets part of a string. The sub-string of *string* starting at location *start* for length *length* will be returned. If the string is too short, the result will be short or NULL. *start* and *length* must be integers.

read (*filename*)
This functions reads individual records from file *filename*. Repeated calls must be made to read consecutive records. The result is a string containing two lines:

the first line is the return code, the 2nd line is the record read from the file. The record may be a maximum of 80 characters. Use the sublin function to separate the result. Return codes are:

0 - ok
1 - open error
2 - end of file
8 - file open for write
9 - I/O error

Files are opened when the first call to read is made for a particular file name. Files are closed when the execution of the script file terminates (note that files remain open between function calls, etc).

write (*filename, record* <, append>)

This functions writes records to output file filename. On the first call to write for a particular file, the file is opened in write mode. This will destroy an existing file! If you use the optional append flag, the file will be opened in append mode, and all writes will be appended to the end of the file. Return codes are:

0 - ok
1 - open error
8 - file open for read

close (*name*)

This function closes the named file. This must be done if you wish to read from a file you have been writing to. This can also be used to rewind a file. Return codes are:

0 - ok
1 - file not open

Commands that complement the scripting language

There are some GrADS commands that, although not designed exclusively for scripts, are most useful in script applications. These include:

query <*option*> or q <*option*>

To see the list of available options, issue the query command by itself. A description of the query options that are most useful for script applications follows.

q define -- Lists all defined variables

q defval *ival jval* -- Gives defined grid value at *ival, jval*

To interactively modify grid point values for a `defined` variable, `q defval` can be used in conjunction with `set defval`. For example, the code shown below queries the value of sst at gridpoint(i,j), then tests to see if the value is less than -1.6, and if it is, sets the sst to a bad value.

```
'q defval sst 'i' 'j
val = subwrd(result,3)
if (val < -1.6)
  'set defval sst 'i' 'j' 'bad_value
endif
```

q dims -- Gives the current dimension environment

q file *n* -- Gives info on file number *n*

q files -- Lists open files

q fwrite -- Gives the name of the file used for fwrite operations

q gxinfo -- Lists graphics settings

This option is handy when trying to find the plot area. The output from `q gxinfo` might look like this:

```
Last Graphic = Line
Page Size = 11 by 8.5
X Limits = 2 to 10.5
Y Limits = 0.75 to 7.75
Xaxis = Lon Yaxis = Val
Mproj = 2
```

The first line indicates that the output is a line plot. The second line gives the page dimensions -- in this case GrADS is in landscape mode. The third and fourth lines give the x and y boundaries of the plot. In this case the plot has 1-inch margins in the x direction and 0.75-inch margins in the y direction. The fifth line tells what kind of axes you have, and the sixth line identifies the map projection:

1	Scaled (no preservation of aspect ratio)
2	Latlon (2-D horizontal fields)

3 Northern polar stereographic
4 Southern polar stereographic
5 Robinson (lon range must be -180 to 180 and lat range must be -90 to 90)

q pos -- Waits for mouse click, returns position

q shades -- Gives colors and levels of shaded contours

q time - gives time range of current open file

q *transform coord1 coord2* -- Coordinate transformations

where *transform* is one of:

xy2w	XY coords to world coords
xy2gr	XY coords to grid coords
w2xy	world coords to XY coords
w2gr	world coords to grid coords
gr2w	grid coords to world coords
gr2xy	grid coords to XY coords

XY coords are inches on the page (screen) where the page is 11x8.5 inches or 8.5x11 inches, depending on how GrADS was started.

World coords are lat, lon, lev, time or val, depending on what the dimension environment is when the grid was displayed. Note that time is displayed (and must be specified) in GrADS absolute date/time format. val is the value coordinate for a 1-D plot (linegraph).

Grid coordinates are the i,j indices the grid being displayed. For station data sets, grid and world coordinates are equivalent except for the time dimension. Note that if you display a grid from a 'wrapped' data set, the grid numbers may be out of range of the actual file grid numbers. (A 'wrapped' data set is a data set that covers the earth in the longitude direction. Wrapping takes place automatically). The conversions are done consistently, but you may want to be sure you can handle the wrapping case if your data set is global.

N.B. Coordinate transform queries are only valid after something has been displayed, and the transformations apply only to the most recent item that has been displayed.

set gxout findstn

When using the graphics output type set gxout findstn, three arguments must be provided with the display command. The first argument is a station data expression. The 2nd and 3rd arguments are the X and Y screen coordinates of the of the desired search location. GrADS will find the station closest to the specified X and Y position, and print its stid, lon, and lat. This graphics output type should only be used when X and Y are the varying dimensions and AFTER a regular display command (that results in graphics output) is entered.

set dbuff on|off

This command sets double buffer mode on or off. This allows animation to be controlled from a script. The clear command also sets double buffer mode off.

swap

Swaps buffers, when double buffer mode is on. If double buffer mode is off, this command has no effect.

The usual usage of these commands would be:

```
set dbuff on
start looping
   display something
   swap
endloop
set dbuff off
```

Widgets

GrADS has the capability to implement a graphical user interface. This interface is used to draw widgets (buttons and pull down menus) that allow a "point and click" interface between the Grads user and the scripting language.

Buttons

Here is a sample from a script illustrating how to draw a button:

```
set rgb 90 100 100 100
set rgb 91  50  50  50
set rgb 92 200 200 200
set button 2 90 91 92 3 90 92 91 6
draw button 1 5.5 1 2 0.5 This is a Button
```

The reference pages for set button and draw button contain information on how to specify the button characteristics and position.

A button's initial "state" is ON. If a user clicks on a button following a q pos command, then the button state will switch from ON (1) to OFF (0). A second q pos followed by a mouse click on the button will return it to the ON state. The button state may also be changed with the redraw button command.

The output from the q pos command is what makes the button widgets so useful. Here is a template of what q pos returns after a mouse click on a button:

Position = *xpos ypos mousebutton widgetclass buttonnumber buttonstate*

where:

xpos, ypos	- coordinates of the mouse click in virtual page units
mousebutton	- either 1, 2, or 3 for the left, center, or right mouse button
widgetclass	- 1 is the widget class number for buttons
buttonnumber	- the number assigned to the button when it was originally drawn
buttonstate	- either 0 (meaning "off") or 1 (meaning "on")

If the user did not click on a button, then *widgetclass* will be 0 and there will be no output for *buttonnumber* or *buttonstate*.

Drop Menus

As with button widgets, dropmenus provide a "point-and-click" interface between scripts and the GrADS user. The reference pages for set dropmenu and draw dropmenu contain information on how to specify the dropmenu characteristics and position.

The output from q pos after a click on a dropmenu is similar to that described above for buttons. Here is a template of what is returned by q pos after a mouse click on a dropmenu:

Position = *xpos ypos mousebutton widgetclass menunumber inum*

where:

xpos, ypos	- coordinates of the mouse click in the menu base in virtual

page units

mousebutton - either 1, 2, or 3 for the left, center, or right mouse button

widgetclass - 3 is the widget class number for dropmenus

menunumber - the number assigned to the dropmenu when it was originally drawn

inum - the menu item number selected from the menu list

If no menu item is selected, then *menunumber* and *inum* will both be -1.

Here is a script sample illustrating how to use a dropmenu:

```
'reset events'
'set rgb 90 100 100 100'
'set rgb 91 150 150 150'
'set rgb 92 200 200 200'
'set dropmenu 1 91 90 92 0 91 90 92 1 91 90 92 90 92 6'
'draw dropmenu 1 1 8 1.5 0.5 Select a Variable | Wind | Temperature |
Height | SLP '
noselect = 1
while (noselect)
  'q pos'
  menunum  = subwrd(result,7)
  menuitem = subwrd(result,8)
  if (menunum = 1)
    if menuitem = 1 ; newbase = 'Variable = Wind'   ; endif
    if menuitem = 2 ; newbase = 'Variable = Temp'   ; endif
    if menuitem = 3 ; newbase = 'Variable = Height' ; endif
    if menuitem = 4 ; newbase = 'Variable = SLP'    ; endif
    'draw dropmenu 1 1 8 1.5 0.5 'newbase' | Wind | Temperature |
Height | SLP '
    noselect = 0
  endif
endwhile
```

Here is another script sample illustrating how to use cascading dropmenus:

```
'clear'
'set rgb 90 100 100 100'
'set rgb 91 150 150 150'
'set rgb 92 200 200 200'
'set button 1 91 -1 -1 1 91 90 92 12'
'draw button 1 1 8 1 0.5 quit'
'set dropmenu 1 91 -1 -1 1 91 90 92 1 91 90 92 90 92 6'
'draw dropmenu 1 1.5 7.5 2 0.5  Menu Base | Space | Earth >05> | Sun |
Moon'
'draw dropmenu 5 cascade Ocean | Land | Atmosphere >11> | Biosphere'
'draw dropmenu 11 cascade Snow | Rain | Mist | Tornado '

while (1)
```

```
  'q pos'
  say result
  ev = subwrd(result,6)
  if (ev!=3); break; endif;
endwhile
```

It is left to the GrADS script writer (that means you!) to run the demo and interpret the output of q pos when clicking on all the options in the cascade of dropmenus.

Rubber banding

GrADS has a widget type called rband for rubber banding. There are two rband modes: box and line. To set up the rband widget, use the following command:

set rband *num mode x1 y1 x2 y2*

where:

num	- widget number
mode	- may be either box or line
x1	- lowest X point where the widget will be active (in virtual page units)
y1	- lowest Y point where the widget will be active (in virtual page units)
x2	- highest X point where the widget will be active (in virtual page units)
y2	- highest Y point where the widget will be active (in virtual page units)

In box mode, as the user clicks and drags the mouse in the active rband area a box is drawn with one corner located at the initial click and the opposite corner located at the release point. In line mode, a line is drawn between these two points.

For example, suppose you want to set up a box rubber band widget in the plot region only.

First, execute q gxinfo to get the X and Y limits of the plot area. The result from q gxinfo might look like this:

```
Last Graphic = Line
Page Size = 11 by 8.5
X Limits = 2 to 10.5
Y Limits = 0.75 to 7.75
Xaxis = Lon  Yaxis = Val
Mproj = 2
```

Second, set up the widget with set rband using the dimensions grabbed from the result of q gxinfo:

```
xlims = sublin(result,3)
ylims = sublin(result,4)
x1 = subwrd(xlims,4)
x2 = subwrd(xlims,6)
y1 = subwrd(ylims,4)
y2 = subwrd(ylims,6)
'set rband 21 box 'x1' 'y1' 'x2' 'y2
```

Finally, use q pos to activate the widget.

```
ga-> q pos
```

This freezes the system until the user clicks, drags, and then releases the mouse somewhere within the active rband area. Here is a template for the output you would get from GrADS after a mouse click and drag in the rband area:

```
Position = xpos1 ypos1 mousebutton widgetclass
widgetnumber xpos2 ypos2
```

where:

xpos1, ypos1	- coordinates of the initial mouse click in virtual page units
mousebutton	- either 1, 2, or 3 for the left, center, or right mouse button
widgetclass	- 2 is the widget class number for rbands
widgetnumber	- the number assigned to the rband widget when it was set up
xpos2, ypos2	- coordinates of the mouse release point in virtual page units

The page coordinates can be then be used to draw a box (or a line) where the user specified, or parsed and used in the coordinate transform q xy2w to recover the lat/lon region selected by the user.

Dynamic Loading of Script Functions

Script variables are generally local to the functions (scripts) they are contained in; they exist in memory only while the function is executing. If a variable name starts with an underscore (_), then it becomes a *global* script variable and keeps its value throughout the execution of the main script file. The drawback to global variables was that the functions they are defined in had to be included in the main script file. With a new capability that comes with GrADS version 1.8, that is no longer the case.

Dynamic loading of script functions means that when your main script calls a function (subscript), all global variables from that function will be retained in memory and can continue to be used by main script. Commonly used functions do not have to be copied into every script that calls them.

The names of GrADS script functions may now have 16 characters and include the underscore, although they still must begin with an alphabetic character. Grads script function names are case sensitive.

Error messages will include the file name where the error ocurred -- in this case, the full path name of the file that was actually opened.

Using GrADS Script Functions

The tricks to using GrADS script functions are (1) to enable dynamic loading and (2) to teach GrADS how to find the script functions you call from your main script.

To enable the dynamic loading of script functions, put this at the top of your script:

```
rc = gsfallow("on")
```

To teach Grads how to find the script functions you call from your main script, you must first know how Grads searches for main script file names.

How GrADS searches for main script file names

Let's assume the user wants to run a script called `do_eof.gs` and gives the Grads command:

```
ga-> do_eof
```

Grads will search in the currently directory for the script name, as provided. If it doesn't find the script, it appends .gs to the script name and tries again. If the script is still not found, then the environment variable GASCRP is examined. GrADS attemps to open the

script file in all the directories listed in the GASCRP variable, first with the provided name, then with the .gs appended.

If GASCRP contains "/usr/local/gradslib /usr/homes/myhome", then GrADS will search for the script do_eof in the following order:

- do_eof
- do_eof.gs
- /usr/local/gradslib/do_eof
- /usr/local/gradslib/do_eof.gs
- /usr/homes/myhome/do_eof
- /usr/homes/myhome/do_eof.gs

GrADS uses the first file it finds. Once found, the directory that contains the script file is remembered as the "main function prefix".

How GrADS searches for script function file names

Continuing with our example, let's further assume that GrADS encounters a function in do_eof.gs that is not included in the stuff already loaded from the main script file. GrADS will look for a .gsf file to load, using the following search path:

- <main-function-prefix>/<function-name>.gsf
- <main-function-prefix>/<private-paths>/<function-name>.gsf
- <GASCRP-paths>/<function-name>.gsf

The private path directory list is an optional list that is provided via the gsfpath function:

```
rc = gsfpath("dirlist")
```

If used, the declaration of the private path directory list should appear at the top of the main script just underneath the statement enabling the dynamic script loading.

For example, if our main script "do_eof.gs" is executed with the command:

```
run /usr/local/gradslib/do_eof
```

and this script file contains the following lines at the front:

```
rc = gsfallow("on")
rc = gsfpath("math1 string2")
```

and the script calls a function `str_chop` which is not found in the main script, then the search path would be:

1. /usr/local/gradslib/str_chop.gsf
2. /usr/local/gradslib/math1/str_chop.gsf
3. /usr/local/gradslib/string2/str_chop.gsf

GrADS Script Library

basemap.gs	Overlays a land or ocean mask that exactly fits the coastal outlines. Requires the following supplemental data files: lpoly_lowres.asc and lpoly_mres.asc and lpoly_hires.asc opoly_lowres.asc and opoly_mres.asc and opoly_hires.asc See instructions in script header for using lpoly_US.asc to mask out non-US areas.
cbar.gs and cbarn.gs	Scripts to draw a long rectangular color legend next to shaded plots. cbarn has more features -- e.g. it draws outlines and triangular endpoints.
cbarc.gs	Draws a small fan-shaped color legend in the corner of shaded plots.
cbar_l.gs and cbar_line.gs	Scripts to draw a legend for line graphs.
cmap.gs	Creates a color table. See additional documentation.
connect_the_dots.gs	Draws a line connecting user's mouse clicks.
define_colors.gs	Defines a variety of colors using the set rgb command.
defval_demo.gs	Illustrates the use of q defval and set defval commands.
font.gs	Displays all the characters in a font set.
isen.gs	Displays a field interpolated to a specified isentropic level.
lats4d.gs	Writes NetCDF, HDF-SDS or GRIB files from GrADS. See additional documentation.
makebg.gs	Creates a background map image that shows topographic texture. It requires a DODS-enabled version of GrADS and also uses the external ImageMagick utility "combine".
map.gs	Automates settings for a variety of useful map projections.
mconv.gs	Calculates moisture convergence.

meteogram_eta.gs meteogram_gfs.gs meteogram_gfsb.gs	These scrips draw meteograms based on NCEP forecast data, which is accessed through the GrADS-DODS Server. You must use a DODS-enabled version of GrADS for these scripts to work.
panels.gsf and panels_demo.gs	Scripts to create global variables that contain the 'set vpage' commands for a multi-panel plot. These also illustrate the dynamic loading of script functions.
script_math_demo.gs	Illustrates the use of the scripting language math funcitons.
pinterp.gs	Displays a field interpolated to a specified pressure level.
plotskew.gs	Draws a SkewT/LogP Diagram.
stack.gs	Delays display while command sequence is entered by user, then executes all at once.
string.gs	Draws a string located at position of user's mouse click.
sweat_index.gs	Calculates the SWEAT Index given relative humidity, temperature, and wind components.
traj.gs	Draws forward and backward trajectories in the horizontal plane.
use.gs	Similar to open except it checks the list of current files to see if the specified file has already been opened. If it has been opened, it changes the default file number to the previously opened file instead of opening the same data file again.
wxsym.gs	Displays available weather symbols.
xanim.gs	Controls an animated display.
zinterp.gs	Displays a field interpolated to a specified height level.
zoom.gs	A simple way to zoom into a plot.

T

U

V

W

X

Y

Z