User Guide welcome

Welcome to the NCAR Graphics Version 4.1 User Guide. The NCAR Graphics functionality and methods of use have changed dramatically since the last major release (Version 3.2) of the software. For NCAR Graphics Version 4.0/4.1, the user documentation is published as a hypertext information system. This system provides faster, easier, and more widely available access to the documentation than printed manuals. Also, document updates will be done online so you can always access the most up-to-date version of the documentation available. If you start using the roadmaps, index, and tables of contents, you will get the most out of this hypertext information system. In all User Guide modules, the Roadmap and Contents navigation tools show User Guide contents only.

Along with the traditional Fortran and C low level interfaces (LLUs), NCAR Graphics 4.1 software has expanded to include a high-level object-oriented interface (High Level Utilities or HLUs) that is also callable from Fortran and C, and a command language interface (the NCAR Command Language or NCL)

With version 4.1 of NCAR Graphics, we are pleased to introduce a new document to help you get started using NCL, called *Getting started using NCL*. It is intended for both beginning and advanced users of NCL.

With version 4.1 of NCAR Graphics, we are also pleased to introduce the new Ngmath package. The Ngmath library is a collection of interpolators for one-dimensional, two-dimensional, and three-dimensional data for which there are equivalent Fortran, C, and NCL entries.

If you are seeking information about using the traditional interface (Fortran and C LLUs), you should start with the *NCAR Graphics Fundamentals* document and the *NCAR Graphics Contouring and Mapping Tutorial* document. Another useful LLU document explains NCAR Graphics' implementation of the Graphical Kernel System standard: *User's Guide for NCAR GKS-0A Graphics*. These documents were created for print and have been converted to hypertext format; you can browse the online versions by clicking on the preceding titles (note that they will not transfer across the network as efficiently as the documents that have been created for hypertext). You can also print the documents from the PostScript files:

- Fundamentals is located in the path "fund/fundhome.ps.Z"
- Tutorial is located in the path "conmaptutor/conmaptutor.ps.Z"
- GKS-0A is located in the path "gks/gks.ps.Z"

How to get started

If you are a new NCAR Graphics user, or if you want to learn how to use the HLUs or NCL, we recommend starting with the Quick Start Guide and this User Guide.

If you are looking for examples of how to create certain types of plots, then we recommend that you use the Quick Start Guide for an overview of the various plot types and examples that are included with the NCAR Graphics software. You can also access the Quick Start Guide by clicking on the "Examples" button at the beginning and end of any module.

The User Guide (this document) provides a system overview of the software architecture, the interfaces, their functionality and features, how to set up your work environment, how to build and compile an application, and how to use the HLUs and the NCL. It also explains the key concepts underlying the NCAR Graphics 4.1 software and helps you understand how to use it.

Document organization

This User Guide is organized into three main sections that describe each of the methods for using the NCAR Graphics software: the High Level Utilities (HLUs), the NCAR Command Language (NCL), and the traditional Low Level Utilities (LLUs).

The modules in each section are typically short and concise because they each offer a general overview of just a few concepts. Much greater detail is provided in the NCAR Graphics Version 4.1 Reference Manual. (Please note that the LLU sections were designed for print and are less effective in the hypertext medium.)

User Guide table of contents

User Guide credits

Acknowledgments

Authors: Ethan Alpert, Brian Bevirt, Dave Brown, Jeff Boote, Fred Clare, Mary Haley, Dave Kennison, Bob Lackman, Jacque Marshall, Tim Scheitlin, Ed Stautler

Editor: Brian Bevirt

Production: Jacque Marshall

Copyright

Trademarks

Published by

National Center for Atmospheric Research, Scientific Computing Division, P.O. Box 3000, Boulder, CO 80307-3000. The National Center for Atmospheric Research is operated by the University Corporation for Atmospheric Research and is sponsored by the National Science Foundation. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the National Science Foundation.

User Guide overview

This module covers a range of topics from a description of the software architecture to practical explanations of how to set up your user environment for using the software. This module provides information about the software structure and points to modules that can help you decide where to start or which areas of NCAR Graphics to explore.

System components

NCAR Graphics provides three general user interfaces that are described in the system elements module. These interfaces are: Low Level Utilities, High Level Utilities, and NCAR Command Language. The User Guide modules for these interfaces are aimed at users who want to learn about the NCAR Graphics software structure and how to use the software to develop applications and generate graphics.

Selecting an interface

The Choosing an interface module describes the differences between the different NCAR Graphics Version 4.1 user interfaces (LLUs, HLUs, NCL). This module also explains the advantages of using each interface and provides the criteria for deciding which interface best suits the user skill level and the user application.

Software and hardware requirements

The Software and hardware requirements module describes the computing resources you need to use NCAR Graphics.

Setting up your work environment

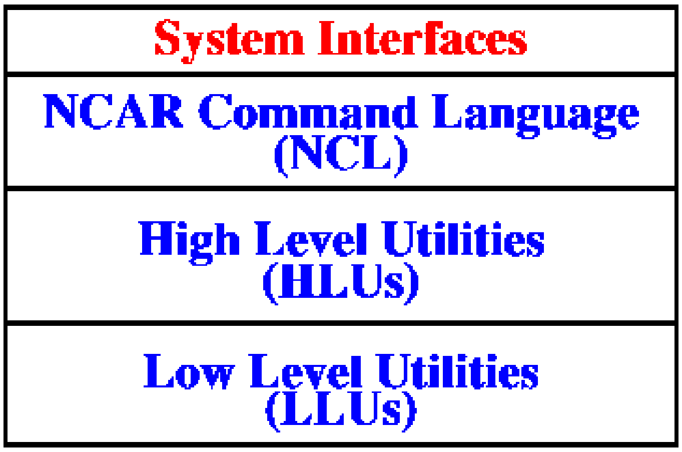
The General user environment module explains how to set up your work environment, shell variables, path names, and resource files to run an NCAR Graphics program, build an NCAR Graphics application, and use the NCAR Graphics software.

Reference tables

The Reference tables module organizes a great deal of reference information for quick access: resource names, HLU APIs, and NCL functions and procedures.

System elements

The NCAR Graphics software is logically divided into several system interfaces: the NCAR Command Language (NCL) interpreter, the High Level Utilities (HLUs), and the Low Level Utilities (LLUs). Each of these components represents a user interface to the software.



Much of the new NCAR Graphics software is built upon the LLUs, which comprise the traditional NCAR Graphics package. Developers can build an NCAR Graphics application by making calls to the NCAR Graphics Fortran and C low level libraries. This interface consists of over two dozen utilities for contouring, mapping, drawing field flows, drawing surfaces, drawing histograms, drawing XY plots, labeling, and more. Currently, this interface contains the richest set of functionality of all the interfaces.

The HLUs are an object-oriented Fortran and C interface that provide a consistent programmatic interface to the software. HLUs manage the graphics state of NCAR Graphics on your behalf, making all function calls to the traditional software in the correct order and managing all internal work arrays and common blocks. This design makes it unnecessary for you to understand and control most of the low-level implementation details of the traditional software. By reducing the complexity of traditional NCAR Graphics, the HLUs make it easier for you to learn the software at the programming level. New users who want to develop NCAR Graphics applications are encouraged to start with this interface.

The NCL is a user interface that provides the option of creating plots and manipulating arrays of data from the command line using an interpreted language rather than from within a compiled Fortran or C program. NCL allows you to import and export data in a variety of formats (ascii, binary, HDF, netCDF) and to manipulate that data with the NCL commands. Interactive data manipulation includes selecting region-of-interest data, thinning data, interpolating irregular grids to regular grids, performing curve fits, removing corrupted data, and performing useful transformations. You will also be able to customize the methods NCL uses to ingest and manipulate data. As a tool for specifying visualizations, NCL allows you to generate 1D, 2D, and 3D plots in many styles with many options for annotating, labeling, and animating the visualizations.

Choosing an interface

This module describes the differences between the NCAR Graphics user interfaces (LLUs, HLUs, and NCL). This module also indicates the advantages of using each interface and provides the criteria for deciding which interface to use depending on the user skill level and user application.

LLUs

The Low Level Utilities are the traditional Fortran and C functional interface to NCAR Graphics. Currently these utilities have the greatest flexibility and the richest set of functionality available for users. This interface has not changed much since the 3.2 release of NCAR Graphics. If you are not a new user, and you are accustomed to using this interface, then you might want to continue using the LLUs until the full suite of NCAR Graphics functionality is available through the HLUs, NCL, and the GUI (coming in NCAR DataVision) .

For a complete description of how to learn and use the LLUs and examples of the types of plots you can create, see the Low Level Utilities module.

HLUs

The High Level Utilities are built on top of the LLUs, and they provide a consistent Fortran and C interface to the NCAR Graphics functionality. If you are a new user beginning to learn NCAR Graphics, we recommend that you start with the HLUs. Unlike the LLUs, the HLUs are more consistent in their approach to each graphics utility; once you learn how to create one type of plot, you will have a much easier time learning how to create other types of plots. Although the HLU functionality is currently limited, NCAR Graphics developers are concentrating on extending the HLU interface. In the future, the HLUs will be the primary mechanism recommended for users who want to write NCAR Graphics applications using a Fortran or a C interface.

Also, HLUs use resource files to set plot attributes like color, line thickness, position, etc. This allows developers to quickly change a plot's characteristics without having to edit and recompile any code. Tweaking the appearance of a plot is simply a matter of editing an ascii resource file, supplying a new value for a plot resource, and rerunning your code.

HLUs are also designed to aid GUI developers in their efforts to create graphical interfaces for NCAR Graphics applications. If you want to add this type of enhancement to your NCAR Graphics programs, then we recommend that you use the HLU library.

For a complete description of how to learn and use the HLUs, see the High Level Utilities module.

NCL

The NCAR Command Language is an interactive, text-based language interpreter that is built on top of the HLU interface. It allows you to manipulate data and create NCAR Graphics plots without having to write and compile any C or Fortran code. No C or Fortran programming expertise is needed to use NCL, although knowledge of common programming constructs and techniques will be very helpful for writing NCL scripts.

NCL contains a rich set of math functions that allow you to read and process your data dynamically from a command line interface or from within a script if you are using NCL in batch mode.

NCL also provides a mechanism for specifying plot characteristics so you can create NCAR Graphics plots by typing commands rather than developing and compiling a C or Fortran program. NCL has built-in defaults for quick and accurate visualization of data. The defaults can also be customized by the user via a user defaults file. Furthermore, visualization specifications can be entered and altered at the command line.

For more information about learning how to use NCL, see the NCAR Command Language module.

General user environment

This module describes how to set up your work environment, shell variables, path names, and resource files to build Low Level Utility and High Level Utility applications, and run the NCAR Command Language.

Environment variables

Before you can use NCAR Graphics, you must make sure that you have set either NCARG_ROOT or the three environment variables NCARG_BIN, NCARG_LIB, and NCARG_INCLUDE, depending on how NCAR Graphics was installed on your system. In some cases, the installer may have set either NCARG_ROOT or NCARG_BIN, NCARG_LIB, and NCARG_INCLUDE globally so that you don't have to worry about it. To figure out if any of these variables have been set, execute:

```
env
```

A list of all the environment variables that have been set will be shown, along with their values.

If you don't have either NCARG_ROOT, or NCARG_BIN, NCARG_LIB, and NCARG_INCLUDE set, then please contact your system administrator.

Path names

You must have the NCAR Graphics binaries in your execution path. If you have the NCARG_ROOT environment variable set, then the location of these files is $NCARG_ROOT/bin, otherwise the location is $NCARG_BIN.

You must add one of these directories to your execution path in order to use the NCAR Graphics software. Contact your system administrator if you need help in adding a directory to your execution path.

Where to go from here

After you have set up the NCAR Graphics environment variables and path names described earlier, you are ready to use the NCAR Graphics software, build Low Level Utility and High Level Utility applications, and run the NCAR Command Language.

Using the LLUs

The NCAR Graphics Low Level Utilities currently contain the richest set of programming interfaces for developers to use to create NCAR Graphics applications. The LLUs consist of both a C and the traditional Fortran interface for creating graphics applications and visualizing scientific data. For more information about using the LLUs and creating an LLU application, see the Low Level Utilities module.

Using the HLUs

The High Level Utilities are a library of tools that provide a consistent programming interface to NCAR Graphics functionality. The HLU design makes it unnecessary for you to understand and control most of the low-level implementation details of the traditional NCAR Graphics software.

The HLUs exploit the benefits of resource files. Resource files are ASCII files that specify resources for controlling the characteristics of your NCAR Graphics plots. These files make it easy to control default plot attributes without having to recompile a program or reset the properties that you want to remain constant in all your NCAR Graphics applications. For more detailed information about setting up resource files in your user environment, see the HLU user environment module.

Using NCL

The NCAR Command Language is a higher level user interface that is built on top of the High Level Utilities. NCL is an interactive scripting language; it works in a fashion similar to a Unix shell scripting language. Before you can use the NCAR Command Language, you must set up your NCAR Graphics environment variables and path names as described earlier.

For more information about running NCL, see the NCL user environment module.

Software and hardware requirements

This module describes the software and hardware requirements for using NCAR Graphics 4.1.

Supported systems

The NCAR Graphics software is supported on several computational platforms listed below. These systems require at least 32 MB of main memory, and the X server needs to be configured with backing store.

The supported systems and their OS versions include:

Cray J90	UNICOS 10.0
Digital Equipment Alpha	Digital UNIX 4.0
HP9000 Series 700	HP-UX 10.20
IBM PC	Linux 2.0.*x*
IBM RISC System/6000	AIX 4.2
Silicon Graphics systems	IRIX 6.2/6.4 (-n32/-64 options)
Sun systems	Solaris 2.5.*x*

The additional software needed to build the NCAR Graphics libraries from source includes:

```
libXaw.a  X11R5

HDF       v4.1rx (optional)
```

Developers who will be creating NCAR Graphics applications and tools will need the following software:

```
ANSI C compiler

FORTRAN 77 compiler  (the Fortran 90 compiler was used
                      on the Cray systems).

libX11.a  X11R5/6

netCDF 3.x  (optional)
```

Documentation for the NCAR Graphics software is available on the World Wide Web (WWW) in the form of HTML documents. You will need to connect to the NCAR server with an HTML browser to view this documentation which *will be kept up-to-date on a continuing basis*.

The HTML documentation is also shipped with the software, along with PostScript copies of all documentation files for sites that want to print a static hardcopy version.

Note: Sites that want to install the HTML documentation locally will first need to install an HTTP server (not included with the distribution).

Reference tables

This module points to many useful reference tables that let programmers using the HLU and NCL interfaces quickly access lots of useful information. These tables organize the information in various ways: by name, by logical groupings, and by index number.

HLU resources

This table of all HLU resource names organizes HLU resources alphabetically by name. This is useful for people who want to learn about resources they find in the examples. This is also useful for experienced programmers who know a resource name but are seeking detailed information about it.

Tables of resources for specific utilities are organized by logical groups in the reference page for each utility. This helps people who are learning about the resources used by each utility. From the alphabetical list of HLU classes, select a utility and scroll to the Resources section, right after the Synopsis. The resource file for the first example of each utility also groups resources this way, and it provides all the default values. You can obtain these resource files and run NCAR Graphics 4.1 examples using the ng4ex command.

HLU types

Table of HLU variable types that are valid for HLU variables. This is useful if you need a complete list of valid types or if you want the definition of any specific type.

HLU functions

Alphabetical listing of all HLU functions.

NCL commands

Table of NCL command names (functions and procedures).

NCAR Graphics 4.x examples

Table of all NCAR Graphics 4.x examples with pointers to source code, resource files, and descriptive explanations for each example.

Fonts

Table of all available type fonts shows the name of each font printed in that font. This table is alphabetized by font name.

Color tables

This table lists the names of all the predefined colormaps and shows each color along with its index number.

Dash patterns

This graph lists all of the predefined dash patterns along with their index numbers.

Fill patterns

This graph shows all of the predefined fill patterns along with their index numbers.

Markers

This graph shows all of the predefined markers along with their index numbers.

MapPlot database table

The MapPlot database table lists the names, group numbers, types, and area identifiers that are used to specify fill attributes and map boundaries in a MapPlot.

High Level Utilities overview

The High Level Utilities (HLUs) are a library of tools that provide a consistent programming interface to NCAR Graphics functionality. Some HLUs are independent of the traditional NCAR Graphics libraries of C and Fortran subroutines, while others make calls into the low level libraries. A goal of the HLUs is to free the user from having to understand the implementation details of the low level libraries. Thus, when the low level libraries are invoked by the HLU, the HLU manages the graphics state, properly initializes and orders the calls to the low level routines, and manages all internal work arrays.

The HLU interface is distinctly different from the LLU interface. The LLU model is the classic programming paradigm. Subroutine calls with selected arguments are stacked in an order that acheives the desired result. The HLU model is object oriented. Objects are created that have a range of action which is determined by the values assigned to a set of object resources. Thus, the functionality of the HLU program depends both on the set of objects created and the set of values assigned to the object resources.

From this module, you can access the following subjects and information about HLUs.

- Overview of HLU library - some general comments about HLUs.
- HLUs vs. LLUs - why you might choose to use one interface or the other.
- Strategy for learning to use HLUs - an approach to learning the HLU interface.
- ng4ex description - an important command for accessing HLU example programs.
- HLU terminology - some object-oriented terms used in HLU documentation.
- Classes and objects - an explanation of class/object relationships.
- Composite classes - a description of composite classes.
- Class hierarchy - a visual display of all parent/child relationships and composite classes in hierarchical diagrams.
- View class - for creating graphical objects.

- Workstation class - for output of graphical objects.
- Data class - for passing data arrays to graphical objects.
- Application class - for connecting objects to an application.
- Error class - for logging and reporting application errors.
- Transform class - for selecting coordinate transformations.
- PlotManager class - for controlling multiple transformations and overlays.
- AnnoManager class - for controlling the placement of annotations.
- Other classes - for special purposes.
- Define resources and how they work
- Three ways to set resources - in the HLU code or in an ASCII file.
- Resource types - functions for setting and getting those types.
- Resource files - for project and application management.
- The HLU Application Programming Interface (API)
- HLU functions - as they relate to the various classes.
- Writing an HLU program - logical steps in creating Fortran or C programs.
- HLU header files - associated with objects to be created.
- How objects are used in programs - the logical function of commonly used objects.
- HLU user environment - needed environment variables for proper installation.
- HLU plot creation sequence - step by step creation of Fortran and C programs.
- Mixing HLUs and LLUs - processes and precautions in merging low level library calls with HLUs.
- HLUs: putting it all together - remembering all the necessary steps.
- Troubleshooting HLUs - common problems which evade error reporting.

Overview of the HLU library

The NCAR Graphics High Level Utilities (HLUs) are an alternate Fortran and C programmatic interface to the Low Level Utilities (LLUs). The HLU interface provides most of the functionality of the approximately 500 user-callable subroutines in the LLUs. It is assumed in this section of the document that you have decided on using a Fortran or C interface. If you have not already considered the NCAR Command Language (NCL) interface, you may want to see if it might be more suitable to your purposes.

The HLU interface was created as one of the necessary components of an integrated interactive graphics package. The graphical capability of NCL is provided through the HLUs. Unlike the LLUs, the HLUs can be configured to allow their graphical output to be modified and regenerated without recompiling code.

The HLUs follow an object-oriented design structure similar to X11 widgets. Like widgets, they use resources and resource files as the way of implementing functionality options. The benefits of following the object-oriented model include:

- Reduction in the amount of redundant code

- Consistency through combined classes

For example, the ContourPlot, StreamlinePlot, VectorPlot and XyPlot classes all use the TickMark class as the source of scaled grids.

If you are in doubt as to whether you want to use HLUs or LLUs, read the HLUs vs. LLUs module.

HLUs vs. LLUs

Reasons why you might want to use the HLU interface

1. Reduced complexity and order dependence

When you use the LLU interface, you may be required to call a number of routines in a prescribed way to accomplish a desired task. This is usually the case with more complex contouring and mapping examples. To determine which subroutines you will need and in which order they must be called, you need to know a lot about the contouring and mapping libraries.

In the HLU case, you only need to know how to assemble an HLU program. This requires that you create a number of objects: one to access the data to be plotted, one to create the plot (in this case the ContourPlot object) and one to tell your program where to send the graphics (a Workstation object). However, this structure is the same in all HLU programs. Once you master the concepts of building an HLU program, you will find that it is relatively straightforward to build other programs with entirely different functionality. Moreover, example HLU programs are available which span the range of HLU functionality. ng4ex describes how the examples can be listed, copied into your working directory, and executed.

2. Integrated interfaces

The LLUs of NCAR Graphics were written and updated at different times by different software developers; thus, the same function in two different utilities, such as setting up a ticked and scaled plot grid, is typically done in different ways. In the case of the HLUs, the TickMark object is used as the source of all 2-D plotting grids.

3. Efficiency

Consider, for example, that you are experimenting with the appearance of a plot in order to get it up to the quality needed for publication. In the LLU case, each time you change the value of any parameter, such as a color, line width, label, or text font, you will need to re-compile, link, load, and run to see the effects of your changes. In the case of the HLUs, you can change the values of resources in a resource file, then re-execute the HLU module and view the new graphic. This process is more efficient in both computer and user time.

4. Functionality

There are some features of the HLUs that are not available in the low level libraries. For example, the XyPlot utility allows you to input a nonlinear data scale for either axis. The data plotted along a nonlinear axis is placed using a spline fit.

Reasons why you might want to use the LLU interface

1. You are already an expert LLU programmer.

If you are adept at programming with the LLUs, you may not want to invest the time necessary to learn a new interface.

The utilities referred to as LLUs in Version 4 of NCAR Graphics are the traditional graphics libraries of Version 3. New features and utilities have been added, but the utilities and tools work in the very same way as they did for Version 3. In fact, any Version 3 user codes should run equally well using the Version 4 libraries. Users can still use the same set of commands and tools including ncargex, ncargcc, and ncargf77. So, if you prefer to use the traditional library utilities such as Conpack, Autograph, or Ezmap, you may continue to do so.

2. The LLUs provide maximum control

The LLUs give you the ultimate user-level control over the software. In order to provide a simplified interface, the HLUs have to make some choices for you.

3. At Version 4.1, all NCAR Graphics functionality is not available through HLUs

Functionality available in LLUs--which is not available in HLUs as of the Version 4.1 release--includes histograms, 3-D wireframe surfaces and isosurfaces, weather maps, and polypack. (Note: it is possible to call the routines in these LLUs from your HLU program.)

Remember that Version 4.1 contains all of the LLU functionality that Version 3.2 had, and more. This functionality can be accessed through Fortran and C programs as described in the LLU documentation whose links are given below. The ncargex command can be used to list, copy, or execute Fortran or C example LLU programs. The comparable HLU command is ng4ex. Man pages exist for both ncargex and ng4ex.

There are also commands for linking your programs with the NCAR Graphics libraries. For linking with the LLU library, use ncargf77 or ncargcc. `ncargf77 myprogram.f` would compile my Fortran program, link it with the LLU libraries, and execute it. `ncargcc myprogram.c` would do the same for my C program. These commands do not link in the HLU libraries.

Similar commands for linking with the HLU library are nhlf77 (Fortran) and nhlcc (C). These commands also link in the LLU Libraries.

Note that the nhlf77 or nhlcc commands can be used to execute LLU programs, but ncargf77 or ncargcc commands cannot be used to execute HLU programs.

For example, cmpgrp.f is an LLU mapping example. It can be executed using either `nhlff cmpgrp.f`, or `ncargf77 cmpgrp.f`. The ncargf77 version will use less memory space since fewer libraries will be linked. However, if you copy the HLU TextItem example, tx03f.f, into your working directory with the command `ng4ex -n tx03f`, and then try to use `ncargf77 tx03f.f` instead of `nhlf77 tx03f.f` to run it, you will get a list of unresolved references in the program load stage.

See also:

- Mixing HLUs and LLUs
- *NCAR Graphics Low Level Utility Fundamentals*
- *NCAR Graphics Low Level Utility Contouring and Mapping Tutorial*

Strategies for learning HLUs

The following steps outline a basic strategy for learning how to use the HLU interface.

Step 1. Do a first reading of the HLU section of the User Guide

To see the logical layout of the HLU section of the User Guide, click on the roadmap icon at the top of this page. You can click on any of the boxes in any roadmap to jump to that module in the User Guide.

In the various sections of the User Guide, you will be introduced to a number of Object Oriented (OO) terms including classes, objects, and resources. When you encounter a new term that is not hypertext-linked to a defining area of the documentation, you can check its meaning by clicking on the glossary icon in any control panel at the top of the module, then searching for the term alphabetically.

As a user of HLUs, you will be creating objects from class (functionality) templates, and manipulating the objects by assigning values to resources. For example, you might create a TextItem object and assign different fonts to the text using the txFont resource. In one instance of a TextItem object, you may use the Courier font, such as to create tables where columns line up because all characters in the Courier font have identical widths. In another instance of a TextItem object, you may want to use a font that was designed for better readability.

Within the HLU section of the User Guide, you will learn what kinds of objects can be created, the resources that apply to each object, and how you can set the values of these resources.

Step 2. Understand the types of classes available and the class hierarchy

If you skimmed this area of the HLU User Guide in Step 1 above, review it again carefully.

Understanding classes and objects describes the types of classes in the HLU library and class relationships that determine how instantiated objects can be manipulated to get the desired results.

For example, there is a View class that provides templates for various graphical objects such as a TextItem or an XyPlot. There is a Workstation class that provides templates for display objects such as an X Window or a file where graphics instructions are stored for later display. In a typical HLU program, you will create objects that define your application, input your data, draw graphics, and determine where the graphics are sent. The chart Class hierarchy diagram shows the hierarchical relationships of the HLU object library.

In a typical HLU program, you will create:

- An App object that will identify the application represented by your HLU program, name your application resource file, and tell where it is located.
- One or more View objects. View objects are used to create the graphics. View objects include Title, TickMark (for grids as well as tick marks), Legend, LabelBar, XyPlot, ContourPlot, StreamlinePlot, VectorPlot, MapPlot, and TextItem.
- One or more Workstation objects. The types of Workstation objects include PSWorkstation for PostScript output, XWorkstation for display in an X Window, and NcgmWorkstation to produce an NCAR-encoded Computer Graphics Metafile (file named with a ".ncgm" extension).
- An optional DataItem object. If you are creating an XyPlot, ContourPlot, StreamlinePlot, or VectorPlot, you will also need to create a DataItem object containing the data to be plotted.

Step 3. Review the resource set of a couple classes

HLU objects use resources like traditional subroutines use parameters or arguments. They allow for the selection of options that define the object behavior. For example, in the case of a graphical object, resources might be used to select line widths, colors, fonts, or other graphical entities.

HLU resources - alphabetically contains an alphabetical list of all resources. The first two characters of a resource name are an identifier of the class to which the resource applies. For example, "am" is for AnnoManager. At the top of the list is a set of class names that link you to the beginning of the resources for that class. At first reading, you might want to look at a couple of the classes with a lesser number of resources like TextItem (prefix "tx") and Workstation (prefix "wk"). The number of resources available to the TickMark, XyPlot, ContourPlot, StreamlinePlot, and VectorPlot classes can be rather daunting at first review. Later, however, when you are creating a finely tailored plot, we hope you will enjoy the detailed level of control these resources provide.

Step 4. Review some example programs

Study some examples using the Quick Start Guide and ng4ex.

These facilities provide you with a full range of HLU programs that use both the Fortran and C HLU interfaces. Study the codes and the example graphics.

Example naming

Examples are named as follows: A two-character prefix designates the graphical object type from this list.

```
Object type            two-character prefix on example names
------------           -------------------------------------
ContourPlot                 cn
GraphicStyle                pr (stands for "primitives")
Legend                      lg
LabelBar                    lb
MapPlot                     mp
```

```
TextItem                    tx
TickMark                    tm
Title                       ti
StreamlinePlot              st
VectorPlot                  vc
XyPlot                      xy
```

Following the two-character object identifier is a two-digit example number. Finally, the fifth character designates the language in which the example is coded, using "f" for Fortran, "c" for C, or "n" for NCL. Thus, "cn03c" indicates ContourPlot example number three coded in C.

Step 5. Write your own HLU program

When you are ready to write your first program, start from an existing example that is similar to what you want to do. There are examples for all of the available graphical objects. You can copy the example code and the example resource file into your working directory using the ng4ex command.

Some useful options include:

```
ng4ex -list -all    list the names of all available examples

ng4ex -list -type   where type is one of: basic, contourplot,
                    graphicstyle, legend, labelbar, mapplot,
                    streamlineplot, textitem, tickmark,
                    title, vectorplot, or xyplot.

ng4ex lg01f         Copies the example lg01f into your working
                    directory, compiles the source code, and runs
                    the code to create an output graphic.

ng4ex -n lg01f      Copies the example lg01f into your working
                    directory without compiling and running the code.
```

Once you have an example to start from, refer to the Writing an HLU program module and start modifying the code to achieve the results you want.

Step 6. Execute your program

Commands are available for compiling and loading your C or FORTRAN-coded HLU programs. These commands are called nhlcc (for C programs) and nhlf77 (for FORTRAN programs).

$NCARG_ROOT is the environment variable that gives the pathname to where NCAR Graphics is installed on your system. See your NCAR Graphics Site Representative or NCAR Graphics Site Installer for help with either creating **make** files, or using the series of commands to run C or Fortran programs that link in all the needed graphics and related libraries.

Once your program has compiled and run your module, either a plot will appear in a window (an XWorkstation object was created) or a graphics file will appear in the directory. If the file is named with a .ncgm suffix, it can be viewed with ctrans. If it has a postscript variant suffix, it can be viewed with a PostScript previewer such as ghostscript, or sent to a PostScript printer.

Step 7. Refine the graphics

Review your graphical output, change the resource values in your code and/or in the resource file, execute the module again, and study the many ways you can refine your output. As you become a more proficient user of HLUs, the content of the HLU User Guide will become more useful to you, and you will find the HLU section of the Reference Manual to be a rich source of detailed information.

See also:

- ng4ex command
- Writing an HLU program
- Troubleshooting an HLU program
- HLU classes - alphabetically
- HLU API - alphabetically

Running NCAR Graphics 4.1 examples with the ng4ex command

The ng4ex command copies, compiles, and runs NCAR Graphics 4.x examples in your local directory. These examples demonstrate how to use the High Level Utilities (HLUs) and NCL to create an NCAR Graphics application. A similar command called ncargex is used to copy, compile, and run NCAR Graphics 3.2 examples that demonstrate how to use the traditional interface (Low Level Utilities--LLUs).

Depending on the example, the output graphic may be displayed in a workstation window or output to a file. More information on the ng4ex command is provided in the ng4ex man page; it is copied here:

```
ng4ex(1Nhl)       Misc. Reference Manual Pages        ng4ex(1Nhl)

NAME
     ng4ex - NCAR Graphics High Level Utilities Example Programs

SYNOPSIS
     ng4ex [-A] [-C] [-Fortran] [-NCL] [-cdf] [-gui] [-anno] [-gsun]
     [-W workstation_type] [-app] [-basic] [-contourplot] [-labelbar]
     [-legend] [-primitives] [-mapplot] [-ngmath] [-streamplot]
     [-textitem]  [-tickmark] [-title] [-vectorplot] [-xyplot]
     [-list] [-clean] [-n] name ...

DESCRIPTION
     ng4ex provides the user with access to several  C,  Fortran,
     and  NCL  examples illustrating the use of the NCAR Graphics
     HLUs (High Level  Utilities)  and  NCL.   ng4ex  copies  the
     source  code  for  the specified example(s) into the current
     directory and then compiles, links, and executes the example
     (if  it's a Fortran or C program) or runs NCL on it (if it's
     an NCL script file).  Depending on the example,  the  output
```

may be sent to an NCGM (NCAR Graphics Metafile) file and/or displayed to an X window on your screen. You must be running X and have your DISPLAY environment set correctly in order for the X window examples to execute properly.

If the requested example creates an NCGM file, it will have the same name as the example, suffixed with ".ncgm". An option allows you to request that only the source code be copied to your directory, without compiling or running the example. Another option allows you to request that only the NCGM file be left in your directory and that all other files created by ng4ex be deleted. The argument name may be selected from the lists that appear below.

Since the HLU library and NCL is available on Suns and SGIs only, these examples will only execute on those machines. In the future, the HLUs, NCL, and their examples will be ported to other systems.

OPTIONS

−A Generate all available examples.

−Fortran
Generate all Fortran examples.

−NCL Generate all NCL examples.

−cdf Generate all the examples that use netCDF files. For the C and Fortran examples, this option requires that you have the netCDF library installed on your system.

−gui Generate all GUI examples. This option requires that you have the Motif libraries on your system.

−anno
Generate all annotation examples.

−W Specify the workstation type. This argument can only be specified as a string. Some examples are meant to display to the X workstation, to an NCGM file, or to a PostScript file only, in which case this option may be ignored. You cannot specify more than one workstation type. If you don’t specify one, then a default one will be used. The following workstation_type strings are valid:

ncgm – NCGM file

x11 – X11 window

ps – PostScript file

−app Generate all the app Fortran, C, and NCL examples that show how to do specific things with resources.

−basic Generate all the basic Fortran, C, and NCL examples that relate to the basic examples discussed in the Quick Start Guide.

-contourplot — Generate all Fortran, C, and NCL ContourPlot examples.

-labelbar — Generate all Fortran, C, and NCL Labelbar examples.

-legend — Generate all Fortran, C, and NCL Legend examples.

-primitives — Generate all Fortran, C, and NCL GraphicStyle examples.

-mapplot — Generate all Fortran, C, and NCL MapPlot examples.

-ngmath — Generate all Fortran, C, and NCL Ngmath examples.

-streamlineplot — Generate all Fortran, C, and NCL StreamlinePlot examples.

-textitem — Generate all Fortran, C, and NCL TextItem examples.

-tickmark — Generate all Fortran, C, and NCL TickMark examples.

-title — Generate all Fortran, C, and NCL Title examples.

-vectorplot — Generate all Fortran, C, and NCL VectorPlot examples.

-xyplot — Generate all Fortran, C, and NCL xyPlot examples.

-gsun — Generate all the GSUN examples (NCL scripts only).

-list — Specifies that the names of the selected examples should be echoed to the screen instead of being copied or generated. This option must be used with one of the other options that selects a particular subset of examples. For example, if you want to see a list of all the available examples, then use -list in conjunction with the -A option. If you want to see a list of all the ContourPlot examples, then use -list with the -contourplot option (and so on).

-clean — Remove everything but the ".ncgm" file.

-n — Specifies that the example should just be copied, and not linked or executed.

name(s) — Name(s) of example programs to generate.

EXAMPLES AVAILABLE

Please notice that the C examples end with the letter 'c', the Fortran examples end with the letter 'f', and NCL examples with the letter 'n'.

ap01c,ap01f,ap01n
Demonstrates various ways to use resource files.

basic01c,basic01f,basic01n
Demonstrates how to draw a contour plot using mostly defaults. No data is used in this example.

basic02c,basic02f,basic02n

Demonstrates how to set the view port and how to produce multiple plots on a single frame.

basic03c,basic03f,basic03n
Demonstrates how to create a scalar data object, set resources using a resource file, set resources during object creation, and set resources after object creation.

basic04c,basic04f,basic04n
Demonstrates how to select and change the workstation device for drawing your output to an NCGM file or an X workstation window.

basic05c,basic05f,basic05n
Demonstrates how to read and manipulate colormaps.

basic06n Demonstrates how to position objects on an output device and how to change their sizes.

basic07f Demonstrates creating three simultaneous workstations.

basic08c,basic08f,basic08n
Demonstrates use of a procedure that limits a plot, including annotations that extend outside the plot viewport, to a pre-defined bounding box within NDC space.

basic09c,basic09f,basic09n
Lists all the fonts with their number, name, and what the font looks like.

cn01c,cn01f,cn01n
Shows all the default settings for the ContourPlot resources using a simple data set.

cn02c,cn02f,cn02n
Demonstrates basic features of the ContourPlot object.

cn03c,cn03f,cn03n
Demonstrates basic features of the ContourPlot object. The first frame emulates the contour plot drawn in cn01.

cn04c,cn04f,cn04n
Emulates the output of the ncargex example "cpex02".

cn05c,cn05f,cn05n
Demonstrates how to create a map plot animation

with a contour overlay and labelbar annotation.

cn06c,cn06f,cn06n
Shows how to read a netCDF file and produce a series of temperature contour plots.

cn07c,cn07f,cn07n
Shows how to read a netCDF file and produce a series of contour plots.

cn08c,cn08f,cn08n
Shows how to read a netCDF file and draw a vertical profiles of temperature for longitude separated by 5 degrees.

cn09c,cn09f,cn09n
Shows how to read a netCDF file and produce a series of surface pressure contour plots.

cn10f,cn10c,cn10n
Demonstrates how to read a netCDF file and produce three contour plots, an xyplot, and a map plot.

cn11c,cn11f
Demonstrates how to combine LLU and HLU calls.

cn12c,cn12f,cn12n
Emulates example "cpex08" and draws a filled contour plot over a map of Africa with a labelbar.

cn13c,cn13f,cn13n
Emulates example "mpex10" and draws a raster contour plot over a map using inverse map transformations.

cn14c,cn14f,cn14n
Shows how to do a contour/map overlay with tick marks showing the lat/lon locations.

cn15c,cn15f,cn15n
Shows how to combine contour plot and xy plot on a single frame. Also shows how to do multiple workstation outputs, and how to position your PostScript output on the page.

cn16c,cn16f,cn16n
Shows how to combine a vector fill contour plot, a raster contour plot, and a map plot on a single frame.

cn17c,cn17f,cn17n
Shows how to restrict a contour plot to a certain area over a satellite projection, how to label the US states with AnnoManager, how to use NhlDataPoly{line,marker} to draw lines/markers on a map projection, and how to mix LLUs/HLUs to draw

text in the map projection. The NCL version doesn't have the LLU/HLU part.

lb01c,lb01f,lb01n
Shows all the default settings for the LabelBar resources.

lb02c,lb02f,lb02n
Shows how to modify some of the default resources to create a solid filled set of boxes.

lg01c,lg01f,lg01n
Shows all the default settings for the Legend resources.

lg02c,lg02f,lg02n
Shows how to modify some of the default resources to create a legend of five markers.

lg03c,lg03f,lg03n
Shows how to create a legend of five lines.

pr01c,pr01f,pr01n
Demonstrates basic GraphicStyle capabilities.

pr02c,pr02f,pr02n
Demonstrates GraphicStyle capabilities and how to some GS resources.

pr03c,pr03f,pr03n
Demonstrates graphics primitives drawn in NDC space and how clipping works.

pr04c,pr04f,pr04n
Demonstrates graphics primitives drawn into an IrregularPlot object.

pr05c,pr05f,pr05n
Demonstrates overlaying graphics primitives on a MapPlot.

mp01c,mp01f,mp01n
Demonstrates basic MapPlot capabilities.

mp02c,mp02f,mp02n
Demonstrates individual control of MapPlot areas.

mp03c,mp03f,mp03n
Demonstrates MapPlot masking; loosely emulates the ncargex example "colcon".

mp04c,mp04f,mp04n
Illustrates the use of AnnoManager objects with MapPlot objects.

mp05c,mp05f,mp05n
Shows how to create the ten different map projections.

nm01c,nm01f,nm01n

Simple example of natural neighbor linear interpolation.

nm02c,nm02f,nm02n
Simple example of natural neighbor linear regridding.

nm03c,nm03f,nm03n
Shows how to retrieve aspects and slopes of an interpolation.

nm04c,nm04f,nm04n
Shows a simple 3D interpolation.

nm05c,nm05f,nm05n
Shows how to vary the exponent of the distances in a simple 2D interpolation..

nm06c,nm06f,nm06n
Smoothing in a simple 2D interpolation.

nm07n Illustrates use of ftcurv, ftcurv, and ftcurvi.

nm08n Illustrates use of ftcurvp and ftcurvpi.

nm09n Illustrates use of ftcurvs and ftcurvps.

nm10n Illustrates use of ftkurv and ftkurvd.

nm11n Illustrates use of ftkurvp and ftkurvpd.

nm12n Illustrates use of ftsurf.

nm13n Illustrates the use of csa1s.

nm14n Illustrates the effect of weighting the input points using csa1xs.

nm15n Illustrates extrapolation into data sparse regions using csa1xs.

nm16n Illustrates two-dimensional approximation and second-order mixed partial using csa2s and csa2xs.

nm17n,nm18n
Illustrates the use of csa3s.

nm19n Illustrates the use of csa2ls.

st01c,st01f,st01n
Demonstrates basic StreamlinePlot example.

st02c,st02f,st02n
Demonstrates line-drawn streamline arrows and some basic resources.

st03c,st03f,st03n
Shows how to overlay a StreamlinePlot on a map projection.

st04c,st04f,st04n
Shows a StreamlinePlot of wind vector data over a

MapPlot.

ti01c,ti01f,ti01n
Shows all the default settings for the Title resources which results in a blank plot.

ti02c,ti01f,ti01n
Turns on the main, x-axis, and y-axis strings to produce a plot with three titles.

ti03c,ti03f,ti03n
Shows how to change resources to get high quality filled and colored titles.

tm01c,tm01f,tm01n
Shows all the default settings for the TickMark

tm02c,tm02f,tm02n
Shows how to modify the resources to create tick marks on the axes.

tm03c,tm03f,tm03n
Shows how to modify the resources to change the scaling and the tick mark labels.

tx01c,tx01f,tx01n
Shows all the default settings for the TextItem resources.

tx02c,tx02f,tx02n
Shows how to modify default resources to produce a text string.

tx03c,tx03f,tx03n
Uses the same resources as example "tx02c", except 114 frames are produced with varying background colors.

tx04c,tx04f,tx04n
Shows how to use the TextItem object of the HLU library.

tx05c,tx05f,tx05n
Demonstrates the TextItem object with text having various heights and various angles.

tx06c,tx06f,tx06n
Demonstrates TextItem text justifications.

tx07c,tx07f,tx07n
Demonstrates TextItem text spacings and aspect ratios.

tx08c,tx08f,tx08n
Simple Annotation example.

vc01c,vc01f,vc01n
Basic VectorPlot example.

vc02c,vc02f,vc02n
Demonstrates line-drawn vector arrows and the use of some basic VectorPlot resources.

vc03c,vc03f,vc03n
Manipulates the FillArrow resources to demonstrate some of the possible stylistic variations on the appearance of filled vector arrows.

vc04c,vc04f,vc04n
Demonstrates how to rotate a VectorPlot 90 degrees while preserving the original relationship of the data elements.

vc05c,vc05f,vc05n
Demonstrates how to overlay a VectorPlot on a MapPlot.

vc06c,vc06f,vc06n
Demonstrates how to use a scalarfield to determine the color of the vector arrow fill.

vc07c,vc07f,vc07n
Emulates the LLU example "fcover"; shows how to overlay contours and vectors on a MapPlot.

vc08c,vc08f,vc08n
Plots wind vectors at a grid stride of 3. Vectors are colored by wind speed.

vc09c,vc09f,vc09n
Plots an animation of the January 1996 snow storm using wind vectors colored by temperature over a pressure field contour plot.

xy01c,xy01f,xy01n
Shows all the default resources for an xyPlot object.

xy02c,xy02f,xy02n
The same as example xy01c, only with some of the X and Y axis resources tweaked.

xy03c,xy03f,xy03n
The same as example xy01c, but with the data modified a little, and the line resources tweaked.

xy04c,xy04f,xy04n
Demonstrates how to create an xyPlot with multiple lines. Some of the line resources are tweaked.

xy05c,xy05f,xy05n
Demonstrates how to create an XyPlot with multiple lines, using multiple data items (except for xy05c, which uses the CoordArrTable object).

xy06c,xy06f,xy06n
Demonstrates how to read in netCDF files to generate various xy plots.

xy07c,xy07f,xy07n
Demonstrates how to draw a "scattergram".

xy08c,xy08f,xy08n

```
           Shows how to use irregular points  to  change  the
           transformation of your plot.

     xy09c,xy09f,xy09n
           Shows how to create different kinds of axes.

     xy10c,xy10f,xy10n
           Shows how to overlay an XyPlot on a MapPlot.

     xy11c,xy11f,xy11n
           Demonstrates how to display  an  xyPlot  to  an  X
           Window, and then copy it to a meta file.

     xy12c     Demonstrates how to incorporate  the  HLU  library
           with  a  GUI.   It  uses an X/Motif interface, and
           also allows the user to copy  frames  from  the  X
           Window to a meta file.

     xy13c     Shows how to interactively  retrieve  data  values
           from a plot by pointing and clicking on the plot.

     xy14c,xy14f,xy14n
           Demonstrates the data manipulation capabilities of
           NCL.

     xy15c,xy15f,xy15n
           Demonstrates the creation of a set of four stacked
           XyPlots.

     xy16c,xy16f,xy16n
           Demonstrates how to draw three different xy  lines
           on the same plot, but with different axis scales.

     xy17c,xy17f,xy17n
           Demonstrates how to stack three xy plots.

SEE ALSO
     ncargex(1NCARG)
```

Running LLU examples with the ncargex command

The ncargex command copies, compiles, and runs NCAR Graphics Low Level Utility (LLU) examples in your local directory. These examples demonstrate how to use the LLUs to create an NCAR Graphics application. A similar command called ng4ex is used to copy, compile, and run NCAR Graphics 4.x examples that demonstrate how to use the High Level utilities (HLUs) and NCL.

Depending on the example, the output graphic may be displayed in a workstation window or output to a file. More information on the ncargex command is provided in the ncargex man page; it is copied here:

```
NCARGEX(1NCARG)           NCAR GRAPHICS          NCARGEX(1NCARG)

NAME
     ncargex - NCAR Graphics  Fortran  and  C  Low-Level  Utility
     Examples

SYNOPSIS
     ncargex [-A] [-E] [-F] [-P] [-T] [-U] [-C] [ - Fortran]  [ -
     class]  [ -ps] [-x11] [-W workstation_type] [-areas] [-auto-
     graph] [-bivar] [-colconv] [-conpack] [- conran_family]  [ -
     conrec_family]  [ - dashline]  [ - dashpack]  [ - ezmap]  [-
     field_flow] [-gflash] [-gks] [-gridall] [-halftone] [-histo-
     gram]  [ - isosrfhr]  [-isosurface] [-labelbar] [-ngmath] [-
     ngmisc] [- plotchar]  [ - polypack]  [ - pwrite_family]  [ -
     scrolled_title]  [-seter] [-softfill] [-spps] [-streamlines]
     [-surface] [-threed] [-vectors] [-wmap] [-misc] [-clean] [ -
     n] [-noX11] [-onebyone] example_name ...

DESCRIPTION
     ncargex provides the user with access to over  300  complete
     example  NCAR Graphics Fortran and C source codes, including
     the examples in the NCAR Graphics Tutorial.  ncargex  copies
     the  source  code  for  the  specified  example(s)  into the
     current directory and then compiles, links, and executes the
     example.   Depending on the type of workstation specified on
     the command line, the output may either  be  an  NCGM  (NCAR
     Graphics  Metafile)  file,  one  of many types of PostScript
     files, or a text dump.  It is also possible for no output to
     be  produced  if  you select the "x11" workstation, in which
     case each frame is displayed directly to a separate X window
     after  it  is  generated.  If no workstation is specified on
     the command line, then it defaults to an "NCGM", unless  the
     example is a special one which is discussed below.

     If you select one of the workstation types that produces  an
     output  file,  then the file name will have the same name as
     the example and ending with an appropriate suffix:  ".ncgm",
     ".txt", ".ps", etc.

     Certain examples were created to  demonstrate  a  particular
     function,  like  how to rename your metafile from within the
     program, how to use the full page when going  to  PostScript
     output, how to use the X11 driver and produce a graphic file
     at the same time, etc.  If the example that you ask  for  is
     one     alert you.

     In order to run  ncargex,  you  must  have  your  NCARG_ROOT
     environment  variable  set to the parent directory where the
     NCAR Graphics libraries, binaries, and  include  files  were
     installed.  If this environment variable is not set, ncargex
     will attempt to set  it  for  you.   If  the  NCAR  Graphics
     libraries,  binaries,  and  include files were not installed
     under one parent directory, then you will need  to  set  the
     environment     variables     NCARG_LIB,     NCARG_BIN,     and
     NCARG_INCLUDE instead.  Please see "man ncargintro" for more
     information.

OPTIONS
     -W    Specify the workstation type.  This  argument  can  be
```

specified as a number or as a string, where the number has the same value you would use in a call to GOPWK. The following numbers are valid:

- 1 - NCGM
- 8 - X11 window.
- 10 - text dump of graphics output.
- 20 - color PostScript in portrait mode.
- 21 - color Encapsulated PostScript (EPS) in portrait mode.
- 22 - color Encapsulated PostScript Interchange format (EPSI) in portrait mode.
- 23 - monochrome PostScript in portrait mode.
- 24 - monochrome Encapsulated PostScript (EPS) in portrait mode.
- 25 - monochrome Encapsulated PostScript Interchange format (EPSI) in portrait mode.
- 26 - color PostScript in landscape mode.
- 27 - color Encapsulated PostScript (EPS) in landscape mode.
- 28 - color Encapsulated PostScript Interchange format (EPSI) in landscape mode.
- 29 - monochrome PostScript in landscape mode.
- 30 - monochrome Encapsulated PostScript (EPS) in landscape mode.
- 31 - monochrome Encapsulated PostScript Interchange format (EPSI) in landscape mode.

If you want to specify the workstation as a string, then there are a few ways this can be done. For example, the following workstation types are valid:

- ncgm - NCGM
- x11 - X11 window
- text - text dump of graphics output

For PostScript output, there are more attributes: the type of PostScript you file want ("ps", "eps", or "epsi"), whether you want color or monochrome ("color" or "mono"), and whether you want portrait or landscape mode ("port" or "land"). The orientation and the color preference can be omitted (as they will will default to "port" and "color" respectively), but you must specify the PostScript file type. Each attribute can be entered in any order, separated by periods. The following are examples of valid PostScript workstation types:

ps.color - color PostScript in portrait mode.

land.eps.mono - monochrome Encapsulated PostScript format (EPS) in landscape mode.

epsi - color Encapsulated PostScript Interchange format (EPSI) in portrait mode.

port.mono.ps - monochrome PostScript in portrait mode.

Any combination of these three types of attributes can be used, as long as one of them is the PostScript file type.

-A Generate all available C/Fortran examples, tests, programmer doc, fundamental and tutorial examples.

-E Generate all available C/Fortran examples.

- F Generate all available fundamental C/Fortran examples.

-P Generate all available programmer doc C/Fortran examples.

-T Generate all available tests.

-U Generate all available tutorial C/Fortran examples.

-C Generate all available C examples.

-Fortran Generate all available Fortran examples.

-class Generate all available class C/Fortran examples.

-ps Generate all C/Fortran examples that use the PostScript driver.

-x11 Generate all C/Fortran examples that use the X11 driver.

-areas Generate all areas C/Fortran examples.

-autograph Generate all autograph C/Fortran examples.

-bivar Generate all bivar C/Fortran examples.

-colconv Generate all colconv C/Fortran examples.

-conpack	Generate all conpack C/Fortran examples.
-conran_family	Generate all conran Fortran examples.
-conrec_family	Generate all conrec Fortran examples.
-dashline	Generate all dashline C/Fortran examples.
-dashpack	Generate all dashpack C/Fortran examples.
-ezmap	Generate all ezmap C/Fortran examples.
-field_flow	Generate all field_flow C/Fortran examples. Includes streamlines and vectors examples.
-gflash	Generate all gflash C/Fortran examples.
-gks	Generate all gks C/Fortran examples.
-gridall	Generate all gridall C/Fortran examples.
-halftone	Generate all halftone Fortran examples.
-histogram	Generate all histogram C/Fortran examples.
-isosrfhr	Generate all isosrfhr Fortran examples.
-isosurface	Generate all isosurface C/Fortran examples.
-labelbar	Generate all labelbar C/Fortran examples.
-ngmath	Generate all ngmath C/Fortran examples.
-ngmisc	Generate all ngmisc C/Fortran examples.
-plotchar	Generate all plotchar C/Fortran examples.
-polypack	Generate all polypack C/Fortran examples.
-pwritx_family	Generate all pwrite Fortran examples.
-scrolled_title	Generate all scrolled C/Fortran examples.
-seter	Generate all seter C/Fortran examples.
-softfill	Generate all softfill C/Fortran examples.
-spps	Generate all spps C/Fortran examples.
-streamlines	Generate all streamlines C/Fortran examples.
-surface	Generate all surface C/Fortran examples.
-threed	Generate all threed C/Fortran examples.
-vectors	Generate all vectors C/Fortran examples.

-wmap Generate all weather map C/Fortran examples.

- misc Generate all miscellaneous C/Fortran examples.

-clean Remove everything but the ".ncgm" file.

-n Specifies that the example should not be compiled, linked, or run, but just copied into your local directory.

-noX11 Do not link in the X library when linking the selected examples and/or tests. A stub will be linked instead.

- onebyone Specifies that the selected examples and/or tests should be generated one at a time and viewed as they are generated. This is intended for use during testing of new releases at NCAR.

Below is a list of all the available ncargex examples. They are listed according to which utility they belong with.

EXAMPLES AVAILABLE

AREAS Examples:

arex01 arex02 arex03 cardb1 cardb2 caredg carfill carline carmap tareas c_tareas

AUTOGRAPH Examples:

agex01 agex02 agex03 agex04 agex05 agex06 agex07 agex08 agex09 agex10 agex11 agex12 agex13 fagaxclr fagaxlbl fagaxmax fagcuclr fagcudsh fagezmxy fagezmy fagezxy fagezy fagilclr fagovrvw tagupw tautog c_agex07

BIVAR Examples:

cbex01 cidsfft c_cbex01

COLCONV Examples:

coex01 coex02 coex03 fcce01 fcce02 tcolcv c_coex02

CONPACK Examples:

cbex01 ccpback ccpcff ccpcfx ccpcica ccpcir ccpcis ccpcit ccpclc ccpcld ccpcldm ccpcldr ccpcll ccpclu ccpcnrc ccpdflt ccpezct ccpfil ccpga ccphand ccphcf ccphl ccphlt ccpila ccpils ccpilt ccpklb ccplbam ccplbdr ccpline ccpllb ccpllc ccplll ccpllo ccpllp ccpllt ccpllw ccpmap ccpmovi ccpmpxy ccpncls ccpnet ccpnof ccpnsd ccppc ccppc1 ccppc2 ccppc3 ccppc4 ccppkcl ccppole ccprc ccprect ccprwc ccprwu ccpscam ccpset ccpsps1 ccpsps2 ccpspv ccpt2d ccptitle ccpvp ccpvs cidsfft colcon cpex01 cpex02 cpex03 cpex04 cpex05 cpex06 cpex07 cpex08 cpex09 cpex10 cpex11 cpex12 cpex13 cpex14 cpex15 tconpa c_cbex01 c_colcon

CONRAN FAMILY Examples:

tconan tconaq tconas

CONREC FAMILY Examples:

tcnqck tcnsmt tcnsup tconre

DASHLINE Examples:

fdlcurvd fdldashc fdldashd fdlsmth tdashc tdashl tdashp tdashs c_fdldashc

DASHPACK Examples:

tdshpk c_tdshpk

EZMAP Examples:

cezmap1 cezmap2 cezmap3 cmpclr cmpdd cmpdrw cmpel cmpfil cmpgci cmpgrd cmpgrp cmpita cmpitm cmplab cmplbl cmplot cmpmsk cmpou cmppos cmpsat cmpsup cmptit cmptra cmpusr eezmpa mpex01 mpex02 mpex03 mpex04 mpex05 mpex06 mpex07 mpex08 mpex09 mpex10 mpex11 mpex12 mpexfi tezmap tezmpa tezmpb c_mpex05 c_eezmpa

FIELD FLOW Examples:

ffex00 ffex01 ffex02 ffex03 ffex04 ffex05 fcover fstream stex01 stex02 stex03 vvex01 vvex02 vvex03 c_ffex03 c_vvex03

GFLASH Example:

tgflas c_tgflas

GKS Examples:

fcell fcell0 fgke01 fgke02 fgke03 fgke04 fgkgpl fgkgpm fgkgtx fgklnclr fgklnwth fcirc fgpm01 pgkex01 pgkex02 pgkex03 pgkex04 pgkex05 pgkex06 pgkex07 pgkex08 pgkex09 pgkex10 pgkex11 pgkex12 pgkex13 pgkex14 pgkex15 pgkex16 pgkex17 pgkex18 pgkex19 pgkex20 pgkex21 pgkex22 pgkex23 pgkex24 pgkex25 pgkex26 pgkex27 c_gtxpac c_pgkex21

GRIDALL Example:

ccpga tgrida c_tgrida

HALFTONE Example:

thafto

HISTOGRAM Examples:

thstgr thstmv c_thstmv

ISOSRFHR Example:

tisohr

ISOSURFACE Examples:

fisissrf fispwrzi tisosr tpwrzi c_tisosr

LABELBAR Examples:

elblba tlblba clbfil clbbar clblbr c_elblba

NGMATH Examples:

csex01 csex02 csex03 csex04 csex05 csex06 csex07 dsex01 dsex02 dsex03
dsex04 dsex05 dsex06 dsex01d nnex01 nnex02 nnex03 nnex04 nnex05 nnex06
nnex07 nnex08 nnex09 nnex01d ftex01 ftex02 ftex03 ftex04 ftex05 ftex06
c_csex01 c_csex02 c_csex03 c_csex04 c_csex05 c_csex06 c_csex07 c_dsex01
c_dsex02 c_dsex03 c_dsex04 c_dsex05 c_dsex06 c_dsex01d c_nnex01 c_nnex02
c_nnex03 c_nnex06 c_nnex01d c_ftex01 c_ftex02 c_ftex03 c_ftex04 c_ftex05
c_ftex06

NGMISC Examples:

fngngdts fngwsym c_fngwsym

PLOTCHAR Examples:

epltch fpchiqu fpcloqu fpcfonts tpltch c_epltch

POLYPACK Examples:

ppex01 tppack c_ppex01

PWRITE FAMILY Examples:

tpwrtx tpwry

SCROLLED TITLE Examples:

fslfont slex01 slex02 tstitl c_slex01

SETER Examples:

tseter

SOFTFILL Examples:

fsfsgfa fsfwrld sfex01 sfex02 tsoftf c_sfex02

SPPS Examples:

fcoord fcoord1 fcoord2 fspcurve fspline fsppoint fspponts
splogy sprevx c_fcoord2

STREAMLINES Examples:

fstream ffex00 ffex01 ffex03 ffex04 stex01 stex02 stex03
tstrml c_ffex03

SURFACE Examples:

fsrezsrf fsrpwrzs fsrsrfac srex01 tsrfac tpwrzs c_srex01

THREED Examples:

```
fthex01  fthex02  fthex03  fthex04  fthex05  tthree   tpwrzt
 c_fthex01

VECTORS Examples:

ffex00 ffex01 ffex02  ffex05  fcover  tvelvc  vvex01  vvex02
vvex03 c_vvex03

WEATHER MAP Examples:

wmex01 wmex02 wmex03  wmex04  wmex05  wmex06  wmex07  wmex08
wmex09 wmex10 wmex11 wmex12 wmex13 wmex14 c_wmex09

Miscellaneous Examples:

bnchmk example ncargworld

X11 Examples:

fgke01 fgke04 c_xwndws
```

SEE ALSO

```
Online:          ncargf77(1NCARG),          ncargcc(1NCARG),
ncargfile(1NCARG),    ng4ex(1NCARG),    ncarg_cbind(5NCARG),
ncargintro(5NCARG)

Hardcopy:  NCAR Graphics Fundamentals, UNIX Version
```

COPYRIGHT

HLU terminology

Many object-oriented terms associated with HLU programming are defined in the glossary. Some of the more important terms include:

- class
- object
- resource
- resource file
- child
- parent
- inheritance
- application (App) class
- View class
- data classes
- workstation classes
- composite classes
- API

See also:

- HLU classes - alphabetically
- HLU API - alphabetically

Understanding classes and objects

In object-oriented terminology, a class is a template for defining objects. It specifies the names and types of variables that can exist in an object, as well as "methods"--procedures for operating on those variables. A class can be thought of as a "type", with the objects being a "variable" of that type. Multiple objects, or instances of a class can be created in a single HLU program, just as you declare multiple variables of the same type in any program.

For example, the TextItem class is a template for creating an object that contains a text string. This object will have a particular set of text attributes such as font, size, and color. If we set the values of the object variables--resources--in a certain way, we can create the TextItem object "Hello World". Resources that are available for objects of the TextItem class include the text string ("Hello World" in this case), the type of font, the color of the characters, the size of the characters, the line width of the characters, etc. A TextItem object is thus an instance of the TextItem class with a set of values assigned to the associated resources. We can create a second TextItem object if we want to with a new set of resource values such as: "THIS IS ALSO A TEXTITEM OBJECT."

Class versus object

Many people get confused by the difference between class and object. The difference is simple and conceptual. A class is a template for objects. A class defines object properties including a valid range of values, and a default value. A class also describes object behavior. An object is a member or an "instance" of a class. An object has a state in which all of its properties have values that you either explicitly define or that are defined by default settings.

This subtle conceptual difference between classes and objects shows why there is a tendency to want to use them interchangeably.

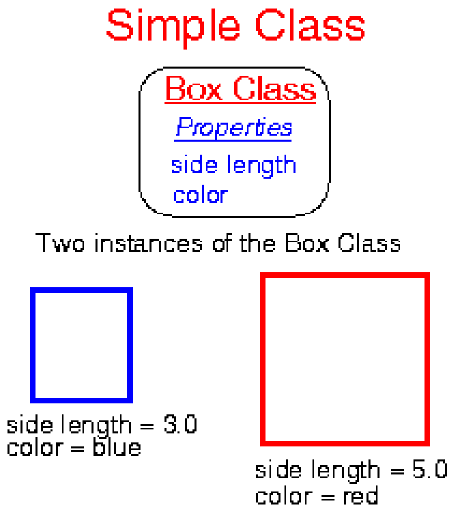
Derived classes and inheritance

Sometimes it is convenient to develop a class that shares properties with another class but yet is distinct from the original. The new class derives properties from an existing class but also extends or adds its own properties. This new class is called a "derived class" and is said to "inherit" its properties and functionality from the original class.

For example, lets say the original class is a class called Box that is defined to have two properties: side length, and color. The valid values for these properties in the Box class are:

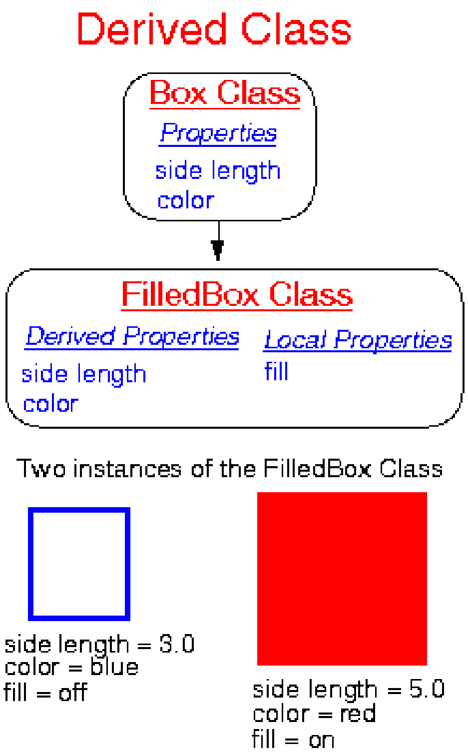
> 0. <= side length <= 10., default= 5.
> Color is red or blue, default = red.

If we create an instance of the Box class or, in other words, an object that is in the Box class, it will have the Box class properties (side length, and color). The values of these properties will be defined as the default unless they are explicitly set.



Now let’s assume we want to create a box object that is filled rather than the hollow boxes created using the Box class template. Instead of creating a completely new template that has many of the same properties of the Box class, we can derive a new class and extend the functionality of the derived class as necessary to create a class that defines filled box objects.

FilledBox will be the name of the derived class. Since it is derived from the Box class, it will automatically inherit all the properties that were in the Box class, namely side length, and color. However, in order to create boxes that are filled, we will add a new property called fill with possible values of "on" or "off" and a default value of "on." Any new objects that are created that are in the FilledBox class will be drawn as filled boxes unless the fill property is set to "off.".



Superclasses and subclasses

Another way to refer to a derived class is as a "subclass" of the original class. The class from which the subclass is derived is known as its "superclass." For example, in the HLU library the superclass of the TextItem class is the View class. The thing that all View subclasses have in common is that they can all be used to create graphical objects. Some other subclasses of the View class are XyPlot, VectorPlot, StreamlinePlot, ContourPlot, MapPlot, Title, TickMark, Legend, and LabelBar. The superclass of all subclasses is the Base class.

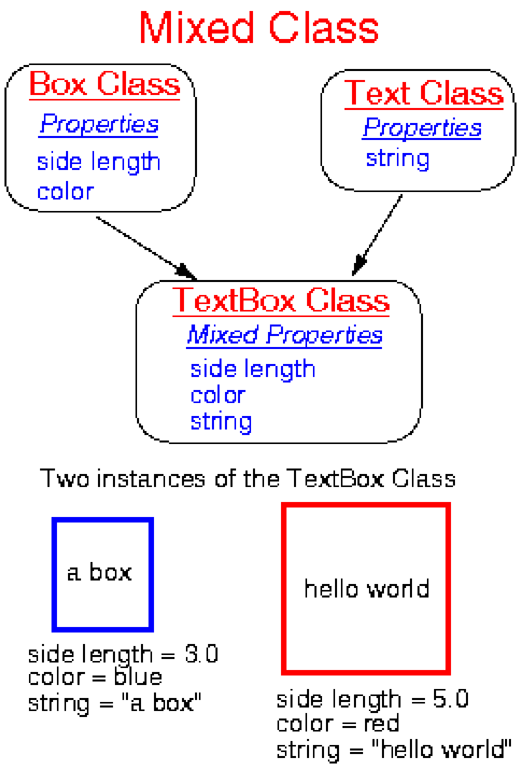
Often the superclasses cannot be instantiated directly. For instance, you cannot create a Base object or a View object. The role of these superclasses is to define common methods and resources that, through inheritance, are automatically available to their subclasses. For example, the non-instantiable Workstation superclass defines a resource, wkColorMap, that contains a list of currently allocated colors. Since the PSWorkstation class, used to output PostScript, is a subclass of Workstation, any PSWorkstation object can access the wkColorMap resource. In fact, all the resources defined by the Workstation class are available to a PSWorkstation object. Indeed, they are available to objects belonging to any subclass of Workstation, such as the NcgmWorkstation class or the XWorkstation class.

Mixed classes

A mixed class is another way to combine the functionality from other classes into a new class. A mixed class manages the properties of other classes and may only use a subset of the functionality of a class, whereas a derived class uses the complete set of functionality of its superclasses and usually extends this functionality.

For example, suppose that in addition to the Box class we have another class called Text. The Text class has one property, called string, with a default value of "hello world."

We could create a mixed class that would manage these two simple classes in order to produce a box object with text. We will call the new class TextBox, and, in this example, it will use all of the properties from both of the classes from which it is composed. In many cases, the mixed class may use only a subset of the properties from these classes. An object that is in the TextBox class would have the following properties: side length, color, and string.



The XyPlot class is an example of an HLU mixed class. It combines the functionality of the TickMark class and the Title class, among others. Since an XyPlot object contains both tick marks and titles (which are both available as HLU classes), it is more efficient to have the XyPlot class manage these properties rather than use its own code and properties to provide redundant functionality.

Composite classes

In the HLU library, mixed classes are known as "composite classes." In the class hierarchy chart, the boxes with a heavy outline are composite classes. For example, if you click on the box PlotManager class, you will see that it has as composite class members the Title, Legend, LabelBar, and TickMark classes. Composite classes allow access to at least a subset of the resources of their composite class members.

Composite class members can be nested. For example, since the ContourPlot class includes the PlotManager as a composite class member, all the composite class members of the PlotManager are available to the ContourPlot class. If you create a ContourPlot object, you will have access not only to the resources of the ContourPlot and its superclasses, but also to resources of the PlotManager and the PlotManager's composite class members, Title, TickMark, Legend, and LabelBar. In other words, you only have to create one object in your HLU program to have the effective functionality of a whole tree of composite class members.

User creatable objects

In a user program, only objects belonging to certain classes can be created directly. In the class hierarchy chart, the yellow boxes denote these user-instantiable classes. The other classes fall into one of four categories:

- Non-instantiable superclasses such as Base and View
- Classes designed to function only as composite class members, such as PlotManager and subclasses of the Transformation class
- The classes that can have only one instance, such as Error and Workspace; they are automatically instantiated when the HLU library is initialized
- Classes that are instantiated by certain objects for a specialized purpose on behalf of the user; these currently include the XyDataSpec and AnnoManager classes

Dynamically associated objects

In addition to the class hierarchy and composite class relationships, the HLU library has a mechanism that allows you to associate independently-created View objects dynamically. You can "overlay" Transform class objects onto a plot object's data space. You can also make any View object into an "annotation" of a plot object. The combination of the base plot object, its overlays, and its annotations acts in many ways like a single object. Plot objects, overlays, and annotations are discussed in the PlotManager class module, and also in the AnnoManager class module.

Class hierarchy versus instance hierarchy

Besides the class hierarchy of subclasses derived from the Base superclass, you should be aware that the HLU library defines an "instance hierarchy" of the objects that are created in the course of executing an HLU program. These two hierarchies are completely distinct, and you should be careful not confuse them.

Whenever you create an object, you must specify the object's "parent" as one of the parameters to the create call. Each object you create is therefore the "child" of some parent object. The initial parent, the "ancestor" of all the objects created, must be an "application" (App) object. Depending on the call used to initialize the HLU library, you may need to create this object yourself, or the library may automatically create it for you.

The instance hierarchy is significant in the following ways:

- When you destroy a parent object all its children are destroyed along with it.

- A View object must have a Workstation class ancestor that supplies the viewspace on which it is drawn.
- The resource database uses the instance hierarchy to determine how resource specifications in resource files apply to particular objects in an HLU program.

See also:

- HLU terminology
- Class hierarchy
- Composite classes
- HLU classes - alphabetically
- HLU API - alphabetically

Class hierarchy (subclasses and superclasses)

The HLU library has a hierarchical class structure. It starts with the Base class as the highest node. All classes are subclassed from the Base class.

Some of the more important nodes on this hierarchical tree are the View class and the Workstation class, plus the App class, DataItem class, Error class, PlotManager class, and AnnoManager class. Later, we will learn how to set object variables (resources) and create specific objects. First, let's examine a bit more about how classes can interact in the HLU library.

Subclasses are classes that are derived from another class. This means that they "inherit" the functionality of the class they are derived from, then they add something to this functionality. Superclasses are classes from which other classes (subclasses) have been derived. For example, TextItem is an immediate subclass of the View class. This means that TextItem is directly below View in the class hierarchy. Therefore, View is also an immediate superclass of TextItem.

Objects of a subclass are also valid objects of classes higher in the hierarchy. Therefore, in the hierarchy of:

```
    Base class  (The highest level)
         |
         |
    View class
         |
         |
  Transform class
         |
         |
  DataComm class
         |
         |
    XyPlot class
```

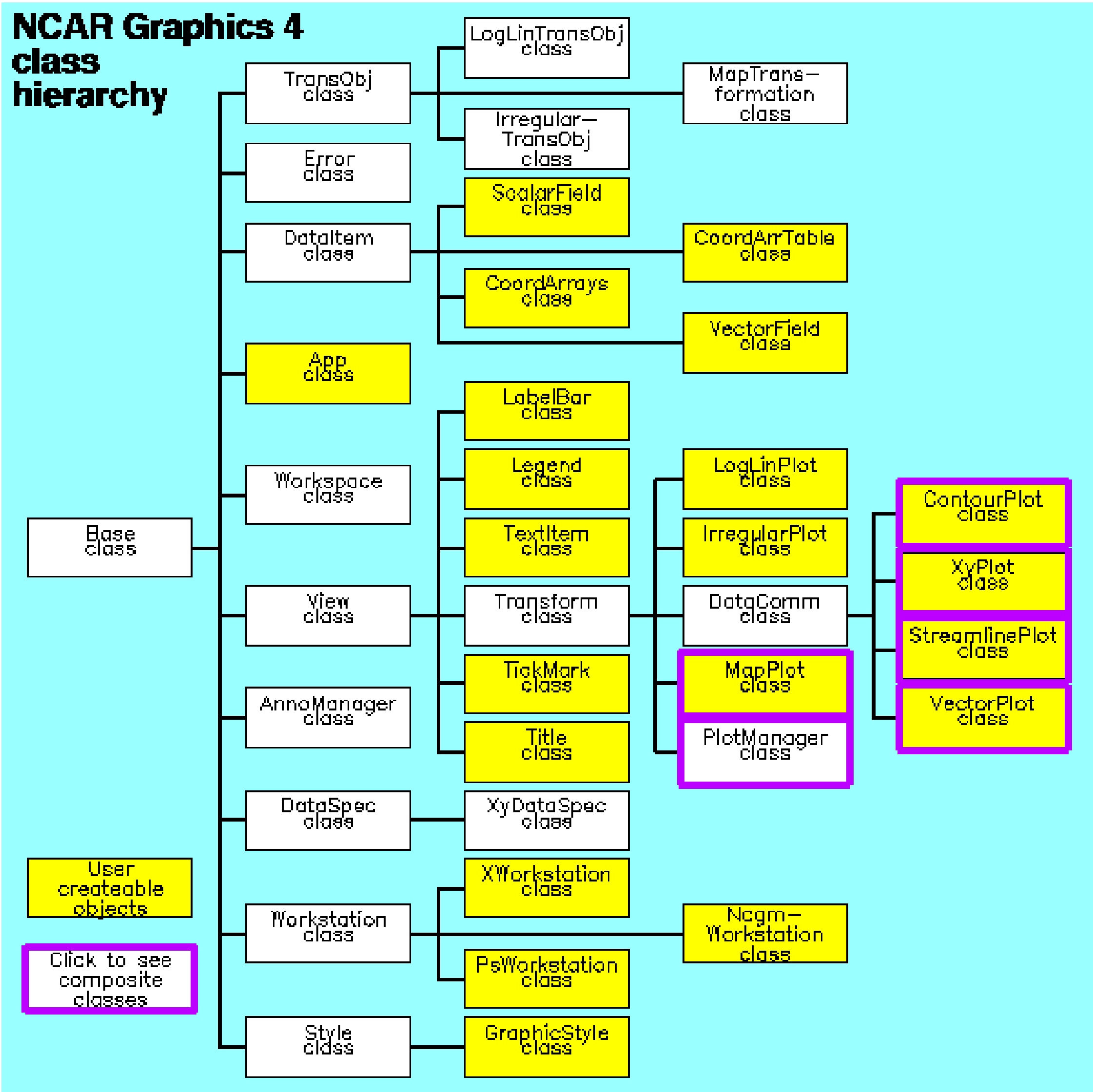
an XyPlot object is considered to have the characteristics of a DataComm object, a Transform object, a View object, and a Base object. Any XyPlot object inherits the resources of all of these superclasses.

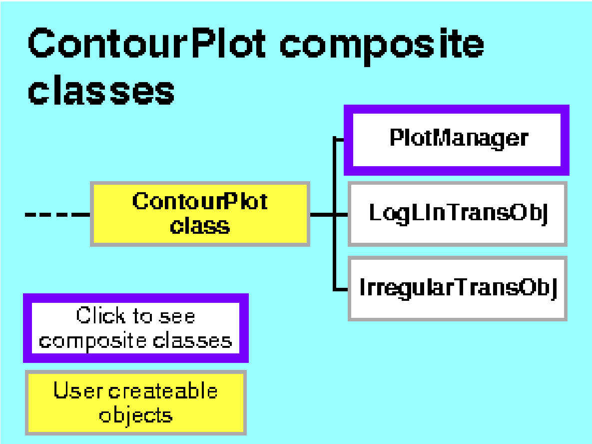
The hierarchical structure and the composite class members are shown in the Class hierarchy diagram. Please study this important diagram if you have not already done so.

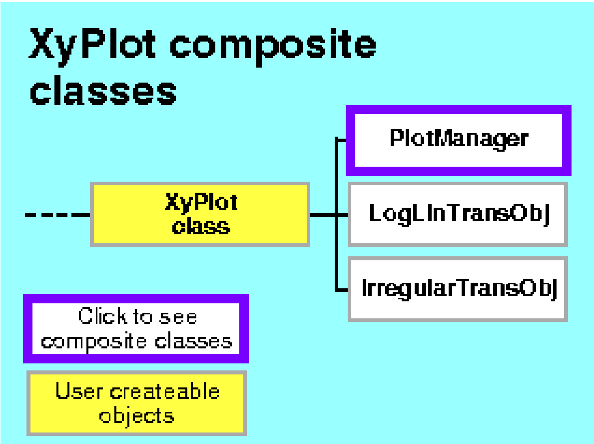
See also:

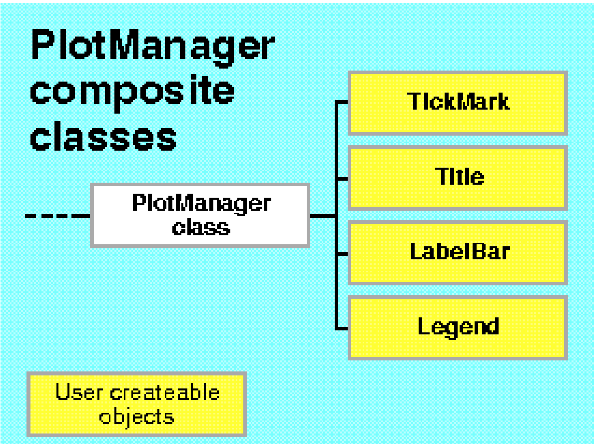
- HLU terminology
- Class overview
- Class composite
- HLU classes - alphabetically
- HLU API - alphabetically

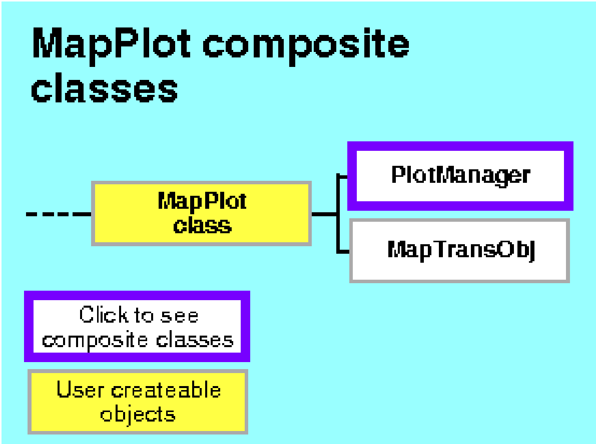
Class hierarchy diagrams

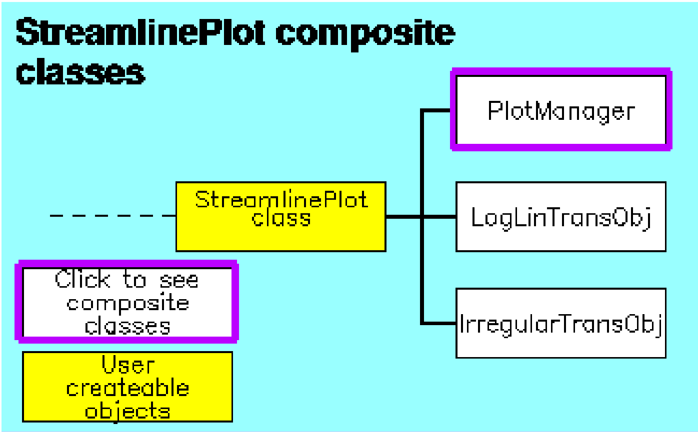



ContourPlot composite classes
PlotManager
ContourPlot class
LogLinTransObj
IrregularTransObj
Click to see composite classes
User createable objects


XyPlot composite classes
PlotManager
XyPlot class
LogLinTransObj
IrregularTransObj
Click to see composite classes
User createable objects


PlotManager composite classes
TickMark
Title
PlotManager class
LabelBar
Legend
User createable objects





Composite classes

Composite classes are classes that incorporate the functionality of other classes that are not directly in their class hierarchy. One of the primary considerations in favor of using the object-oriented paradigm in software development is that once a class has been created, all of the functionality of that class can be used in another class. This results in the elimination of much repeated code.

For example, to create a complete plot of variable X versus variable Y, one may need a set of X axis, Y axis, and main titles, a tick-marked set of axes with scale labels, and the curve of X versus Y. To create a contour plot of a 2-D temperature array, you still need the set of titles and the tick-marked axes with scale labels. So we created a Title class and a TickMark class which can then be incorporated into the higher level View classes such as XyPlot, ContourPlot, VectorPlot, and StreamlinePlot. Not only does this reduce the amount of code, it also ensures consistency in how things are done and how things work. Such consistency is extremely valuable for the implementation of Graphical User Interfaces.

Objects created from composite classes have all the resources of their class, their superclasses, and their composite member classes.

The composite class members of a class are shown on the diagram of class hierarchies; click on any class in a colored box.

See also:

- HLU terminology
- Class overview
- Class hierarchy
- HLU classes - alphabetically
- HLU API - alphabetically

View class

All graphical objects are created from subclasses of the View class. They include TextItem, Title, TickMark, Legend, LabelBar, XyPlot, ContourPlot, MapPlot, StreamlinePlot, and VectorPlot.

View class objects inherit the viewport resources of the View class. The viewport is a rectangular subspace of NDC space (0. to 1. in both dimensions) that serves as the principal boundary of all View objects. The default viewport is (.2,.8) in X and (.2,.8) in Y. The viewport can be set through resource values in a resource file.

All View objects must have an ancestor in the instance hierarchy that is a Workstation object. A view can be only be drawn on the workstation that is its ancestor.

Some view objects such as XyPlot, ContourPlot, StreamlinePlot, and VectorPlot acquire their data from a DataItem class.

Composite graphical objects are created using the PlotManager class. Any View class name that ends in "Plot" (ContourPlot, XyPlot, StreamlinePlot, VectorPlot, and MapPlot) uses the PlotManager class to manage how multiple graphical objects are overlayed on the plot.

Annotations are placed on a plot under the control of the AnnoManager class. Annotations can be added to any plot that uses a View class object ending in the word "Plot," such as XyPlot, ContourPlot, StreamlinePlot, VectorPlot, or MapPlot.

See also:

- Workstation class
- Data class
- PlotManager class
- AnnoManager class
- HLU classes - alphabetically
- HLU API - alphabetically

Workstation class

Workstation objects specify output devices to which the graphics are sent. The supported workstations include a direct X Window, an NCAR-encoded Computer Graphics Metafile (NCGM), and PostScript in its many forms.

The Workstation class contains three subclasses: XWorkstation, NcgmWorkstation, and PSWorkstation. Only one NcgmWorkstation can be open at one time in an application, but up to 15 workstations can be open simultaneously with the remaining workstations of either type PSWorkstation or XWorkstation.

The HLU function NhlChangeWorkstation can be used to switch between workstation types in an application. NhlClearWorkstation can be used to clear the workstation device.

See also:

- View class
- HLU classes - alphabetically
- HLU API - alphabetically

Data classes

Data classes are any of the classes that are used to provide user input data to any of the objects that utilize such data. Included are the CoordArrays class, the CoordArrTable class, the ScalarField class, and the VectorField class.

StreamlinePlot, VectorPlot, XyPlot, and ContourPlot are View classes that require an input data class.

Creating a data object and linking it to a view object is a two-step process. Examples for one-dimensional data to be used in an XyPlot and a two-dimensional field to be used in a ContourPlot are shown below in both C and Fortran.

Data input for an XyPlot

Step 1

Create a data object by assigning data values using the caXArray and caYArray resources of the CoordArrays class.

Fortran:

The CoordArrays data class is used for bringing one-dimensional data curves into an XyPlot object in a Fortran HLU program. The set of Y values for the curve is defined by the caYArray resource. The set of

X values of the curve can be explicitly set in the caXArray resource, or can implicitly default to the point number if not set as a caXArray resource.

```
C
C Create the dataid object after assigning n values of the temp array
C to the caYArray resource, and n values of the time array to caXArray.
C
      call NhlFRLClear(rlist)
      call NhlFRLSetFloatArray(rlist,'caXArray',time,n,ierr)
      call NhlFRLSetFloatArray(rlist,'caYArray',temp,n,ierr)
      call NhlFCreate(dataid,'xyData',NhlFCoordArraysClass,0,
     +     rlist,ierr)
```

If the resource caXArray is not explicitly set, as done above, it will default to a set of integers from 1 to n.

C code equivalent:

```
    NhlRLClear(rlist);
    NhlRLSetFloatArray(rlist,NhlNcaYArray,(float *)temp,n);
    NhlRLSetFloatArray(rlist,NhlNcaYArray,(float *)time,n);
    NhlCreate(&dataid,"xyData",NhlcoordArraysClass,
              NhlDEFAULT_APP,rlist);
```

Step 2

Create an XyPlot object using the xyCoordData resource to connect the data object, dataid, to the XyPlot object, plotid.

Fortran:

```
C
C The id for this dummy data object will now become the resource
C value for "xyCoordData".
C
      call NhlFRLClear(rlist)
      call NhlFRLSetInteger(rlist,'xyCoordData',dataid,ierr)
      call NhlFCreate(plotid,'xyPlot1',NhlFXyPlotClass,xworkid,
     +     rlist,ierr)
```

C code equivalent:

```
/*
 * The id from this dummy data object will now become the resource
 * value for "xyCoordData".
 */
    NhlRLClear(rlist);
    NhlRLSetInteger(rlist,NhlNxyCoordData,dataid);
    NhlCreate(&plotid,"xyPlot1",NhlxyPlotClass,xworkid,rlist);
```

xy06f.f and xy06c.c are examples that show the input and display of several datasets using CoordArrays and XyPlot objects.

A C language alternative for XyPlots

The data class CoordArrTable can also be used for creating one-dimensional data objects. It is especially useful for creating plots with multiple curves, where each curve can have a different number of points. Consider the following C code segment from the example program, xy05c.c. It shows setting the CoordArrTable resources ctYTable and ctYTableLengths. There are NCURVE curves of possibly varying lengths equal to length(1), length(2), ..., length(NCURVE). The data values all reside in array y, with pointers defined for each of the NCURVE curves.

```
/*
 * Create the CoordArrTable object which defines the data for the
 * XyPlot object. The id from this object will become the value for
 * the XyPlot resource, "xyCoordData".
 */
    NhlRLClear(rlist);
    NhlRLSetIntegerArray(rlist,NhlNctYTableLengths,length,NCURVE);
    NhlRLSetArray(rlist,NhlNctYTable,y,NhlTPointer,sizeof(NhlPointer),
                  NCURVE);
    NhlCreate(&dataid,"xyData",NhlcoordArrTableClass,
              NhlDEFAULT_APP,rlist);
/*
 * Create the XyPlot object and identify the xyCoordData resource as
 * the data object, dataid.
 */
    NhlRLClear(rlist);
    NhlRLSetInteger(rlist,NhlNxyCoordData,dataid);
    NhlCreate(&plotid,"xyPlot",NhlxyPlotClass,xworkid,rlist);
```

Data input for a ContourPlot

The ScalarField data object is used to pass two-dimensional datasets to a ContourPlot object. Sample Fortran and C codes are given below.

Fortran:

```
C
C Assign the two-dimensional data array z(N,M) to the sfDataArray resource,
C and create a ScalarField data object.
C
      call NhlFRLClear(rlist)
      len_dims(1) = N
      len_dims(2) = M
      call NhlFRLSetmdfloatarray(rlist,'sfDataArray',z,2,len_dims,ierr)
      call NhlFCreate(dataid,'Gendat',nhlfscalarfieldclass,appid,
     1                 rlist,ierr)
C
C Create a ContourPlot object, supplying the ScalarField object as data.
C
      call NhlFRLClear(rlist)
      call NhlFRLSetinteger(rlist,'cnScalarFieldData',dataid,ierr)
      call NhlFCreate(cnid,'ContourPlot1',nhlfcontourplotclass,
     1   wid,rlist,ierr)
```

C code equivalent:

```
/*
 * Assign the two-dimensional data array z(N,M) to the sfDataArray resource,
 * and create a ScalarField data object.
 */
    NhlRLClear(rlist);
    len_dims[0] = N;
    len_dims[1] = M;
    NhlRLSetMDFloatArray(rlist,NhlNsfDataArray,z,2,len_dims);
    NhlCreate(&dataid,"Gendat",NhlscalarFieldClass,appid,rlist);
/*
 * Create a ContourPlot object, supplying the ScalarField object as data.
 */
    NhlRLClear(rlist);
    NhlRLSetInteger(rlist,NhlNcnScalarFieldData,dataid);
    NhlCreate(&cnid,"ContourPlot1",NhlcontourPlotClass,wid,rlist);
```

cn04f.f and cn04c.c are examples that show the use of ScalarField and ContourPlot objects.

See also:

- HLU classes - alphabetically
- HLU API - alphabetically

Application class

Objects that are instances of the App class are used to keep track of resource databases. Every application must create at least one App object. One is created automatically if you begin your HLU program with an NhlOpen (C), or NhlfOpen (Fortran). Alternately, if you begin the program with an NhlInitialize (C) or NhlfInitialize (Fortran), you must create an App object yourself using NhlCreate (C), or NhlfCreate (Fortran).

See Steps 3 and 4 of HLU plot creation sequence.

See also:

- HLU classes - alphabetically
- HLU API - alphabetically

Error class

The Error class is used to configure the error-reporting module of the HLU library. It works in conjunction with the Error functions.

The Error class basically works in two modes: C and Fortran. The errBuffer, errLevel, and errPrint resources are not affected by the mode. The errFilePtr, errFileName, and errUnitNumber resources do depend on the mode.

The language mode that is used is determined by how the HLU library is initialized. If the library is initialized using a Fortran initialization function such as NhlfInitialize or NhlfOpen, then the error object uses Fortran mode, otherwise it uses C mode.

The resources are used to determine if errors should be buffered, what level of severity errors should be included, and where and when the messages should be printed.

See the Error class module for an explanation of Error resources.

See the Error API module for an explanation of Error API functions.

See also:

- HLU classes - alphabetically
- HLU API - alphabetically

Transform class

Subclasses of the Transform class support a data coordinate space along with forward and reverse transformations between data coordinates and normalized device coordinates.

All HLU "plot objects" are instances of Transform subclasses. The existing Transform subclasses include XyPlot, ContourPlot, and MapPlot, as well as LogLinPlot and IrregularPlot. Each of these classes includes as composite class members one or more subclasses of the Transformation class. The Transform class itself provides the data coordinate space while the Transformation subclass performs the actual transformations of the data. Note that although the TickMark object understands and uses transformations, it does not belong to the Transform class, since it does not support its own data space. The following sections discuss the transformational capabilities of each of the Transform subclasses.

XyPlot

The XyPlot class includes both the LogLinTransformation and the IrregularTransformation as composite class members. It allows access only to the Transformation resources used to set the boundaries of the data coordinate space. Each XyPlot axis can be set individually to use a linear, log, or irregular transformation.

The transformation type of an axis for an XyPlot object is set using the resources xyXStyle and xyYStyle. The options are:

- Irregular
- Linear
- Log

The default transformation type is Linear.

If you set either axis to "Irregular", you must supply an array of points as a representation of the irregular coordinate system. For the X Axis, you would set the array resource xyXIrregularPoints, and for the Y Axis, you would set xyYIrregularPoints.

ContourPlot

The ContourPlot class includes both the LogLinTransformation and the IrregularTransformation as composite class members. It allows limited access to a number of Transformation resources.

Selecting the transformation type for the ContourPlot object is a little more subtle than for the XyPlot. If either of the ScalarField object resources sfXArray or sfYArray is non null, then the axis corresponding to that array is assumed to be irregular. It does not need to be explicitly set by another resource.

To set a ContourPlot axis to logarithmic scaling, you set the LogLinTransformation composite resources, trXLog or trYLog, to a value of True. A value of False, the default, will result in that axis having linear scaling, unless it is assumed to be irregular.

MapPlot

The MapPlot class includes the MapTransformation as a composite class member. It allows access to all but one of the MapTransformation resources. Among the MapPlot transformation options are ten different geographic mapping projections, selected by setting the resource mpProjection:

- Orthographic
- Stereographic
- LambertEqualArea
- Gnomonic
- AzimuthalEquidistant
- Satellite
- Mollweide
- Mercator
- CylindricalEquidistant
- LambertConformal

CylindricalEquidistant is the default MapPlot transformation.

The MapTransformation class also includes a number of resources for limiting the boundaries of the projected area.

LogLinPlot

The LogLinPlot class includes the LogLinTransformation as a composite class member. It allows unrestricted access to all LogLinTransformation resources. Each axis of a LogLinPlot may be linear or logarithmic, depending on the setting of the boolean LogLinTransformation resources trXLog and trYLog.

IrregularPlot

The IrregularPlot class includes the IrregularTransformation as a composite class member. It allows unrestricted access to all IrregularTransformation resources. Each axis of an IrregularPlot may be linear, logarithim, or irregular, depending on the setting of the resources trXAxisType and trYAxisType. The options are:

- IrregularAxis
- LinearAxis
- LogAxis

The default is LinearAxis.

If you set either axis to "IrregularAxis", you must supply an array of points as a representation of the irregular coordinate system. For the X Axis, you would set the array resource trXCoordPoints, and for the Y Axis, you would set trYCoordPoints.

Transformation classes

Keep in mind that you never need (or are allowed) to instantiate any of the Transformation subclasses since they are built in as composite class members of the Transform subclasses. The resources belonging to these transformation classes are available to instantiated Transform class objects. Of course, as with any composite class member, the Transform subclasses may intercept or disable some or most of the Transformation class resources. For examples of how to set transformation options in your code, or in your resource files refer to the examples listed in the following section.

Examples

- xy08f.f and xy08c.c select irregular X and Y axes in the code.
- xy09f.f and xy09c.c select logarithmic axes in the xy09.res resource file.
- cn03f.f and cn03c.c select an irregular Y axis by setting sfYArray in the code.
- mp01f.f and mp01c.c select map projections other than the default in the code.

To copy any of these examples into your current working directory, you can type:

```
ng4ex -n example_name
```

(exclude the .c or .f, for example: `ng4ex -n tm03f`). To run the example, omit the -n.

See also:

- LogLinTransformation object
- LogLinPlot object

- IrregularTransformation object
- IrregularPlot object
- MapTransformation object
- Transform object
- HLU classes - alphabetically
- HLU API - alphabetically

PlotManager class

In order to use the PlotManager effectively, it is helpful to understand what is being managed. The PlotManager manages *plot objects*. What, then, are plot objects?

Plot objects

Plot objects are used to create plots, that is, graphical representations of data accompanied by annotations like labelled tickmarks that give a context to the data and help reveal its meaning. Think of a plot object as an organized collection of View objects that you can draw with a single call. Its focus is a rectangular space within which the data is visualized. This space is called the data coordinate space and is the viewport of a Transform object. By itself, a Transform object is not a plot object, because although it contains a representation of the data, it has no means of placing the data in context. This is where the PlotManager comes in.

You create a basic plot object whenever you instantiate a Transform object with an active PlotManager. Normally, you do not even need to be aware that you are doing this, because, by default, the PlotManager is activated whenever you instantiate one of the composite Transform classes that can be used as a plot object. These include XyPlot, ContourPlot, MapPlot, LogLinPlot, and IrregularPlot.

Role of the PlotManager

The PlotManager operates for the most part behind the scenes. One of its basic roles is to manage the built-in annotations that provide a context for the data. These built-in annotations (known as intrinsic annotations) include titles, tickmarks, a labelbar, and a legend; they are available to any of the plot object classes that choose to use them. The PlotManager does not concern itself with the content of these annotations; its role is limited to turning them on and off, positioning and sizing them relative to the plot object's current viewport, and ensuring that they do not overlap.

The PlotManager provides a number of resources for controlling these attributes. For instance, the resource pmLabelBarDisplayMode determines when the LabelBar should be displayed; and the resource pmLabelBarSide determines the side of the coordinate space viewport on which the LabelBar is to appear. Since the PlotManager is a composite member of the plot object, its resources are set as if they belonged directly to the plot object itself. As an example, to display the LabelBar on the left side of the viewport of a ContourPlot you are creating, you could execute:

```
C:
```

```
    NhlRLClear(rlist);
    NhlRLSetString(rlist,"pmLabelBarDisplayMode","always");
    NhlRLSetString(rlist,"pmLabelBarSide","left");
    NhlCreate(&cnid,"Contour1",NhlcontourPlotClass,wid,rlist);

Fortran:
    call NhlFRLClear(srlist)
    call NhlFRLSetString(rlist,'pmLabelBarDisplayMode','always',ierr);
    call NhlFRLSetString(rlist,'pmLabelBarSide','left',ierr);
    call NhlFCreate(cnid,'Contour1',NhlfcontourPlotClass,wid,rlist,ierr)
```

Besides managing the built-in annotations, the PlotManager has a number of other capabilities. It allows you to overlay other Transform objects into the data coordinate space of the plot object. It also allows you to add any number of independently-created View objects as external annotations. These overlays and annotations are called "plot members", and they function very much as if they are an integral part of the plot object. Drawing the plot object causes the plot members to be drawn as well; changing the plot object's size or position causes proportional changes to the plot members.

Overlays

Since an overlay must be a Transform object, it is interesting to note that the overlay might itself be a plot object. Most classes of plot object can perform either role, that of providing the data coordinate space on which Transform objects are overlaid, or that of becoming overlays themselves. A plot object that provides the data coordinate space is called a "base plot object". All plot objects are base plot objects when they are first created. When a plot object becomes an overlay, its PlotManager cedes its management responsibilities to the base plot object's PlotManager. Also note that the base plot determines the boundaries of the coordinate space. If the overlay's data coordinate space extends outside these boundaries, it is clipped.

A common use of overlays would be to render a contour plot of scalar data, such as temperature, projected onto a plot of the region of the earth to which the data values apply. You accomplish this by overlaying a ContourPlot object on a MapPlot object. First you independently create both the ContourPlot and the MapPlot, supplying the appropriate data to the ContourPlot and the appropriate limits and the desired projection to the MapPlot. Then when you execute the following call:

```
C:
    NhlAddOverlay(mapplotid,contourplotid,-1);

Fortran:
    call NhlfAddOverlay(mapplotid,contourplotid,-1,ierr)
```

the ContourPlot becomes an overlay of the MapPlot. Essentially, the ContourPlot and the MapPlot are merged into a single plot under control of the MapPlot's PlotManager. The ContourPlot's coordinate space is transformed into the data space of the MapPlot, as defined by one of the ten MapPlot projections and the current projection limits. You can no longer draw the ContourPlot as an independent object, but when you draw the MapPlot the contour data appears properly located within the MapPlot viewport.

If you want to draw the ContourPlot independently again, you must first remove it from the plot on which it is overlaid. This you would do using the NhlRemoveOverlay function.

You can look at the C example mp03c.c or the Fortran example mp03f.f to see a working example of a ContourPlot overlaid on a MapPlot. The command "ng4ex -n mp03c" will deposit the C code in your working directory. Substitute mp03f to get the Fortran code.

Multiple overlays

You can overlay more than one Transform object on the data coordinate space of a base plot. You could, for instance, display one dataset using filled contours, some related datasets using line contours with distinctive line styles, and, in addition, perhaps locate some important features using markers drawn by XyPlot. All of these components of the final graphic can be drawn on top of each other in a prescribed order controlled by the PlotManager.

If all of the graphical components are line drawings the order in which they are overlaid is not as important as it is when some of the objects are solid filled. For example, contour lines would have to be placed on top of a filled geographic map or they would disappear. Likewise a geographic continental outline would have to be placed on top of a set of filled contours or it would disappear.

The order in which the overlays are drawn is based on the overlay sequence, which you can determine by getting the value of the array resource, pmOverlaySequenceIds. By default, the contents of the base plot are drawn first, followed by each overlay in the order it was added, but you can control the sequence of overlays by manipulating the third parameter of the NhlAddOverlay function. However, the PlotManager also uses a three-phase draw sequence. If the plot object class also supports draw phases, you can gain a greater control over when particular elements of a plot are drawn by manipulating their draw phase. Currently only the ContourPlot and MapPlot have draw phase capabilities.

Managing the location of annotations

The PlotManager uses a scheme based on a series of concentric areas surrounding the plot object's viewport for positioning annotations. Built-in annotations like titles and labelbars, annotations created by plot objects, and external annotations created by the user are all treated alike. Each concentric area is an "annotation zone". Zones are numbered consecutively and except for zones 0 and 1, each zone is shaped like a rectilinear doughnut surrounding the zone preceding it. If no annotation objects are placed in a zone, it will have no area, meaning that its exterior boundary will be coincident with its interior boundary. There is no upper limit to the number of zones used.

The viewport of the plot object, for example, an XyPlot, defines the outer boundary of zone 1. Unlike the outer zones, zone 1 has no "hole" in the middle, because zone 0 is simply a point at the center of the viewport. The boundaries of zone 1 are defined by the viewport resources, vpXF, vpYF, vpWidthF, and vpHeightF, according to:

- vpXF is the left edge of the viewport in NDC space (0.2 default).
- vpYF is the top edge of the viewport in NDC space (0.8 default).
- vpHeightF is the height of the viewport in NDC space (0.6 default).
- vpWidthF is the width of the viewport in NDC space (0.6 default).

In the default case, 40% of the drawable viewspace is available for the outer concentric zones in which titles, legends, labelbars, and other annotations can be placed. In the example below, we put titles for the

X and Y Axes into zone 2. Then a main title and a legend are put in zone 3.

```
 -------------------------------
| [Zone 3]                      |
|MAIN TITLE EXTENDS TO HERE     |
|    -------------------------  |
|    |  Y   [Zone 2]            |
|    |                          |
|    |  A ---------------       |
|    |  X | [Zone 1]     |      |
|    |  I |              |      |
|    |  S |  (X vs Y     |      |
|    |    |  plot here)  |      |
|    |  T |              |      |
|    |  I ---------------       |
|    |  T  X AXIS TITLE EXTENT  |
|    |  L                       |
|    |  E                       |
|    -------------------------  |
|       *  #  @  +              |
|   (Curve legend here)         |
 -------------------------------
```

The important thing to note about this diagram is that the extent of the annotations occupying a zone determines the area of the zone. In other words the bounding box (the smallest containing rectangle) of the annotations in a zone defines its exterior boundary, as well as the interior boundary of the next higher numbered zone. So, for example, since the X Axis title in zone 2 extends further to the right than the Main Axis title in zone 3, the right boundaries of zones 2 and 3 are colinear, and equal to the right boundary of the bounding box of the X Axis title.

The advantage of this scheme is that you can position annotations without specifying the location in NDC space. Simply by placing annotations in different zones you can ensure that they do not overlap each other. For exact placement relative to the previous zone's boundary, numerical offset resources are available.

Annotation zones are discussed further in the description of the PlotManager Location Control Model.

External annotations

When you add an independently-created View object as an annotation to a plot object, the PlotManager creates an AnnoManager object that you use to specify locational and size attributes. The AnnoManager class module discusses how to create and manage external annotations and under what circumstances you might want to make annotations out of independent View objects.

Terminology

The glossary contains definitions of the terms used to describe the PlotManager's capabilities. The most significant are:

- plot object
- base plot
- plot member

- overlay
- annotation
- overlay sequence
- overlay functions
- annotation functions

See also:

- AnnoManager class module
- PlotManager Location Control Model
- HLU classes - alphabetically
- HLU API - alphabetically

AnnoManager class

You can add annotations to any of the plot object classes. These are classes whose name ends with "Plot", as in XyPlot, ContourPlot, or MapPlot. Annotations that you add are called "external annotations", as opposed to the built-in, or "intrinsic", annotations that are automatically available to all plot objects. The remainder of this module refers exclusively to external annotations.

Annotations can be created from any View class object. Thus, an annotation can be a simple TextItem, a LabelBar, a Legend, or even a complex ContourPlot.

The PlotManager is actually responsible for positioning and sizing annotations. However, the AnnoManager object associated with each annotation contains resources that specify how to place and size the annotation.

Why use annotations?

Suppose that, besides the built-in title annotations, you want to associate some additional explanatory text with a plot. You may want to create a TextItem and make it into an annotation of the plot. If the plot is to be drawn repeatedly, an advantage is that the TextItem annotation is drawn automatically each time you draw the plot. Also, whenever the plot object is moved or resized, the size and position of the TextItem is modified proportionally. It is as if the TextItem becomes an integral part of the plot object. On the other hand, if the plot object always maintains its original size, and you are only creating a few View objects, it may not be worth the extra code and overhead required to make the TextItem into an annotation.

Adding annotations to a plot

Once you have created the set of View objects intended to be annotations, there are two ways to add them as annotations of a plot. One is simply to set the array resource pmAnnoViews with an array containing each of the View object ids. The other is to add the annotations implicitly to the pmAnnoViews array, one at a time, by using the function NhlAddAnnotation, as in:

C:

```
anno_manager_id = NhlAddAnnotation(plot_id, view_id);
```

Fortran:

```
Call NhlFAddAnnotation(plot_id, view_id, anno_manager_id)
```

where "plot_id" is the identifier of the plot object, "view_id" is the identifier of the annotation object, and "anno_manager_id" is a returned identifier of the AnnoManager object that is created to manage the annotation.

For example, say you have created a MapPlot object whose identifier is "mapid". You want to add a TextItem annotation, so you create an object whose identifier is "textid". You then add the TextItem as an annotation of the MapPlot using NhlAddAnnotation. A Fortran code segment would look like this:

```
C
C  Create a plot object with the identifier mapid.
C
      call NhlFRLClear(rlist)
      call NhlFCreate(mapid,'Map0',NhlFMapPlotClass,wid,rlist,ierr)
C
C  Create a TextItem with identifier, textid.
C
      call NhlFRLClear(rlist)
      call NhlFRLSetstring(rlist,'txString',text,ierr)
      call NhlFCreate(textid,'Text0',NhlFtextitemClass,wid,rlist,ierr)
C
C  Add the TextItem to the MapPlot object as an annotation.
C
      call NhlFAddAnnotation(mapid,textid,annoid)
```

Alternately, you can use pmAnnoViews directly:

```
C
C  Create the TextItem with identifier, textid.
C
      call NhlFRLClear(rlist)
      call NhlFRLSetstring(rlist,'txString',text,ierr)
      call NhlFCreate(textid,'Text0',NhlFtextitemClass,wid,rlist,ierr)
C
C  Create the plot object, mapid, and assign textid to pmAnnoViews.
C
      call NhlFRLClear(rlist)
      call NhlFRLSetintegerarray(rlist,'pmAnnoViews',textid,1,ierr)
      call NhlFCreate(mapid,'Map0',NhlFMapPlotClass,wid,rlist,ierr)
```

Data-tracking annotations

By default, annotations are positioned and sized relative to the plot viewport. However, it is also possible to place the annotations relative to data coordinate space. For example, consider the case of a series of MapPlot frames that, when animated, shows a rotating globe. Suppose you want to label a number of world cities on this rotating globe.

In this situation, you would first need to create a TextItem for each label and set the txString resource to the name of the city. Then you would make the TextItem objects into a set of annotations that are positioned relative to the lat-lon coordinate system used as the data coordinates of MapPlot. To do this, you would set to True the amTrackData resource of each AnnoManager object created to manage the TextItem annotations. At the same time, you would set each of the AnnoManager objects' amDataXF and amDataYF resources to the longitude and latitude, respectively, of each city.

The ng4ex examples mp04c.c (C), and mp04f.f (Fortran), demonstrate this data-tracking process.

Removing or disabling annotations

Annotations can be removed from pmAnnoViews using the function NhlRemoveAnnotations (C), NhlFRemoveAnnotations (Fortran), or can simply be disabled by setting the AnnoManager resource amOn to False.

See also:

- PlotManager class
- HLU classes - alphabetically
- HLU API - alphabetically

Other classes

Most of these "other" classes are used by the classes that are specifically listed on the HLU User Guide tables of contents and roadmaps. These other classes typically allow for enhanced functionality in composite type classes. For example, the Workspace class helps in memory management. Transformation, MapTransformation, IrregularTransformation, and LogLinTransformation manage forward and reverse transformations. Base provides the base functionality used by all library objects.

As an HLU user, you may want to set resources of composite objects that include resources for some of these additional classes, but you will not need to create these objects as part of your HLU program. The composite resources available to any object are given in the Reference Manual descriptions of each object.

See also:

- HLU classes - alphabetically
- HLU API - alphabetically

Anatomy of a ContourPlot

Resources that control the various aspects of a contour plot relate to one of the following: the input data to be contoured, informational labels, the contouring operation, or composite objects. The ContourPlot resources description details all resources for this object. These resources can be logically reduced to the following categories:

- The input dataset to be contoured
    - The data object

      The data object associated with ContourPlot is ScalarField. The resource cnScalarFieldData specifies the id of a ScalarField data object. It must be set to create a contour plot.

    - Constant field data

      Often when the entire input dataset presented to ContourPlot is a constant value, this is an indication of a problem reading the dataset. There are about two dozen resources that allow you to configure how constant field labels appear on the plot. They begin with the prefix "cnConstF".

    - Missing data values

      NCAR Graphics utilities are written to allow for missing values in a dataset. Actions are taken to allow contouring to proceed by interpolating a value for the missing value. These resources begin with the prefix "cnMissingVal".

    - Raster contouring

      Contouring is normally done in a vector mode with polylines and filled polygons. However, contouring can also be done in raster mode with a color assigned to each cell of the two-dimensional data array based upon the data value. The resolution of the plot depends on the granularity of the data. These resources begin with a prefix of "cnRaster".

- Labels

  ContourPlot provides extreme flexibility in labeling of highs, lows, range of the data, and contouring levels. The resources that begin with the prefix "cnLabel" apply to all kinds of labels. In addition, there are resources for each of the following kinds of labels:

    - Information labels

      The information label typically states the range of data being contoured and the range of the interval between contours. These resources begin with the prefix "cnInfoLabel".

    - Labeling field highs

The relative maxima in the data field can be printed within masked boxes. These resources begin with the prefix "cnHighLabel".

- Labeling field lows

  The relative minima in the data field can be printed within masked boxes. These resources begin with the prefix "cnLowLabel".

- Labeling contour lines

  The field value of individual contour lines can be labeled. These resources begin with the prefix "cnLineLabel".

- Contouring
  - Contour levels

    These controls allow you to select the number of contour levels, and the field value increment between levels. Levels can have constant spacing or be set in an explicit manner. These resources begin with a prefix of "cnMaxLevel", "cnMinLevel", "cnMonoLevel", or "cnLevel".

  - Contour lines

    Lines can be drawn at the selected contour levels. These lines can be of differing widths, dash patterns, and colors. These resources begin with a prefix of either "cnLine" or "cnMonoLine".

  - Filling contour level regions

    The regions between contour levels can be pattern-filled or color-filled. These resources begin with a prefix of either "cnFill" or "cnMonoFill".

  - Contour smoothing

    Contour lines can be smoothed for appearance. These resources begin with a prefix of "cnSmoothing".

- Composite objects

  Composite objects associated with a ContourPlot object include TickMark, LabelBar, and Legend. TickMark resources begin with the prefix "tm". LabelBar resources begin with the prefix "lb". Legend resources begin with the prefix "lg".

  - The plot boundary

    ContourPlot uses the TickMark class to set up the plot boundary, scales, tick marks, and grid; however, ContourPlot has a few resources that override the TickMark resources with respect to the grid boundary. These resources begin with the prefix "cnGridBoundPerim".

- LabelBar ranges of filled levels

  A LabelBar showing contour level ranges is often included on contour plots that use color-filled or pattern-filled regions. In addition to the many resources of the LabelBar object, there are a couple ContourPlot resources specific to the use of a LabelBar. They are cnExplicitLabelBarLabelsOn and cnLabelBarEndLabelsOn.

- Legend ranges of contour lines

  A Legend showing contour line values is often included on plots that do not have color-filled or pattern-filled regions. In addition to the many resources of the Legend object, there are a couple ContourPlot resources specific to the use of a Legend. They are cnExplicitLegendLabelsOn and cnLegendLevelFlags.

Understanding resources

An object can be made to produce many different results. Just as a typical NCAR Graphics library subroutine can be manipulated through setting different arguments in the subroutine call or by parameters via a parameter-setting call, an object can be manipulated by entering different values for its resources. In fact, HLU object resources and LLU parameters are analogous. A typical resource in a View subclass might change a color, the width of a line, or the font of a label.

Three ways to set resources shows where resources can be set. They can be set in the HLU program in two ways. They can also be set in a third way, by using a resource file.

Applying resources to objects

Resource files are ascii files containing lines of data in which values are assigned to object resources. For example, the ng4ex example ti03c.c has an associated resource file ti03.res. The contents of ti03.res are a series of lines of the form:

```
ti03.ti03Work.Titles.tiMainString         : Title Resource Setting
```

This line sets the value of the resource tiMainString to "Title Resource Setting". Note that quotes are not needed when setting a string in a resource file.

When you assign a value to a resource, you have to make sure that you assign the value to the correct instantiation of the object. The generic assignment path for a Title resource is:

```
{App_obj_name}.{Workstation_obj_name}.{Title_obj_name}.{Resource_name}: Resource_
```

The App, Workstation, and Title objects are all assigned a name and id number which delineate the particular instance of that class.

For example, our HLU program ti03c would have the following three create calls in a C program:

```
NhlCreate(&appid,"ti03",NhlappClass,NhlDEFAULT_APP,rlist);
NhlCreate(&wid,"ti03Work",NhlxWorkstationClass,NhlDEFAULT_APP,rlist);
NhlCreate(&pid,"Titles", NhltitleClass,wid,rlist);
```

Argument 1 in these calls returns an object id number. Argument 2 defines an object name. The object names are used in resource linking. For more details, see NhlCreate function overview.

Call 1 sets the App object name for this instance to "ti03".
Call 2 sets the XWorkstation object name for this instance to "ti03Work".
Call 3 sets the Title object name for this instance to "Titles".

In this example, the resource file name will be the App object name with ".res" appended: ti03.res. The X Window where the title object will be displayed is named ti03Work. This particular title object is named Titles. Thus, the resource assignment linkage

```
{App_obj_name}.{Workstation_obj_name}.{Title_obj_name}.{Resource_name}: Resource_
```

becomes:

```
ti03.ti03Work.Titles.tiMainString
```

By adding a colon and the resource value after the resource name we get:

```
ti03.ti03Work.Titles.tiMainString   :   Title Resource Setting
```

Wild cards

Wild-card (*) designators are allowed for any of the objects. The interpretation is that the designated resource value will apply to all instances of a wild-carded object. Thus, the linkage might be *tiMainString, *Titles.tiMainString, *ti03Work.Titles.tiMainString, or ti03Work*tiMainString.

Instantiations of multiple objects

Consider the following examples of a main title resource:

```
*xwork.Title1.tiMainString          : Ozone Depletion
*pswork.Title2.tiMainString         : Satellite Tracking
*ncgmwork.Title1.tiMainString       : Ozone Depletion
```

In this case, there are three instances of Workstation objects and two instances of the Title object, Title1, and Title2. Title1 gets a main title string of "Ozone Depletion". Title 2 gets a main title of "Satellite Tracking". Title1 gets output to an X Workstation and an NCAR CGM file. Title2 goes to a PostScript file.

In these examples, the App object name has been replaced by a wild-card asterisk. Thus, the resource values we are assigning will be applied to all applications that might be created.

The wild-card replacement capabilities for applying resource values to objects can be very powerful. It is important to realize that the most specific definition of a resource will be used. This makes it possible to wild-card all instances of a particular type of object, but still directly specify one of them.

Assigning values to resources

Syntax for scalar resources

Scalar resources can be of the form integer, real, or string. To set any scalar resource, the value is simply placed after the colon in an assignment line of a resource file. For example,

```
ti02.ti02Work.Titles.tiMainFont              : times-roman
```

sets a string resource, tiMainFont, to a Times-Roman filled font for the main title of a Title object. A complete listing of all fonts available to the HLUs appears in the NCAR Graphics fonts table.

The line

```
ti02.ti02Work.Titles.tiMainFontHeightF      : .025
```

sets a scalar real resource, tiMainFontHeightF, to .025. This sets the height of the characters of the main title to .025 NDCs. Since a maximum viewport in NDC space is 1. by 1. this implies a title which is 1/40th of a maximum screen size.

The line

```
ti01.ti01Work.Titles.tiMainFontColor        : 1
```

sets the integer resource tiMainFontColor to 1. This selects the RGB color index 1 from the color table currently in place. If not set by the user, the HLU default color table will be used.

Syntax for array resources

Array resources can also be set in a resource file. The array of values begins with a left paren and a slash. The values are delimited by commas, and the array ends with a slash and right paren.

For example, consider the following resources from XyPlot example xy04:

Integer resource arrays

The line

```
xy04.xy04Work.xyPlot.xyData.xyLineColors:       (/50,75,100,40/)
```

shows the assignment of four integer values to an array resource xyLineColors. In this example, an XY plot with four curves is being created. The lines of the four curves are being assigned different color indices from the HLU color table. Unless the user creates and inputs a colormap, the HLU default color table is used.

Note that the above resource assignment path includes an additional object in the pathname. xyLineColors is a resource of an CoordArrays data object created in the HLU program. In C, the call would be:

```
NhlCreate(&dataid,"xyData",NhlcoordArraysClass,NhlDEFAULT_APP,rlist);
NhlCreate(&dataid,"xyData",NhlcoordArraysClass,NhlDEFAULT_APP,rlist);
```

In Fortran, the call would be:

```
call NhlFCreate(dataid,'xyData',NhlFCoordArraysClass,0,rlist,ierr)
```

In this case, the object path is application.workstation.view.data. The Class hierarchy diagrams module shows the hierarchical paths for HLU objects. If a resource can be applied to all instances of parent objects, you can use the path assignment shorthand of **resource_name* : *resource_value*. If you want to assign different resource values to different instances of objects, then you must determine the correct hierarchical path as we showed above when we assigned different main titles for two title objects and three workstation objects in the Instantiations of multiple objects section.

Real resource arrays

The line

```
xy04.xy04Work.xyPlot.xyData.xyLineThicknesses:  (/1.0,2.0,3.0,4.0/)
```

shows a similar real array assignment of four values. In this case, the four curves are getting line widths of 1. to 4. units. The default width of a line is 1. The curve with a width of 4. will be four times as thick as the curve with a width of 1.

String resource arrays

The lines

```
xy04.xy04Work.xyPlot.xyData.xyExplicitLabels:  (/Curve A,Curve B,Curve C/)
```

and

```
xy04.xy04Work.xyPlot.xyData.xyExplicitLabels:   (/1,2,3/)
```

show a string assignment of three values. In the first case, the curves are labeled Curve A, Curve B, and Curve C. In the second case, they are simply labeled 1, 2, and 3.

Resource line continuation

Arrays can span multiple lines in a resource file. Each line before the last one ends with a backslash to escape the carriage return. The end of the array is indicated by the /). There can be any number of blanks at the beginning or end of a line.

The line

```
xy04.xy04Work.xyPlot.xyData.xyLineThicknesses:  (/1.0,\
                                                  2.0,\
                                                  3.0,\
                                                  4.0/)
```

would also be a legal entry for the xyLineThicknesses array.

Note: In a continuation line, the carriage return must immediately follow the backslash delimiter.

Syntax for assigning a two-dimensional array resource

Suppose we want to use a color table other than the HLU default. We can do this through the Workstation resource wkColorMap. This is a two-dimensional array. The inner (fast varying) dimension is 3 elements (red, green, and blue.) The outer dimension is the number of elements that will exist in the new color table.

```
tx04.tx04work.wkColorMap:   (/(/ 0.00, 0.00, 0.00 /),\
                              (/ 0.66, 0.66, 0.66 /),\
                              (/ 0.40, 0.40, 0.40 /),\
                              (/ 0.00, 1.00, 1.00 /),\
                              (/ 0.20, 0.56, 0.80 /),\
                              (/ 0.00, 0.00, 1.00 /),\
                              (/ 0.50, 0.00, 1.00 /),\
                              (/ 1.00, 0.00, 1.00 /),\
                              (/ 0.14, 0.56, 0.14 /),\
                              (/ 0.00, 1.00, 0.00 /),\
                              (/ 1.00, 1.00, 0.00 /),\
                              (/ 0.86, 0.58, 0.44 /),\
```

```
(/ 0.65, 0.16, 0.16 /),\
(/ 1.00, 0.50, 0.00 /),\
(/ 1.00, 0.00, 0.00 /)/)
```

This color table has 15 colors which are: white, light gray, dark gray, cyan, sky blue, blue, blue magenta, magenta, forest green, green, yellow, tan, brown, orange, and red.

Note that the resource assignment path only includes the application (xy04) and the workstation (xy04Work) because this is a Workstation resource.

Ranges of resource values

In many cases, a resource has a limited range of values. For example, the options for tiMainFontQuality are HIGH, MEDIUM, and LOW. There are many fonts available from NCAR Graphics fonts, nine predefined colormaps, sixteen predefined dash patterns, sixteen predefined fill patterns, and seventeen predefined markers.

To find the choices available for a resource such as tiMainFontQuality, you can navigate through the online documentation as follows:

- Start from Doc Home
- Click on "HLU reference documents"
- Click on "HLU classes - alphabetically"
- Click on "Title"
- (Page down to tiMainFontQuality)
- Click on the type of this resource if it is linked.
- If it is not linked, or is linked but does not show the options, Click on the name of the resource. It may show the options as part of the resource definition.
- If neither of these show the options, press the comment "ear" icon and send a documentation error report to the NCAR Graphics staff.

Composite resources

In our discussions of class hierarchy, we have described a composite class. It has access to the resources of its composite member classes. Class hierarchy diagrams shows that the XyPlot class has composite member classes of PlotManager, Legend, LabelBar, TickMark, and Title. It also has superclasses of DataComm, Transform, View, and Base, from which resources are inherited. Thus, within the resource file of an XyPlot object, we can set the values of resources for any of these classes. Consider these resource value assignments from the ng4ex example file xy06.res:

```
xy06.xy06Work.xyPlot.vpXF:                      0.15
xy06.xy06Work.xyPlot.vpYF:                      0.9
xy06.xy06Work.xyPlot.vpWidthF:                  0.8
xy06.xy06Work.xyPlot.vpHeightF:                 0.8
xy06.xy06Work.xyPlot.tiMainString:              Example xy06
xy06.xy06Work.xyPlot.tiXAxisString:             X
xy06.xy06Work.xyPlot.tiYAxisString:             Y
xy06.xy06Work.xyPlot.xyDataDep.xyDashPattern:   0
xy06.xy06Work.xyPlot.tmXBMinorOn:               False
```

```
xy06.xy06Work.xyPlot.tmYLMinorOn:              False
xy06.xy06Work.xyPlot.tmXTMinorOn:              False
xy06.xy06Work.xyPlot.tmYRMinorOn:              False
xy06.xy06Work.xyPlot.tmXBMode:                 EXPLICIT
xy06.xy06Work.xyPlot.tmYLMode:                 EXPLICIT
```

See also:

- Three ways to set resources
- Resource types
- Resource files
- Resource database
- HLU classes - alphabetically
- HLU API - alphabetically

Three ways to set resources

Resources can be set in three ways:

- In your program code, using a resource list argument in an NhlCreate call.

  First you load specific resources and the values to be assigned to those resources into a resource list (rlist in this case). Then you issue a call to create an object with the resource list as an argument.

  Examples:

  ```
  Fortran:    call nhlfrlsetfloat(rlist,'vpXF',.2,ierr)
              call nhlfrlsetfloat(rlist,'vpYF',.8,ierr)
              call nhlfrlsetfloat(rlist,'vpWidthF',.6,ierr)
              call nhlfrlsetfloat(rlist,'vpHeightF',.6,ierr)
              call nhlfcreate(pid,'Titles',nhlftitleclass,wid,rlist,ierr)

  C:          NhlRLSetFloat(rlist,NhlNvpXF,.2);
              NhlRLSetFloat(rlist,NhlNvpYF,.8);
              NhlRLSetFloat(rlist,NhlNvpWidthF,.6);
              NhlRLSetFloat(rlist,NhlNvpHeightF,.6);
              NhlCreate(&pid,"Titles",NhltitleClass,wid,rlist);
  ```

  Note that in C, the full resource name, including the NhlN prefix, is given. The NhlN prefix is a C macro for the string. If you wanted, you could use the short name in double quotes. However, the advantage of using the NhlN define is that the compiler will catch misspellings. In Fortran, the short name is enclosed in single quotes.

  In this example, we are setting the upper left corner of a device viewport (vpXF,vpYF), then specifying the width and height of the viewport, vpWidthF and vpHeightF.

- In your program code, using a resource list argument in an NhlSetValues call.

  Again you load specific resources and the values to be assigned to those resources into a resource list (rlist in this case). Then you issue a call to set the resource values with rlist as an argument.

  Examples:

```
Fortran:    call nhlfrlsetfloat(rlist,'vpXF',.2,ierr)
            call nhlfrlsetfloat(rlist,'vpYF',.8,ierr)
            call nhlfrlsetfloat(rlist,'vpWidthF',.6,ierr)
            call nhlfrlsetfloat(rlist,'vpHeightF',.6,ierr)
            call NhlFSetValues(pid,rlist,ierr)

C:          NhlRLSetFloat(rlist,NhlNvpXF,.2);
            NhlRLSetFloat(rlist,NhlNvpYF,.8);
            NhlRLSetFloat(rlist,NhlNvpWidthF,.6);
            NhlRLSetFloat(rlist,NhlNvpHeightF,.6);
            NhlSetValues(pid,rlist);
```

- In a resource file.

  The third method is to set the resource value in a resource file. This is the recommended method for most cases because resource files are read by the program as data. Thus, you can change resource values and re-execute the program module without having to recompile, link, and load. If you change a resource value through the functional API, then you must recompile, etc. Clearly the resource file approach is more efficient in user and machine time.

  The disadvantage of using resource files is that errors, such as misspellings, are not reported, and if a resource value changes during execution, resource files are not adequate.

The Resource table module is an alphabetical list of all currently available resources.

See also:

- Understanding resources
- Resource types
- Using resource files
- Resource database
- HLU classes - alphabetically
- HLU API - alphabetically

Resource types

Every resource has one of the following basic types:

- Integer
- Real

- String
- Enumerated (a number represents a string)
- Array
- 2D array

These string values for these types and enumerations are defined in the HLU library and listed in the HLU types module. The HLU library also contains many enumeration values; these are listed in the HLU enumeration values table.

Resources that have an enumerated type can be set using the SetInteger function with the enumerated value or the integer value, or using the SetString function with the string value. Note that the set of characters that are enclosed in quotes in the C comment field is the string representation of the enumerated value. This string value can be used in the SetString function, and the string value is case insensitive.

For example, the cnConstFLabelFontQuality resource has a type NhlTFontQuality. NhlTFontQuality is an enumerated type defined as follows:

```
typedef enum {
    NhlHIGH          = 0,     /* "High"        */
    NhlMEDIUM        = 1,     /* "Medium"      */
    NhlLOW           = 2      /* "Low"         */
} NhlFontQuality;
```

So, for example, the resource could be set (in C) to "Medium" in any of 3 ways:

1. NhlRLSetString(list_id, "cnConstFLabelFontQuality", "Medium");
2. NhlRLSetInteger(list_id, "cnConstFLabelFontQuality", NhlMEDIUM);
3. NhlRLSetInteger(list_id, "cnConstFLabelFontQuality", 1);

Where "list_id" is the integer id of the resource list to add the name/value pair to.

Since the C enumeration values cannot be used in FORTRAN programs, FORTRAN programmers would have to use one of the following calls to set the cnConstFLabelFontQuality resource:

```
NhlFRLSetString(list_id, 'cnConstFLabelFontQuality','Medium', ierr)
```

or

```
NhlFRLSetInteger(list_id, 'cnConstFLabelFontQuality', 1, ierr)
```

where "list_id" is the integer id of the resource list to add the name/value pair to, and "ierr" is the integer value error code that is returned by the function.

See also:

- Understanding resources
- Three ways to set resources
- Using resource files
- Resource database

- NhlRLSet
- NhlRLSetArray
- HLU classes - alphabetically
- HLU API - alphabetically

Using resource files

Resource file hierarchy

To provide maximum flexibility in applications and reasonable default values for resources, a series of up to four resource files can exist. One is created by the software developer. The others can be set by application project managers and project staff. These files provide default resource values.

The four resource files

1. $(NCARG_SYSRESFILE)
2. System Application Specific Resource file.
3. $(NCARG_USRRESFILE)
4. Users Application Specific Resource file ("*name*.res")

Resource file 1 is intended for use by the designers/installers of NCAR Graphics to set resources that should apply globally at a site. For example, an NCAR Graphics site installer may decide to turn on printing of all error messages.

Resource file 2 is intended to be used by developers of multi-user applications to set resources specific to an application for all users of that application. For example, the developer may want to default the viewport resources for a particular output device.

Resource files 3 and 4 are for users; 3 allows the user to provide default resources for all HLU applications, similar to Resource file 1, $(NCARG_SYSRESFILE). 4 allows the user to provide resources on an application-specific basis, similar to Resource file 2, the System Application Specific Resource file.

The final value assigned to each resource is determined by proceeding from file 1 through file 4 with each succeeding value taking precedence over the previous one.

Resources set in the HLU code override the resource file

However, the final word concerning the value of a resource occurs in the HLU program. Resource values set in the code supersede resource values set in resource files. Thus, if you change the color of a line in a resource file, re-execute the program module, but discover that the line color does not change, inspect your HLU program to make sure that you are not setting the line color resource in your program.

Each HLU program needs an application (App) object. It is either created through an NhlOpen call, or explicitly created in your program through an NhlCreate call after an NhlInitialize call as in:

```
Fortran:    call nhlfcreate(appid,'xy05',nhlfappclass,0,rlist,ierr)

C:          NhlCreate(&appid,"xy05",NhlappClass,NhlDEFAULT_APP,rlist);
```

The resource list, rlist, in the above examples could contain the pathnames of the system application specific resource file (file 2), and the users application specific resource file (file 4), as in:

```
Fortran:    call nhlfrlsetstring(rlist,'appUsrDir','./',ierr)
Fortran:    call nhlfrlsetstring(rlist,'appFileSuffix','.res1',ierr)
Fortran:    call nhlfrlsetstring(rlist,'appSysDir','~/',ierr)

C:          NhlRLSetString(rlist,NhlNappUsrDir,"./");
C:          NhlRLSetString(rlist,NhlNappFileSuffix,".res1");
C:          NhlRLSetString(rlist,NhlNappSysDir,"~/");
```

Adding resource values in the rlist is simply adding them to a dynamic array that is not actually applied until the NhlFCreate or NhlCreate call. Note that we can also set the suffix of a type 4 file. The default value is .res.

These resources are also settable from the resource database; however, they cannot be specified in the application-specific resource files (2 and 4) because these resources are used to find resource files 2 and 4.

Directory locations of resource files

$NCARG_ROOT is the directory in which NCAR Graphics is installed on your system. You must put a statement of the form

```
setenv NCARG_ROOT /path/name/where/installed
```

in your .cshrc file. We can then locate the default resource files relative to this install location.

1. Environment variable $(NCARG_SYSRESFILE) specifies the pathname of resource file 1. The default file $NCARG_ROOT/lib/ncarg/sysresfile will be used if $(NCARG_SYSRESFILE) is not set. We suggest that you use the default resource file that has been created for you by the developers.
2. The appSysDir resource of an App object is used to specify the directory where resource file 2 is located. This resource defaults to the value of the $(NCARG_SYSAPPRES) environment variable. See sysappres.
3. Environment variable $(NCARG_USRRESFILE) specifies the pathname of resource file 3. $(NCARG_USRRESFILE) defaults to ~/.hluresfile if not set by the user.
4. The appUsrDir resource of an App object is used to specify the the directory where resource file 4 is located. This resource defaults to the value of the current working directory (./). For example,

   ```
   NhlCreate(&appid,"ti01",NhlappClass,NhlDEFAULT_APP,rlist);
   ```

   shows the creation of an application named ti01. Thus, if the resource file suffix for this application is allowed to default to .res, the file will be named ti01.res.

The resource file pathnames can be retrieved by typing:

```
% ncargpath sysresfile
% ncargpath sysappres
% ncargpath usrresfile
```

The file usrresfile will not exist unless the user creates it.

See also:

- Understanding resources
- Three ways to set resources
- Resource types
- Resource database
- HLU classes - alphabetically
- HLU API - alphabetically

Understanding the Application Programming Interface

HLU Application Programming Interface (API) overview

There are several classes that are important in the writing of HLU programs: application class, data classes, workstation class, and view class. The application class (App) allows user access to resource databases; it allows for the specification of resource datafile pathnames. Objects belonging to the data class (DataItem) define and input the data to be plotted. Objects of the view class create the plots. Objects of the workstation class set up the destination devices for the plots.

Version 4.1 contains the following classes that you may want to use in your HLU program:

- Application class
    - App - Used to manage the resource database and other application-specific attributes.
- DataComm and DataItem classes

  The DataComm class provides the functionality for its subclasses to communicate with the DataItem class. The DataItem class provides a way to describe data to the HLU library.

  At this writing, there are four DataItem subclasses:

    - ScalarField - Defines a 2-D dataset as is needed by the ContourPlot class from C or Fortran.
    - VectorField - Defines a 2-D dataset as is needed by the VectorPlot class or StreamlinePlot

class from C or Fortran.

- CoordArrays - A method for importing X and Y arrays of data from C or Fortran. It uses two arrays, caXArray and caYArray, to describe data to the HLU library. It is often used in combination with the XyPlot class.

- CoordArrTable - A method for importing X and Y arrays of data (C interface only). It is often used in combination with the XyPlot class. It is useful in defining multiple coordinate arrays of different lengths.

- Workstation subclasses

  There are currently three devices to which graphical output can be sent: an NCAR Computer Graphics Metafile (NCGM), a PostScript file, and an X11 window on the workstation monitor.

  - NcgmWorkstation - Draw commands are output to an NCAR Computer Graphics Metafile (NCGM).

  - PSWorkstation - Draw commands are output to a PostScript file.

  - XWorkstation - Draw commands are output to an X11 workstation window.

- View subclasses

  There are two kinds of View classes; one needs data from a DataItem object, and the other does not.

  View objects that require a DataItem object:

  - ContourPlot - Draws a wide range of 2-D contour plots.

  - StreamlinePlot - Draws 2-D streamlines.

  - VectorPlot - Draws vectors at 2-D grid points.

  - XyPlot - Draws plots of field X versus field Y.

  View objects that do not require data input include:

  - GraphicStyle - controls the appearance of graphics primitives such as line, fill, marker, and text.

  - LabelBar - Draws boxes filled with colors or patterns and labels that relate the boxes to plot features.

  - Legend - Draws dashed lines and markers and associated labels that relate the lines/markers to plot features.

  - MapPlot - Draws world and regional maps in many scientifically useful projections.

- TextItem - Draws strings of text.
- TickMark - Draws plot grids with scaled, ticked, and labeled axes.
- Title - Draws main, X, and Y axis titles.

- Error class
    - Error - Used for logging and reporting errors.
- Transformation class
    - Transform - Provides subclassed objects with the ability to handle coordinate transformations.

      There are a number of support functions associated with the Transform class that allow data to be defined in one space and retrieved in another, or for polylines to be drawn.
- PlotManager class
    - PlotManager - Manages overlays of multiple graphical objects.
- AnnoManager class
    - AnnoManager - Manages the overlay of annotational items and graphical objects.
- Other classes

  There are additional classes in the HLU library. Most play a behind-the-scenes role. For a complete list see:

- HLU classes - alphabetically
- HLU API - alphabetically

API structure (HLU function interface)

Functions that apply to all HLU classes are called base functions. This is because they are defined for all objects that are members of the Base class. There are specific functions that apply to only one type of class. There is one function, NhlDraw, that applies to two high-level classes that contain many subclasses. Finally, there are a few general functions which are not class-related. The functions are listed below by category.

Base API functions

Most functions in the HLU API are specific to individual class; however, the following four functions apply to every HLU class:

- NhlCreate

  Creates an instance of any HLU object.

- NhlSetValues

  Modifies the current values of resources for any HLU object.

- NhlGetValues

  Retrieves the current values of resources for any HLU object.

- NhlDestroy

  Destroys any HLU object, freeing all memory and system resources used by the object.

API functions used by more than one class

- NhlDraw

  This function is called by the user to produce the graphical output. This function is available for View and Workstation objects. If it is called on the View class, that object's graphical output is sent to the workstation. If it is called on the Workstation class, all of the Workstation children are drawn.

Specific API functions by class

Application class

- NhlAppGetDefaultParent

  Retrieves the ID of the current default parent App object.

Data class

- NhlAddData

  Associates a DataItem with a data resource.

- NhlRemoveData

  Unassociates a DataItem from a data resource.

- NhlUpdateData

Tells a DataComm object to update its internal state with the changes of any DataItem objects that are associated with its data resources.

Workstation class

- NhlChangeWorkstation

  Re-routes graphical output to another workstation. It can be used to output a graphic to more than one workstation type.

- NhlUpdateWorkstation

  Updates the drawing area of an instance of a Workstation object. It flushes any output that has been sent to the workstation but that has not yet been displayed.

- NhlClearWorkstation

  Clears the drawing area of an instance of a Workstation object.

- NhlFrame

  Updates and then clears the drawing area of an instance of a Workstation object.

- NhlNewColor

  Adds a color specified as an RGB triple to the colormap of a Workstation instance and returns its HLU color index.

- NhlSetColor

  Sets the RGB color values for a specified HLU color index.

- NhlFreeColor

  Removes a color, specified by its HLU index, from the Workstation color map.

- NhlIsAllocatedColor

  Returns True if a color has been assigned to a particular HLU color index, and False otherwise. (IsAllocatedColor is a boolean function.)

See also XWorkstation, NcgmWorkstation, and PSWorkstation.

View class

- NhlGetBB

Given the id of a View object, returns the coordinates of the left, right, top, and bottom edges of the object's bounding box in NDC units.

Transform class

- NhlAddAnnotation

  Adds an arbitrary View object to a plot object as an external annotation, and returns the id of the AnnoManager object created to manage it.

- NhlRemoveAnnotation

  Removes a View object annotation from a plot object, destroying the AnnoManager object created to manage it.

- NhlAddOverlay

  Adds a Transform object as an overlay plot member to a base plot object.

- NhlRemoveOverlay

  Removes an overlay plot member from a base plot object.

Error class

- NhlPError, NHLPERROR, NhlErrGetId, NhlErrNumMsgs, NhlErrGetMsg, NhlErrCleanMsgs, NhlErrAddTable, NhlErrSPrintMsg, NhlErrFPrintMsg

Workspace class

- NhlGetWorkspaceObjectId

  Retrieves the id of the Workspace object.

API functions not related to class

There are several important functions that are not class-related; these include:

- NhlOpen

  Opens and initializes the HLU library. If called, it must be the first API function called in an HLU user program. NhlOpen automatically creates an App object which attempts to access the level 1 and 3 application resource files whose pathnames are defined by the $NCARG_SYSRESFILE and $NCARG_USRRESFILE variables. (See alternative call NhlInitialize.)

- NhlInitialize

Opens and initializes the HLU library. NhlInitialize differs from NhlOpen in that the programmer must create an App object using NhlCreate before creating any other objects.

- NhlClose

  Closes the HLU library. It must be the last API function called in an HLU user program.

See also:

- HLU classes - alphabetically
- HLU API - alphabetically

Writing an HLU program

This module describes general steps to follow in writing an HLU program; it provides examples for writing a TextItem program in Fortran and C.

1. Write an HLU program in either Fortran or C using API functions. The logical program steps are:

   - Select a header file that matches each object type to be created.
   - Open the HLU library.
   - Create an App object to provide access to the resource database (optional).
   - Create a DataItem object (only needed for XyPlot or ContourPlot objects).
   - Create workstation object(s) (XWorkstation, NcgmWorkstation, or PSWorkstation).
   - Create view object(s) (ContourPlot, XyPlot, TextItem, etc.).
   - Assign resources to objects in the code or in a resource file.
   - Draw to send the graphics to the output workstation.
   - Frame to cause a pause or erase (X11), or a picture advance (PS, NCGM).
   - Destroy all created objects.
   - Close the HLU library.

   A recommended approach is to look at the many examples existing under ng4ex and select the one which is closest to your desired task.

2. Create a resource data file. Set resources to desired values.

3. Compile and execute the program using nhlcc, nhlf77, or the make files.

4. Run the program module and look at the graphical output.

5. Change resources in the resource file and re-execute the program module.

Components of a TextItem program

Building a Fortran program

1. Select the needed objects.

   First, let's set our resources in a local resource file which is specific to this application. Thus we will use the combination of NhlInitialize and an NhlCreate of an app object. We could also have chosen to use NhlOpen and set the resources in the file whose pathname is given by system variable $(NCARG_USRRESFILE). Or we could choose to set resources directly in the program using arguments in Create calls or by using the NhlSetValues functions.

   We do not need a data object since TextItem does not import data. The text string to be written is merely entered as a resource.

   Next, we need a workstation object in which to draw or write our graphical instructions. Suppose we just choose an X11 window which is an XWorkstation object.

   Finally, we need a view object, namely TextItem.

2. Select the needed header files.

   Now that we know which objects we need, we can define the header files that are needed; these include:

   Fortran:

   ```
   external nhlfappclass

   external nhlfxworkstationclass

   external nhlftextitemclass
   ```

   App, TextItem, and XWorkstation are the classes that we are going to use in this program.

   Fortran headers lists the headers for all objects.

3. Open and initialize the HLU library.

   Fortran:

   ```
   call nhlfinitialize
   ```

4. Create an application context. Set the app dir to the current directory so the application looks for a resource file in the working directory.

   Fortran:

   ```
   call nhlfrlcreate(rlist,'setrl')

   call nhlfrlclear(rlist)

   call nhlfrlsetstring(rlist,'appDefaultParent','True',ierr)

   call nhlfrlsetstring(rlist,'appUsrDir','./',ierr)

   call nhlfcreate(appid,'tx01',nhlfappclass,0,rlist,ierr)
   ```

   This section of code creates a resource list needed to configure the application object, clears the list, sets the NhlNappUsrDir resource of the App class, with a NhlRLSetString call, to the current working directory ("./"), then creates an application object whose id is returned within the parameter appid. Since the name of the application is set to "tx01" in the create call, the resource file is named tx01 with a .res appended: tx01.res. This app object is of type NhlappClass.

   When we run this program, it will expect to find a resource file named tx01.res in the current working directory.

5. Create an XWorkstation object.

   Fortran:

   ```
    call nhlfrlclear(rlist)

    call nhlfrlsetstring(rlist,'wkPause','True',ierr)

    call nhlfcreate(wid,'tx01Work',nhlfxworkstationclass,

   $      0,rlist,ierr)
   ```

   This section of code clears the resource list, sets a resource which will cause the XWorkstation object to pause when NhlFrame is called on the the XWorkstation object. This is done by setting the XWorkstation resource NhlNwkPause to True. The XWorkstation object is then created an XWorkstation object. The id is returned in wid. This is an object of type NhlxWorkstationClass.

6. Set any resources in rlist (optional) and create a TextItem object.

   Fortran:

   ```
   call nhlfrlclear(rlist)

   call nhlfrlsetfloat(rlist,'txPosXF',.5,ierr)

   call nhlfrlsetfloat(rlist,'txPosYF',.5,ierr)
   ```

```
call nhlfrlsetfloat(rlist,'txFontHeightF',.2,ierr)
call nhlfrlsetstring(rlist,'txString','NCAR Graphics',ierr)
call nhlfcreate(pid,'TextItems',nhlftextitemclass,
$     wid,rlist,ierr)
```

This section of code clears the resource list, defines the position to locate the output text string in the X11 window. The position is set to (.5,.5) in (X,Y) viewport coordinates, called normalized device coordinates. The entire viewport is 0. to 1. in both dimensions. This section also sets the height of the output string to be .2 or approximately one fifth of the X11 window. The resources NhlNtxPosXF, NhlNtxPosYF, and NhlNtxFontHeightF are added to rlist using the NhlRLSetFloat function and NhlNtxString is set using the NhlRLSetString functions. The TextItem object is then created and returns its id in pid. This is an object of type NhltextItemClass, it has the parent wid, and its resources are set by rlist.

Note: The location, height and string resources could also be set in the resource file tx01.res.

7. Output the graphics to the X11 window.

Fortran:

```
call nhlfdraw(pid,ierr)           ...or call nhlfdraw(wid,ierr)
call nhlfframe(wid,ierr)
```

The draw function causes the TextItem object, id = pid, to be drawn. The Frame function flushes the output device. Since we set the NhlNwkPause XWorkstation resource to True, the plot will be drawn and held in the window awaiting a mouse click in the window.

8. Destroy all objects that are no longer needed (optional).

If you intend to continue the program and create some new and different App, Plot, or Workstation objects, and if you no longer need any of the current objects, then good programming practice is to delete the objects that are no longer required in order to keep the memory size of your program down. If you are about to end the program, go to Step 9.

Fortran:

```
call nhlfdestroy(pid,ierr)
call nhlfdestroy(wid,ierr)
call nhlfdestroy(appid,ierr)
```

9. Close the HLU library and exit.

Fortran:

```
call nhlfclose

stop

end
```

This section of code destroys all existing objects, closes the HLU library, and exits our program.

Building a C program

1. Select the needed objects.

   First, let's set our resources in a local resource file that is specific to this application. Thus we will use the combination of NhlInitialize and an NhlCreate of an app object. We could also have chosen to use NhlOpen and set the resources in the file whose pathname is given by system variable $(NCARG_USRRESFILE). Or we could choose to set resources directly in the program using arguments in Create calls or by using the NhlSetValues functions.

   We do not need a data object since TextItem does not import data. The text string to be written is merely entered as a resource.

   Next, we need a workstation object in which to draw or write our graphical instructions. Suppose we just choose an X11 window which is an XWorkStation object.

   Finally, we need a view object, namely TextItem.

2. Select the needed header files.

   Now that we know which objects we need, we can define the header files that are needed; these include:

   C:

   #include <ncarg/hlu/hlu.h>

   The HLU library header is always required.

   C:

   ```
   #include <ncarg/hlu/App.h>

   #include <ncarg/hlu/TextItem.h>

   #include <ncarg/hlu/XWorkstation.h>
   ```

   App, TextItem, and XWorkstation are the objects we are going to use in this program.

   C headers lists the headers for all objects.

3. Open and initialize the HLU library.

   C:

```
NhlInitialize();
```

4. Create an application context. Set the app dir to the current directory so the application looks for a resource file in the working directory.

   C:

```
rlist = NhlRLCreate(NhlSETRL);

NhlRLClear(rlist);

NhlRLSetString(rlist,NhlNappDefaultParent,"True");

NhlRLSetString(rlist,NhlNappUsrDir,"./");

NhlCreate(&appid,"tx01",NhlappClass,NhlDEFAULT_APP,rlist);
```

   This section of code creates a resource list needed to configure the application object, clears the list, sets the NhlNappUsrDir resource of the App class, with a SetString call, to the current working directory ("./"), then creates an application object whose id is returned within the parameter appid. Since the name of the application is set to "tx01" in the create call, the resource file is named tx01 with a .res appended: tx01.res. This app object is of type NhlappClass.

   When we run this program, it will expect to find a resource file named tx01.res in the current working directory.

5. Create an XWorkstation object.

   C:

```
NhlRLClear(rlist);

NhlRLSetInteger(rlist,NhlNwkPause,True);

NhlCreate(&wid,"txxwork",NhlxWorkstationClass,NhlDEFAULT.APP,rlist);
```

   This section of code clears the resource list, sets a resource which will cause the XWorkstation object to pause when NhlFrame is called on the the XWorkstation object. This is done by setting the XWorkstation resource NhlNwkPause to True. The XWorkstation object is then created an XWorkstation object. The id is returned in wid. This is an object of type NhlxWorkstationClass.

6. Set any resources in rlist (optional) and create a TextItem object.

   C:

```
NhlRLClear(rlist);
```

```
NhlRLSetFloat(rlist,NhlNtxPosXF,.5);
NhlRLSetFloat(rlist,NhlNtxPosYF,.5);
NhlRLSetFloat(rlist,NhlNtxFontHeightF,.2);
NhlRLSetString(rlist,NhlNtxString,"NCAR Graphics");
NhlCreate(&pid,"TextItems",
          NhltextItemClass,wid,rlist);
```

This section of code clears the resource list, defines the position to locate the output text string in the X11 window. The position is set to (.5,.5) in (X,Y) viewport coordinates, called normalized device coordinates. The entire viewport is 0. to 1. in both dimensions. This section also sets the height of the output string to be .2 or approximately one fifth of the X11 window. The resources NhlNtxPosXF, NhlNtxPosYF, and NhlNtxFontHeightF are added to rlist using the NhlRLSetFloat function and NhlNtxString is set using the NhlRLSetString functions. The TextItem object is then created and returns its id in pid. This is an object of type NhltextItemClass, it has the parent wid, and its resources are set by rlist.

Note: The location, height and string resources could also be set in the resource file tx01.res.

7. Output the graphics to the X11 window.

   C:

```
NhlDraw(pid);
NhlFrame(wid);
```

   The draw function causes the TextItem object, id = pid, to be drawn. The frame function flushes the output device. Since we set the NhlNwkPause XWorkstation resource to True, the plot will be drawn and held in the window awaiting a mouse click in the window.

8. Destroy all objects that are no longer needed (optional).

   If you intend to continue the program and create some new and different app, plot, or workstation objects, and if you no longer need any of the current objects, then good programming practice is to delete the objects that are no longer required in order to keep the memory size of your program down. If you are about to end the program, go to Step 9.

   C:

```
NhlDestroy(appid);
NhlDestroy(pid);
NhlDestroy(wid);
```

9. Close the HLU library and exit.

   C:

```
NhlClose();

exit(0);
```

   This section of code destroys all existing objects, closes the HLU library, and exits our program.

See also:

- ng4ex command
- HLU classes - alphabetically
- HLU API - alphabetically

HLU header files

In either C or Fortran programs, header files must be included for objects that are created. In C programs, the header file is preceded by an #include. In Fortran programs, the header file is an external statement. Tables of C and Fortran headers are given below.

C headers

```
AnnoManager.h
App.h
Base.h
ContourPlot.h
CoordArrTable.h
CoordArrays.h
DataComm.h
DataItem.h
Error.h
GraphicStyle.h
IrregularPlot.h
IrregularTransObj.h
LabelBar.h
Legend.h
LogLinPlot.h
LogLinTransObj.h
MapPlot.h
MapTransObj.h
NcgmWorkstation.h
PlotManager.h
PSWorkstation.h
ResList.h
ScalarField.h
StreamlinePlot.h
Style.h
TextItem.h
```

```
TickMark.h
Title.h
TransObj.h
Transform.h
VectorField.h
VectorPlot.h
View.h
Workspace.h
Workstation.h
XWorkstation.h
XyPlot.h
hlu.h
```

Fortran external class functions

```
external nhlfAnnoManagerClass
external nhlfAppClass
external nhlfContourPlotClass
external nhlfCoordArraysClass
external nhlfGraphicStyleClass
external nhlfIrregularPlotClass
external nhlfLabelBarClass
external nhlfLegendClass
external nhlfLoglinPlotClass
external nhlfMapPlotClass
external nhlfNcgmWorkstationClass
external nhlfPSWorkstationClass
external nhlfScalarFieldClass
external nhlfStreamlinePlotClass
external nhlfStyleClass
external nhlfTextItemClass
external nhlfTickMarkClass
external nhlfTitleClass
external nhlfVectorFieldClass
external nhlfVectorPlotClass
external nhlfXWorkstationClass
external nhlfXyPlotClass
```

How objects are used in an HLU program

This module provides some insight into the relationships between objects in HLU programs.

- The App object is used to manage the resource database that will be used by all of the children of the App object.

  An HLU programmer can choose to use, or not use, a local application resource file. There are two cases in which such a file is not needed:
  1. If the default values of all resources are acceptable
  2. All resources to be changed are changed in the program through the appropriate NhlSetValues function calls or NhlCreate calls

- A DataItem object is used to bring in data for Plot objects such as ContourPlot, StreamlinePlot, VectorPlot, or XyPlot.

- A Workstation object is used to direct the graphics to an X Workstation, to an NCGM, or to a PostScript file.

- View objects are used to generate the graphics. One or more View objects can be drawn before a frame is completed. In this context, a frame is either the extent of graphics sent to an X Window before a pause or window clear is initiated, or it is the completion of an NCGM image or PostScript page.

- The Error object is created automatically. It is used to buffer and output error messages, among other things.

See also:

- HLU classes - alphabetically
- HLU API - alphabetically

HLU user environment

HLU user environment

Environment variables

Every user of NCAR Graphics will either need to set the environment variable NCARG_ROOT, or the three environment variables NCARG_BIN, NCARG_LIB, and NCARG_INCLUDE. Which is done depends on how NCAR Graphics was installed on your system. To determine how these variables have been set, execute:

```
env
```

A list of all the environment variables that have been set will be shown. If none of the above variables appear, please contact your system administrator to determine where NCAR Graphics has been installed.

If the NCAR Graphics binaries, libraries, and include files were installed under one root directory, say for example /usr/local/bin, /usr/local/lib, and /usr/local/include, respectively, then you will need to do the following:

```
setenv NCARG_ROOT /usr/local
```

If the binaries, libraries, and include files were NOT installed under one root directory, say for example they are in /usr/local/ncarg/bin, /usr/local/ncarg/lib, and /usr/local/include, then you would need to do the following:

```
setenv NCARG_BIN /usr/local/ncarg/bin
setenv NCARG_LIB /usr/local/ncarg/lib
setenv NCARG_INCLUDE /usr/local/include
```

The environment variable NCARG_GKS_OUTPUT may be used to direct the metafile output from the NCAR Graphics GKS package to a file (use just the file name) or to pipe it to a translator (give the name of the translator, preceded by a "|").

The environment variable NCARG_GKS_PSOUTPUT may be used to direct the PostScript output from the NCAR Graphics GKS package to a file (stdout is allowed).

The environment variables GRAPHCAP and FONTCAP may be used to tell the NCGM translators "ctrans", "ictrans" and "idt" what default "graphcap" and "fontcap" to use. See the man pages for ctrans(1NCARG), ictrans(1NCARG), and idt(1NCARG) for more information.

The environment variable DISPLAY is not actually used by the translators; if you are using X Windows, it determines where the translator output is to be displayed.

Resource files

There are two resource files which can be installed at system installation time. They are:

- $(NCARG_SYSRESFILE)
- $(NCARG_USRRESFILE)

If these files have been created, you can use the ncargpath command to locate them as in "ncargpath NCARG_SYSRESFILE", or "ncargpath NCARG_USRRESFILE".

Important man files

The most important file under the UNIX man facility is ncargintro. If you issue the command "man ncargintro" it will list for you all of the more important components to be found in the NCAR Graphics package.

See also:

- HLUs vs. LLUs

HLU plot creation sequence

Components of a TextItem program

Building a Fortran program

1. Select the needed objects.

   First, let's set our resources in a local resource file which is specific to this application. Thus we will use the combination of NhlInitialize and an NhlCreate of an App object. We could also have chosen to use NhlOpen and set the resources in the file whose pathname is given by system variable $(NCARG_USRRESFILE). Or we could choose to set resources directly in the program using arguments in Create calls or by using the NhlSetValues functions.

   We do not need a data object since TextItem does not import data. The text string to be written is merely entered as a resource.

   Next, we need a workstation object in which to draw or write our graphical instructions. Suppose we just choose an X Window, which is an XWorkstation object.

   Finally, we need a View object, namely TextItem.

2. Select the needed header files.

   Now that we know which objects we need, we can define the header files that are needed; these include:

   Fortran:

   ```
   external nhlfappclass
   external nhlfxworkstationclass
   external nhlftextitemclass
   ```

   App, XWorkstation, and TextItem are the classes we are going to use in this program.

   Fortran headers lists the headers for all objects.

3. Open and initialize the HLU library.

   Fortran:

   ```
   call nhlfinitialize
   ```

4. Create an application context. Set the app dir to the current directory so the application looks for a resource file in the working directory.

   Fortran:

   ```
   call nhlfrlcreate(rlist,'setrl')
   call nhlfrlclear(rlist)
   call nhlfrlsetstring(rlist,'appDefaultParent','True',ierr)
   call nhlfrlsetstring(rlist,'appUsrDir','./',ierr)
   call nhlfcreate(appid,'tx01',nhlfappclass,0,rlist,ierr)
   ```

   This section of code
   - creates a resource list,
   - clears the list,

- adds NhlNappUsrDir resource of the App object to the resource list with a setstring call to the current working directory (./), then
- creates an application object.

It returns the id tag appid. The name of the resource file is tx01 with .res appended: tx01.res. This App object is of type NhlappClass. Its resources are initialized using the values in the resource list rlist.

When we run this program, it will expect to find a resource file named tx01.res in the current working directory.

5. Create an XWorkstation object.

   Fortran:

```
      call nhlfrlclear(rlist)
      call nhlfrlsetstring(rlist,'wkPause','True',ierr)
      call nhlfcreate(wid,'tx01Work',nhlfxworkstationclass,
     $     0,rlist,ierr)
```

   This section of code
   - clears the resource list,
   - puts the resource wkPause in the resource list with a value of True, then
   - creates an XWorkstation object.

   The id is returned in wid. This is an object of type NhlxWorkstationClass. Its resources are defined by initialized by the values in rlist. The class contains defaults for the values that are not defined by rlist or the resource file.

6. Set any resources in rlist (optional), and create a TextItem object.

   Fortran:

```
      call nhlfrlclear(rlist)
      call nhlfrlsetfloat(rlist,'txPosXF',.5,ierr)
      call nhlfrlsetfloat(rlist,'txPosYF',.5,ierr)
      call nhlfrlsetfloat(rlist,'txJust','CenterCenter',ierr)
      call nhlfcreate(pid,'TextItem1',nhlftextitemclass,
     $     wid,rlist,ierr)
```

   This section of code
   - clears the resource list,
   - puts the resources that will define the position of the TextItem in the resource list, then
   - creates the TextItem object.

   The id is returned in pid, it is an object of type NhltextItemClass, it has the parent wid, and its resources are set by the values in rlist.

   Note: The resources could also be set in the resource file tx01.res.

7. Output the graphics to the X Window.

   Fortran:

```
call nhlfdraw(pid,ierr)           or call nhlfdraw(wid,ierr)
call nhlfframe(wid,ierr)
```

The draw function causes the TextItem object, id = pid, to be drawn. The frame function flushes the output device. Since we set the NhlNwkPause XWorkstation resource to True, the plot will be drawn and held in the window awaiting a click in the window.

8. Destroy all objects which are no longer needed (optional).

   If you intend to continue the program and create some new and different App, Plot, or Workstation objects, and if you no longer need any of the current objects, then good programming practice is to delete the objects that are no longer required in order to keep the memory size of your program down. If you are about to end the program, go on to Step 9.

   Fortran:

   ```
   call nhlfdestroy(pid,ierr)
   call nhlfdestroy(wid,ierr)
   call nhlfdestroy(appid,ierr)
   ```

9. Close the HLU library and exit.

   Fortran:

   ```
   call nhlfclose
   stop
   end
   ```

   This section of code destroys all existing objects, closes the HLU library, and exits our program.

Building a C program;

1. Select the needed objects.

   First, let’s set our resources in a local resource file which is specific to this application. Thus we will use the combination of NhlInitialize and an NhlCreate of an App object. We could also have chosen to use NhlOpen and set the resources in the file whose pathname is given by the environment variable $(NCARG_USRRESFILE). Or we could choose to set resources directly in the program using resource list arguments in Create calls or by using the NhlSetValues functions.

   We do not need a data object since TextItem does not import data. The text string to be written is merely entered as a resource.

   Next, we need a workstation object in which to draw or write our graphical instructions. Suppose we just choose an X Window, which is an XWorkstation object.

   Finally, we need a View object, namely TextItem.

2. Select the needed header files.

Now that we know which objects we need, we can define the header files that are needed; these include:

C:

```
#include <ncarg/hlu/App.h>
#include <ncarg/hlu/TextItem.h>
#include <ncarg/hlu/XWorkstation.h>
```

App, XWorkstation, and TextItem are the objects we are going to use in this program.

C headers lists the headers for all objects.

3. Open and initialize the HLU library.

   C:

   ```
           NhlInitialize();
   ```

4. Create an application context. Set the app dir to the current directory so the application looks for a resource file in the working directory.

   C:

   ```
           rlist = NhlRLCreate(NhlSETRL);
           NhlRLClear(rlist);
           NhlRLSetString(rlist,NhlNappDefaultParent,"True");
           NhlRLSetString(rlist,NhlNappUsrDir,"./");
           NhlCreate(&appid,"tx01",NhlappClass,NhlDEFAULT_APP,rlist);
   ```

   This section of code
   - creates a resource list,
   - clears the list,
   - adds the NhlNappUsrDir resource of the App object to the resource list with a SetString call to the current working directory (./), then
   - creates an application object.

   It returns the id in appid. The name of the resource file is tx01 with a .res appended: tx01.res. This App object is of type NhlappClass. Its resources are initialized by the resource names and values contained in rlist.

   When we run this program, it will expect to find a resource file named tx01.res in the working directory.

5. Create an XWorkstation object.

   C:

   ```
           NhlRLClear(rlist);
           NhlRLSetInteger(rlist,NhlNwkPause,True);
           NhlCreate(&wid,"txxwork",NhlxWorkstationClass,NhlDEFAULT.APP,rlist);
   ```

   This section of code

- clears the resource list,
- puts the resource NhlNwkPause in the resource list with a value of True, then
- creates an XWorkstation object.

The id is returned in wid. This is an object of type NhlxWorkstationClass. Its resources are initialized by rlist.

6. Set any resources in rlist (optional) and create a TextItem object.

   C:

```
NhlRLClear(rlist);
NhlRLSetFloat(rlist,NhlNtxPosXF,.2);
NhlRLSetFloat(rlist,NhlNtxPosYF,.8);
NhlRLSetInteger(rlist,NhlNtxJust,NhlCENTERCENTER);
NhlCreate(&pid,"TextItem1", NhltextItemClass,wid,rlist);
```

   This section of code
   - clears the resource list,
   - puts the resources that will define the position of the TextItem in the resource list, then
   - creates the TextItem object.

   The id is returned in pid, it is an object of type NhltextItemClass, it has the parent wid, and its resources are set by the values in rlist.

   Note: The resources could also be set in the resource file tx01.res.

7. Output the graphics to the X Window.

   C:

```
NhlDraw(pid);
NhlFrame(wid);
```

   The draw function causes the TextItem object, id = pid, to be drawn. The frame function flushes the output device. Since we set the NhlNwkPause XWorkstation resource to True, the plot will be drawn and held in the window awaiting a click in the window.

8. Destroy all objects that are no longer needed (optional).

   If you intend to continue the program and create some new and different App, Plot, or Workstation objects, and if you no longer need any of the current objects, then good programming practice is to delete the objects that are no longer required in order to keep the memory size of your program down. If you are about to end the program, go on to Step 9.

   C:

```
NhlDestroy(pid);
NhlDestroy(wid);
NhlDestroy(appid);
```

   Or you could just destroy the App object since it is the ancestor of the other objects; this will cause all of its children to be destroyed as well. (This is also why the destroy calls above must be in the order in which they appear.)

C:

```
NhlDestroy(appid);
```

9. Close the HLU library and exit.

C:

```
NhlClose();
exit(0);
```

This section of code destroys all existing objects, closes the HLU library, and exits our program.

The TextItem example code

The complete programs for the examples presented above can be loaded into your current directory by issuing the commands:

```
ng4ex tx01c
```

for the C version, or

```
ng4ex tx01f
```

for the Fortran version.

The TextItem resource file

In the TextItem example 01, we set resources for our objects directly in the code. We can also set resource values in the resource file tx01.res. Remember that if you set a resource in both places, the value set in the code takes precedence.

Either of the ng4ex commands above will also load tx01.res into your working directory.

An XyPlot example

Now consider a View object that needs a data object. xy01f.f is a Fortran program that generates an XyPlot using default values for all resources. The list of XyPlot resources are commented in the associated resource file xy01.res. An equivalent C version of this example is xy01c.c. You can view any of these files by typing:

```
ng4ex xy01c
```

for the C version, or

```
ng4ex xy01f
```

for the Fortran version.

More examples

Now to see what happens as you change resource values for different objects, move on to the Quick Start Guide examples. There are examples for each View object, each Workstation object, and each data object. If you already have experience using NCAR Graphics, you might want to immediately jump to the example set available through the ng4ex command. ng4ex is also documented under the standard UNIX man facility. ng4ex has many options for listing, copying, or executing C, Fortran, and NCL examples.

See also:

- ng4ex command
- The HLU Application Programming Interface (API)
- HLU functions
- Writing an HLU program
- HLU header files
- How objects are used in an HLU program

Mixing HLUs and LLUs

Some of the utilities of Version 3.2 have not yet been converted to HLUs (missing are histograms, 3-D wireframe surfaces and isosurfaces, weather maps, and polypack). Moreover, the wide range of flexibility available at the low level can never be fully implemented in a high level interface.

Thus, there may be situations for some time in which you will want to directly call routines in the low level libraries. Consequently, it is important to learn how to mix LLU calls with HLUs in an HLU program.

Some major considerations are:

- In a C program, you need to include the GKS library and the LLU C-binding library.

  ```
  #include <ncarg/gks.h>
  #include <ncarg/ncargC.h>
  ```

  These libraries are automatically loaded for Fortran programs.

- The GKS output connection for low level output and HLU output must be the same.

  To accomplish this, you must retrieve the workstation id used by the HLU program and assign it to the LLU workstation. In Fortran, the calls would look like this:

  ```
  C
  C Activate the GKS workstation using the HLU workstation id, gkswid.
  C
        call NhlFRLClear(grlist)
  ```

```
      call NhlFRLGetinteger(grlist,'wkGksWorkId',gkswid,ierr)
      call NhlFGetValues(wid,grlist,ierr)
      call gacwk(gkswid)
```

Then issue the LLU calls you want to make. Then deactivate the GKS workstation with the call:

```
      call gdawk(gkswid)
```

In a C program, these calls would look like:

```
/*
 * Activate the GKS workstation using the HLU workstation id, gkswid.
 */
    NhlRLClear(grlist);
    NhlRLGetInteger(grlist,NhlNwkGksWorkId,&gkswid);
    NhlGetValues(wid,grlist);
    gactivate_ws(gkswid);

/*
 *  Add your LLU codes here.
 */

    gdeactivate_ws(gkswid);
```

- The viewport and transformations must be synchronized.

  In example cn11f.f (which you can copy into your local directory and execute using ng4ex), a call is made to the LLU contouring utility Conpack. The viewport and transformation are set with a call to the LLU set routine. The code is:

```
C
C  The following are calls to the low level ncar graphics library to
C  draw labeled contour lines
C
      call gacwk(gkswid)
      call cpseti('set',0)
      call cpsetr('xc1',-90.0)
      call cpsetr('xcm',90.0)
      call cpsetr('yc1',0.0)
      call cpsetr('ycn',9.0)
      call cpsetr('cis',5.0)
      call cpsetr('dps',.02)
      call cpseti('lis',2)
C
C  Note that y axis user coordinates are set to grid coordinates
C  rather than data coordinates. this is done because the input
C  data is irregularly spaced in the y direction. This 'tricks'
C  Conpack into thinking the grid points are evenly spaced.
C
      call set(0.2,0.8,0.2,0.8,-90.0,90.0,0.0,9.0,1)
      call cprect(t,73,73,10,rwrk,5000,iwrk,1000)
      call cpcldr(t,rwrk,iwrk)
      call gdawk(gkswid)
C
C  End of NCAR Graphics section. Note deactivation of gks workstation
C
```

You can retrieve and execute the equivalent C example by issuing the command `ng4ex cn11c`.

- Watch out for automatic frame advances in the LLUs.

  Some LLU codes automatically advance the frame before returning. In such cases, you will want to issue an HLU draw before the LLU call.

See also:

- HLUs vs LLUs
- NCAR Graphics Low Level Utilities Fundamentals
- NCAR Graphics Contouring and Mapping Tutorial

HLUs: Putting it all together

Here are some reminders to help you get started using HLUs effectively:

- Make sure that the NCAR Graphics software is properly installed by running an ng4ex example in a working directory.

- Create a new application by copying and renaming an ng4ex example and its resource file (ends with .res) that produces a graphic similar to what you need.

- Edit the example source code so it reads your dataset, creates the output you want, and sends the output to the type of workstation you want.

- Edit your assignment paths for resources in the .res file. Enter the values you want for the resources in the .res file.

- Compile and execute your program. Look at your graphic. If it is output to an X Window, it will appear on your monitor. If you wrote an NCGM file, use ctrans, ictrans, or idt to view it. If you wrote a PostScript file, you can use a PostScript previewer such as ghostscript to view it, or you can send it to a PostScript printer.

- If you detect problems with the image, select new values for the appropriate resources in a resource file and re-execute. If there are problems with the graphical objects, return to and edit your HLU program, as needed.

See also:

- Strategy for learning HLUs
- Writing an HLU program
- HLU plot creation sequence
- Understanding classes and objects
- Understanding resources
- ng4ex command

Troubleshooting HLUs

This module describes some of the common problems users encounter when they compile and run HLU applications.

The HLU APIs were designed so that a descriptive error message is generated when a problem is encountered. The Error class can be used to trap and print these messages. However, there are some cases where errors can occur which go unreported. This module will point out such cases as we discover them.

Also, note that the form of the compiler error messages listed in this module are system- and compiler-dependent.

Problem 1

A value is changed in a resource file, but the resulting graphic, or other object, does not reflect the change.

Solution:

1. Check that your resource file is in the proper directory and is correctly named. Resource files describes the way user resource files are named for an application.
2. There could be a spelling error in the assignment path of that resource in the resource file.

   You may have spelled the resource name incorrectly, spelled an object id incorrectly, or formed an illegal object pathname. Errors of these kinds in a resource file are not reported. To see if the problem is of this kind, set the resource in question directly in your HLU code as shown in Three ways to set resources. Compile and run your code using nhlcc or nhlf77 and watch for an error message. If there is no error message and the plot now reflects the change in the resource value, then there is indeed a pathname error in defining that resource in your resource file. To determine if the error is in the pathname, default the resource to apply to all objects in its hierarchy using the wild card character as in:

   ```
   *resourcename : resource value
   ```

3. Check whether you set this resource in your HLU program. Resource setting in the program overrides resource setting in a resource file. If this is the problem, you will need to either pull the code that sets the resource out of your program, or set the new value there. Either case will require that you re-compile, link, load, and execute to create a new program module.

Problem 2.

You get error messages of the form:

```
cfe: Error: xy04c.c, line 109: 'NhlNxyCoordData' undefined; reoccurrences will r
     NhlRLSetInteger(rlist,NhlNxyCoordData,dataid);
 --------------------------^
cfe: Error: xy04c.c, line 110: 'NhlxyPlotClass' undefined; reoccurrences will no
     NhlCreate(&plotid,"xyPlot",NhlxyPlotClass,xworkid,rlist);
```

```
--------------------------------^
```

Solution:

Check the include statements in your C program. In this case, the include statement

```
#include <ncarg/hlu/XyPlot.h>
```

was omitted. This is the header file needed by the XyPlot object.

Problem 2a.

You get error messages of the form:

```
fatal:nhlfcreate:Invalid Class Function
fatal:NhlDraw: Invalid plot ID: -4
```

Solution 2a.

You omitted the statement

```
external nhlfxyplotclass
```

from your Fortran program.

Likewise, if you omit

```
external nhlfxworkstationclass
```

you will get errors like:

```
fatal:nhlfcreate:Invalid Class Function
fatal:_NhlCreate:Invalid Parent id #-4
fatal:NhlDraw: Invalid plot ID: 0
fatal:Unable to update Workstation with PID#-4
fatal:Unable to clear Workstation with PID#-4
fatal:Unable to Destroy (Bad PID#-4)Consulting Contacts:
```

Omitting

```
external nhlfappclass
```

will result in:

```
fatal:nhlfcreate:Invalid Class Function
fatal:_NhlCreate:Unable to access "default" App object
fatal:_NhlCreate:Unable to access "default" App object
fatal:_NhlCreate:Unable to access "default" App object
fatal:NhlDraw: Invalid plot ID: 0
fatal:Unable to update Workstation with PID#0
fatal:Unable to clear Workstation with PID#0
fatal:Unable to Destroy (Bad PID#0)
fatal:Unable to Destroy (Bad PID#-4)
```

Omitting

```
external nhlfcoordarraysrclass
```

will produce:

```
fatal:nhlfcreate:Invalid Class Function
fatal:Unable to access plot with pid:-4
warning:Unable to add DataItem "(null)" to DataList "xyCoordData"
fatal:CompileDataList:DataList has no valid members
```

You will get a plot with a grid and no data.

The NCAR Graphics consulting staff is available via e-mail at graphuse@ncar.ucar.edu. Installation consulting is available at ncargfx@ncar.ucar.edu. The comment form is available for submittal from any page of the documentation. Just click on the Feedback icon in the control panel and go from there.

See also:

- Strategy for learning HLUs
- Writing an HLU program
- HLU plot creation sequence
- Understanding classes and objects
- Understanding resources
- ng4ex command

NCAR Command Language (NCL) overview

The NCAR Command Language (NCL) is an interactive, text-based language interpreter that helps you manipulate data and create NCAR Graphics images. NCL allows you to import and export data in a variety of formats (ASCII, binary, HDF, netCDF) and to interactively manipulate that data.

Interactive data manipulation includes selecting region-of-interest data, thinning data, interpolating irregular grids to regular grids, removing corrupted data, and performing useful transformations. You can also customize the methods NCL uses to ingest and manipulate data.

As a tool for specifying visualizations, NCL allows you to generate 1D and 2D NCAR Graphics plots in many styles with many options for annotating, labeling, and animating the visualizations.

Here are the primary modules describing NCL:

- Strategy for learning to use NCL - how to learn to use NCL.
- Getting started using NCL - a practical introduction to NCL, for both beginning and advanced users
- Detailed NCL overview - description of NCL and explanations about NCL functionality.
- When to use NCL - information about when to use NCL, HLUs, and LLUs.
- NCL user environment - how to set up your environment and run NCL.
- Data model - how data is modeled in NCL and how variables, coordinates, data attributes,

dimensions, and array subscripts are used.

- NCL syntax and semantics - details on NCL syntax and semantics.
- Expressions and operators - how to use NCL expressions and operators. Also, tables that describe coercion behavior, list the operators, and present the resulting types when types are mixed in an expression.
- NCL commands, functions, and procedures - the NCL commands, functions, procedures and information about extending the function set.
- Mixing NCL with HLUs and LLUs - how to access HLU objects and the LLUs from NCL.
- NCL plot creation sequence - an overview of the steps used for creating a plot.
- Importing and exporting data - how to get data into NCL and how to save the data after it has been processed.
- Data manipulation - examples of how to process data.
- Visualization specification process - how to specify the resources and characteristics of an NCAR Graphics plot and create overlays.
- Visualizing images - How to create output workstations and send plots to them.
- Manipulating images - how to position images on an output workstation and how to alter and redraw objects.
- Batch mode instructions - how to use NCL in batch mode and create scripts.
- NCL API - description of the NCL programming interface and the NCL programming library.
- Putting it all together - examples and plots that demonstrate the topics discussed earlier.
- Troubleshooting NCL - how to troubleshoot problems you may encounter with NCL.

Detailed NCL overview

The NCAR Command Language (NCL) supports the needs of users who want to conduct interactive data exploration and analysis. NCL can be thought of as a complete programming language. It has types, variables, operators, expressions, conditional statements, and loops. NCL also has features that are not found in common programming languages. These additional features handle manipulation of metadata and the configuration of the output graphics. NCL is an open environment allowing users to extend the NCL function set by developing their own functions and procedures. NCL is not meant to be a replacement for programming in other structured languages, but it is meant to provide an integrated environment where data can be selected, manipulated and visualized interactively without requiring compilation.

Easy and intuitive access to datasets is a fundamental prerequisite for interactive data exploration, analysis, and visualization. Since datasets often come in a variety of data formats, grid sizes, grid resolution, and units, very different datasets often need to be combined, compared, and used at the same time. NCL allows different datasets to be imported into one uniform and consistent data manipulation environment. NCL data management is patterned after the NetCDF data model. See the NCL data model module for detailed information about how data is managed in NCL.

Easy and intuitive output specification is also a prerequisite for interactive data analysis and exploration. NCL has built-in defaults for quick and accurate visualization of data. The defaults can also be customized by the user in the form of a user defaults file. Furthermore, visualization specifications can be entered and altered at the command line.

NCL can operate in two modes. The first mode of operation is as an interactive command line interpreter where every statement is executed immediately after the user enters a command or expression. The second mode of operation is as a batch command interpreter where an entire NCL script is read in at one time and executed.

NCL uses the following model for interactive data analysis: data is selected and read from a file, data is processed, a visualization specification is made, then the visualization is rendered.

NCL user environment

This module describes the path names and environment variables that users must set to use the NCAR Command Language, and it describes how to run NCL.

Environment variables

Before you can use NCL, you must make sure that you have set either NCARG_ROOT or the three environment variables NCARG_BIN, NCARG_LIB, and NCARG_INCLUDE, depending on how NCAR Graphics was installed on your system. In some cases, the installer may have set either NCARG_ROOT or NCARG_BIN, NCARG_LIB, and NCARG_INCLUDE globally so that you don't have to worry about it. To figure out if any of these variables have been set, execute:

```
env
```

A list of all the environment variables that have been set will be shown, along with their values.

If you don't have either NCARG_ROOT, or NCARG_BIN, NCARG_LIB, and NCARG_INCLUDE set, then please contact your system administrator.

Path names

You must have the NCAR Graphics binaries in your execution path. If you have the NCARG_ROOT environment variable set, then the location of these files is $NCARG_ROOT/bin, otherwise the location is $NCARG_BIN.

You must add one of these directories to your execution path in order to use the NCAR Graphics software. Contact your system administrator if you need help in adding a directory to your execution path.

How to run NCL

After you have set up your environment variables and execution path so that you can access the NCAR Graphics software, you can invoke NCL by typing "ncl". A command line interpreter will start up and the following prompt will appear on your screen:

```
ncl 0>
```

You can now begin typing your NCL commands.

Alternatively, you can start NCL and load a script to run in batch mode. A script is an ASCII file that contains NCL commands and comments. To run a script when you invoke NCL, type:

```
ncl < scriptname
```

where *scriptname* is the name of the NCL script.

You can also load and run an NCL script from within NCL. After you invoke NCL and get a prompt, type:

```
load "scriptname"
```

scriptname will be loaded and executed. After the script finishes, the NCL prompt will return, and you can continue typing NCL commands or load another script.

How to exit NCL

Type

```
quit
```

at the NCL command line prompt to exit NCL.

When to use NCL rather than HLUs and/or LLUs

Introduction

This module discusses the benefits and limitations of using NCL and when it may be more appropriate to use high level and low level utilities for creating an NCAR Graphics application. Some of the strengths of NCL include its array manipulation capabilities and data I/O functions. However, if users are very familiar with the traditional NCAR Graphics interface, they may want to continue using the low level utilities until more functionality is added to the HLUs and the NCL.

User experience

New users are encouraged to learn and use the NCAR Command Language for creating contours, maps, and graphs. NCL has a simple grammar and uses most of the standard and familiar programming constructs found in common computer and scripting languages. Veteran users of the NCAR Graphics software, however, may want to continue to use the traditional C and Fortran interfaces rather than learn a new system of creating plots. As the software evolves and more functionality is added to the NCL and the HLUs, then traditional users will want to migrate to the new interfaces.

Data I/O

One of the strengths of NCL is its data I/O capabilities. NCL can be used to read several different formats of data using only a handful of NCL statements. On the other hand, if users want to import data using the HLU or LLU interface, then they will need to learn how to use the interface to the libraries for each data format they want to read and write.

So, for example, to read a NetCDF data file using NCL, users only need to use the NCL addfile statement to import netCDF data. On the other hand, a C or Fortran application that reads NetCDF data must contain the appropriate include files, link and load options, and correct sequence of statements to open and read the file.

Type conversions

Another advantage of using NCL over other interfaces is that NCL automatically performs resource type conversion so that it is not necessary, as with the HLUs, to use a different function to set resources depending on the resource type.

So, for example, to set the text font height to a float value and the text font color to an integer index value using the HLU C interface, you would use two different HLU commands.

```
NhlRLSetFloat(rlist,NhlNtxFontHeightF,0.01);
NhlRLSetInteger(rlist,NhlNtxFontColor,18);
```

With NCL, however, there is no need to use a separate function for setting float, integer, or any other type of resources. In NCL, all resource assignments can be performed with a single, consistent setvalues statement.

```
setvalues plot_reference
    "txFontHeightF" : 0.01
    "txFontColor" : 18
end setvalues
```

Array operations

NCL supports array operations that are not available in C and Fortran without special libraries or user functions. Users can perform many single-statement array operations in NCL without having to write special matrix functions and procedures that support full array manipulation.

For example, if AR1 and AR2 are two-dimensional arrays, then multiplication of these two matrices is accomplished with a single statement.

```
AR3 = AR1 * AR2
```

NCL also supports a wide range of array selection methods. Users can extract subsets of data, specify strides, and select data ranges using special array subscripts. Although similar functionality is available in FORTRAN90, this functionality is not available in C.

Compilation considerations

Since NCL is an interpreted language, it is not necessary to compile an NCL script in order to execute a program. Script changes can be made quickly with a program editor and tested without having to recompile the NCL program.

Functionality

NCL functionality mirrors what is available with the HLUs. However, the current release of the software does not support isosurfaces, field flows, cell arrays, histograms, 3-D plots, and weather maps. Users will have to use the low level utilities to gain access to this functionality at this time. In future releases of NCAR Graphics, however, the HLUs and NCL will contain the bulk of NCAR Graphics functionality.

Mixing NCL with HLUs and LLUs

Introduction

This module describes how to produce plots with code that uses NCL and calls HLUs and LLUs.

Accessing HLUs

NCL has three statements for interfacing with the High Level Utilities library:

create allows you to create an HLU object and assign vlaues to its resources.
setvalues allows you to modify resources in an HLU object after it has been created.
getvalues allows you to retrieve the values of the resource setting in an HLU object.

As well as the three NCL statements above, NCL has versions of most of the functions available to the HLUs. See NCL versions of HLU functions and procedures in the NCL commands module.

Almost all of the examples available via the ng4ex command illustrate accessing HLUs from NCL. Type

```
man ng4ex
```

for details on ng4ex.

Accessing LLUs

There is no way to access the Low Level Utilities library from an NCL program without implementing the desired functions by extending the NCL function set.

Strategy for learning to use NCL

Introduction

This module describes basic strategies for learning NCL. Since different people have different optimal learning styles, several techniques are presented. You may want to mix and match to suit your own style.

NCL scientific tutorial

If you are a person who prefers to jump right in to a new language, the best starting point may be the NCL scientific tutorial. If, on the other hand, you prefer to read more about the underlying principles of the language, then you may prefer to start by reading the User Guide.

User Guide

Read through the modules listed in the NCL Command Language module. As you read various modules, you may want to run NCL interactively to test various features; you can do this by using the NCL scientific tutorial concurrently with the User Guide. See the NCL user environment module for information about how to set up your environment and run NCL.

Use the "Roadmap" button on the control panel to get an idea of where NCL fits into the big picture. Use the "Contents" button on the control panel to see a listing of the contents of the entire User Guide, and where NCL fits in.

Reference Manual

For a more structured approach, and for details, you need to study the NCL Reference Manual.

Examples

No matter what approach you choose, it will be valuable for you to see examples of running NCL scripts. A starting point is to look at the NCL codes that are contained in the NCL scientific tutorial. The tutorial does not cover all of NCL's functionality, so you may also want to run some of the many NCL examples contained in the Quick Start Guide and available via the ng4ex command. These examples vary from simple Basic examples to more complex overlay and animation examples.

Quick Start Guide examples begin at the Pick a utility module. The examples available via the ncargex command can be accessed with the following useful calls:

```
man ng4ex
```

Gives details of how to use the ng4ex command.

```
ng4ex -list -A
```

Lists the names of all available *ng4ex* examples.

```
ng4ex -list -NCL
```

Lists the names of all available ng4ex NCL examples.

```
ng4ex -NCL
```

Copies all ng4ex NCL scripts and associated data files into the local directory and runs them, creating output graphics.

```
ng4ex -n ex00n
```

Copies the NCL script and associated data files (if any) into the local directory, but does not run it. *ex00n* must be the name of a legal ng4ex example.

```
ng4ex ex00n
```

Copies the NCL script and associated data files (if any) into the local directory and runs it. *ex00n* must be the name of a legal ng4ex example.

Data files and HLUs

The two primary uses of NCL are for interactive manipulation of data files and interfacing with the HLUs. The most common format for data file input and output is netCDF. If you are going to work with netCDF datasets, you can learn about it from the UNIDATA netCDF pages. If you want to produce graphics via the NCAR Graphics HLUs, you will want to look at the modules listed in the High Level Utilities overview.

Data model

This module introduces the NCL data model and explains the concepts that are needed to understand how data is managed and manipulated using NCL.

The NCL data model is patterned after the data model used in netCDF (network Common Data Form) files from Unidata. NetCDF data structures are stored in self-describing, machine-independent files. The data structures contain variables, coordinates, dimensions, and metadata information that describe data and file characteristics such as units of measure, or other descriptive text. NCL shares these concepts with netCDF. In particular, the NCAR Command Language supports the use of metadata information for managing data. For example, users can read or assign descriptive text about a variable such as variable names, units, valid ranges, data minimums and maximums, missing values, and coordinate indices.

NCL manages your data using the concept of a file variable. A file variable is an internal data structure that references and describes file variables and provides information about dimensions, coordinates, and attributes for each variable in the file in addition to the file itself. A file variable can hold up to 512 variables and 32 dimensions, in addition to an unlimited number of attributes of many primitive data types. Variables, attributes, and coordinate variables can be one of the following primitive data types: float, double, short, long, integer, character, or byte.

Although the NCL data model is patterned after netCDF, the netCDF format is not the only file format that will be supported by NCL. NCL is designed to allow other formats similar to netCDF to be added to the NCL environment.

In addition to supporting self-describing formats like netCDF, NCL provides functions and procedures for reading and writing ascii and binary files. However, attributes and other metadata information cannot be read from or written to ascii and binary files. This information must be added manually using NCL statements.

NCL data is described and managed using named dimensions, named variables, coordinate variables, and attributes.

Dimensions

NCL dimensions define the shape and size of a variable. A variable or data file can have up to 32 dimensions, and they can be any size. These dimensions may also have names.

Variables

NCL variables are symbolic representations of multidimensional arrays. Scalar variables are represented as a single dimension array of size 1. Variables can have any of the following types: float, double, short, long, integer, byte, character, or string.

Variable names are case sensitive and must begin with a letter or an underscore '_'. Any combination of characters and numbers can make up the rest of the name. The maximum length of a variable name is 256 characters.

Coordinate variables

NCL Coordinate variables are used to represent data coordinates. Each dimension of a variable can have a one-dimensional array of coordinate points that define the coordinates of the data along a single dimension.

Array subscripting

Array subscripting or indexing is used to select subsets or specific elements of a data array. Data is ordered in a row x column fashion, and NCL supports three different methods of array subscripting so that users can access and manipulate their data in a way that is the most meaningful and intuitive to them. Standard subscripting is similar to array indexing used in FORTRAN 90. Coordinate subscripting uses coordinate variable data to select array subsets. Named subscripting uses the names of the dimensions to allow for array dimension reordering operations.

Attributes

NCL attributes are the metadata that describe a variable or data file and its contents. The attributes can specify data range, variable sizes, a file title, a file history, and any other user-defined descriptive information. Attributes can have any valid NCL type. Common variable attributes are the valid range of the variable, the units, the long name of the variable and other user-defined information. Common data file attributes are the file title and date. Variables and files can have as many attributes as the user desires.

Data types and coercion

Extensive information about NCL data types and coercion between different types appears in the Data Types Overview module in the NCAR Graphics Reference Manual.

NCL variable dimensions

Dimensions define the shape and size of a variable. A variable can have up to 32 dimensions, and they can be any size. Variable dimensions can also have names.

The variable dimension names are referenced using the "!" character following the variable name, and variable dimension numbers start at 0. So, for example, the first dimension name of a variable called t is referenced in NCL as t!0, the second dimension name as t!1, and the third dimension name as t!2. The ordering of the dimensions of a variable reflect the ordering of the data in row/column order.

For example, if we create a variable, t, with three dimensions of size 31, 40, and 50, that contains temperature data for a 31 day period, then the first dimension might be named "day", the second dimension "lat", and the third dimension "lon".

```
t = new(/31, 40, 50/), float)

t!0 = "day"

t!1 = "lat"

t!2 = "lon"
```

The dimension sizes of a variable are referenced using the dimsizes function as shown in the following example. The dimension sizes are stored in the variable n, and then printed with the print statement.

```
n = dimsizes (t)

print(n)
```

Data files can store named dimensions that are used by file variables. A file’s dimensions are the dimension names and sizes that are stored in the file and used by data variables in the file. For example, a data file called file1 could have four dimensions named lat, lon, day, and year, and two variables called t and p. The t and p variables use the file’s dimensions to define their shapes and sizes.

The file dimension names are referenced using the "!" symbol: file1!0, file1!1, file1!2, and file1!3. The order of a file’s dimensions does not reflect any type of data ordering. The order only represents the order in which the dimensions were defined.

The following NCL example would read a netCDF file called file1.cdf and print out the first two dimension names defined in the file.

```
file1 = addfile ("file1.cdf", "r")

dim1=file1!0

dim2=file1!1

print(dim1)

print(dim2)
```

The size of the dimensions, as well as lists of all the variables, dimensions, coordinates, and attributes in a file, can be seen by printing the file.

```
print (file1)
```

NCL variables

NCL variables are symbolic representations of multidimensional arrays. Scalar variables are represented as a single dimension array of size 1. Variables can have any of the following types: float, double, short, long, integer, byte, character, or string.

Variable names are case sensitive and must begin with a letter or an underscore ’_’. Any combination of characters and numbers can make up the rest of the name. The maximum length of a variable name is 256 characters.

File variables vs. variables in memory

There are two general types of variables in NCL: file variables and variables in memory. File variables are variables that are read from an external file. These variables are referenced in NCL using a file record. Variables in memory are variables that have been created through assignment statements or the "new" statement.

File variables are stored in an external file and are referenced using a file record. File records contain pointers to file variables in an external file and metadata information, such as coordinate variables, dimension definitions, and attributes. Attributes, coordinate variables, and dimension names, when assigned to a file variable, are automatically written into the file when write permissions are available.

In-memory variables, on the other hand, are variables that are dynamically created using the "new" operation and assignment statements; they are not read from an external file. Attributes, coordinate variables, dimension names, etc. are assigned using NCL statements.

File variables are operated on and subscripted in the same way as in-memory variables using the '!', '&', and '@' operators. Dimensions and coordinate variables are not unique to individual file variables; they are shared by all variables in the file with the same dimension names. This means that adding or modifying dimensions and coordinate variables for one file variable may change the dimensions and coordinates for all the variables in the file.

For example, if memvar is a two-dimensional float in-memory variable, and filevar is a file record that references an external file that contains a file variable called precip, then you could reference the dimensions in the following fashion:

```
; Assign a name to the first dimension of the memvar variable.
memvar!0 = "latitude"

; Assign a name to the first dimension defined in the file
; referenced by filevar.
filevar!0 = "latitude"
```

To reference a variable that is stored in an external file, the '->' operator must be used.

```
; Assign a name to the first dimension of the precip variable
; that is stored in the external file.
filevar->precip!0 = "latitude"
```

Note that when file variables are changed, the external file is automatically and immediately updated if the file was opened with write permission.

Creating variables

In-memory variables are created through variable assignment on the command line or by using the "new" statement. For example, the following NCL variable assignment would create an integer variable in memory called var_1 with a value of 5.

```
var_1 = 5
```

A float variable is created by adding a decimal point to the value, and string variables are created by enclosing the value in quotes.

```
var_2 = 5.894

var_3 = "This is a string variable"
```

Logical variables are created by using the keywords "True" and "False".

```
var_4 = True
```

To create variables of type double, long, short, byte, and character, you can use the intrinsic operation called "new". The "new" operation takes two required arguments to specify the dimensions and type, and one optional argument to specify the missing or fill value in a variable.

```
new ( dimension_array, type, optional_missing_value )
```

dimension_array is an array argument of dimension sizes such as:

> (/30, 50, 365/) - a three-dimensional variable with dimension sizes of 30, 50, 365.
> (1) - a scalar variable.
> (20) - a one-dimensional variable with a single dimension of size 20.
> (/100, 200/) a two-dimensional variable with dimension sizes of 100 and 200.

type is the variable type. It may be double, float, long, integer, short, byte, string, or character.

optional_missing_value is the fill value that each element is assigned to in your variable array.

The following NCL commands create variables of various dimensions and types.

```
;var1 is a three-dimensional float variable of size 22x33x44.
var1 = new((/22, 33, 44/), float)

;var2 is a scalar byte variable.
var2 = new ((1), byte)

;var3 is a single dimensioned integer variable of size 100.
;The elements in this array will all be pre-assigned the value of -999.
var3= new ((100), integer, -999)
```

Deleting variables

The delete function is used to delete in-memory variables from NCL. The function accepts one argument: the variable that you want to delete. For example, if you have a variable called var1 of any dimension and type, the following NCL statement will delete all of the memory associated with the variable and its reference in the symbol table.

```
delete(var1)
```

File variables are a special type of variable and can not be deleted using the delete procedure. This is primarily because the netCDF and HDF formats do not allow the deletion of variable in a file. One way to accomplish this is to create another file and only assign the variable you would like to it leaving out the ones you wish to delete.

Coordinate variables

Coordinate variables are used to represent data coordinates. Each dimension of a variable can have a one-dimensional array of coordinate points that define the coordinates of the data along a single dimension.

So, for example, if you have a two-dimensional temperature variable that contains data that is mapped over an area defined in terms of latitude and longitude, two coordinate variables (one-dimensional arrays) would define the latitude and longitude points of the data. One coordinate variable would define all the latitude points of the temperature data, and one coordinate variable would define all the longitude points. The sizes and names of these variables would be equal to the sizes of their corresponding dimensions.

The values in a coordinate variable array must be monotonically increasing or decreasing. For example, the following are valid coordinate arrays:

> -3, -1, 4, 10, 15
> 200.4, 33.2, 0., -100.2, -311.0, -500.0

The following are not valid coordinate arrays because they are not monotonically increasing or decreasing:

> -10, 5, -8, 9, -6, 13
> 15, 14, 13, 12, 13, 14, 15

The ’&’ operator is used to reference and assign coordinate variables. If the variable called temp has dimension names lat and lon, then its coordinate variables are referenced with the same names, i.e. temp&lat, temp&lon.

For example, the following NCL statements assign two coordinate variables to a temperature variable.

```
; create the temperature variable ( a 5x4 array of data)
temp = new((/5,4/), float)

; assign dimension names
temp!0 = "lon"
temp!1 = "lat"

; initialize the values in the coordinate arrays
lon_coordinates = (/-90, -85, -80, -75, -70/)
lat_coordinates = (/0, 20, 40, 60/)

; assign the coordinate arrays to the variable
; Note that the coordinate variables must have the same name and
; size as their corresponding dimension.
```

```
temp&lon = lon_coordinates
temp&lat = lat_coordinates
```

NCL array subscripting

Array subscripting or indexing is used to select subsets or specific elements of a data array. Data is ordered in a row x column fashion, and NCL supports three different methods of array subscripting so that users can access and manipulate their data in a way that is the most meaningful and intuitive to them. **Standard subscripting** is similar to array indexing used in FORTRAN 90. **Coordinate subscripting** uses coordinate variable data to select array subsets. **Named subscripting** uses the names of the dimensions to allow for array reordering operations.

Standard subscripting

Standard subscripting is used to select subsets of data from an array. The indexing is performed in a way that is very similar to the method used in FORTRAN 90. Standard subscripting allows users to thin data, select ranges, specify data strides, and reverse the order of data.

Selecting subsets

Subsets of data are selected by specifying the numerical location of the data in an array. For example, if precip is the name of a three-dimensional variable with dimensions of 5x4x8, then the first and last elements in the array are selected in the following way:

```
precip(0,0,0)
precip(4,3,7)
```

Array subscripts always begin at 0 when using standard subscripting, and they must be of type integer. Valid subscripts cannot be less than 0 nor greater than the dimension size minus one. Individual subscripts are separated by commas, and the entire subscript list is surrounded by parentheses.

A range of data is selected using the colon ':' character. The number preceding the colon represents the starting subscript, and the number after the colon represents the ending subscript. The beginning and ending subscripts are inclusive, so the following references would select a 3x1x1 and a 2x4x1 subset of the precip data array. The actual output dimensionality of these selections are 3 and 2x4 respectively. When a dimension becomes a single dimension the rank of the array is reduced by one. If the results of these selections where assigned to another array the rank of the other array would be 1 and 2 respectively.

```
precip(1:3,3,1)
precip(2:3,0:3,2)
```

If a number does not appear before the colon, then the beginning of the range is automatically selected. Similarly, if the number following the colon is omitted, then the end of the range is automatically selected. For example, the following statement selects the entire range of the first and last dimensions of the variable.

```
precip(:,0:3,:)
```

Data can be thinned by specifying a stride in the subscript so that every other, or every third, every fourth, etc. array element is selected. The optional stride value is specified by adding an additional colon after the subscript range followed by a positive integer value for the stride. The stride value is added to the current index to determine the subset of data selected in a given range. For example, a stride of 3 would select the 1st, 4th, 7th, 10th, etc. data elements in the range.

```
precip(0:4:2,0:3,0:7:3)<
```

The above statement selects a 3x4x3 array subset.

Reversing the data order

There is no restriction on having the start of a subscript range be less than the end of the range. When the start is greater then the end, a reverse selection is done, meaning the variable data order is reversed.

The following example shows how data order is reversed by printing the same variable twice, but reversing the subscripts in the second case.

```
; create a 3x4 integer variable
a=new((/3,4/), integer)

; initialize the variable
a=(/(/1,2,3,4/),(/5,6,7,8/),(/9,10,11,12/)/)
```

The statement, print(a(0:2,0:3)), prints the data in its original order.

```
(0,0)   1
(0,1)   2
(0,2)   3
(0,3)   4
(1,0)   5
(1,1)   6
(1,2)   7
(1,3)   8
(2,0)   9
(2,1)   10
(2,2)   11
(2,3)   12
```

However, reversing the subscripts reverses the order. The statement, print(a(2:0,3:0)), prints the data in opposite order.

```
(0,0)   12
(0,1)   11
(0,2)   10
(0,3)   9
(1,0)   8
(1,1)   7
(1,2)   6
(1,3)   5
(2,0)   4
(2,1)   3
(2,2)   2
(2,3)   1
```

Using vectors to select data

A vector of integer indices can also be used as a subscript. As long as all of the entries in the vector are within the bounds of the given dimension, then the vector could be any size. You can also select a single index more than once. For example, the following statement uses a vector subscript to print several values from the array defined in the previous section.

```
print (a(2,(/0,0,0,1,1,1,2,2,2,1,1,1,0,0,0/)))
(0)     9
(1)     9
(2)     9
(3)     10
(4)     10
(5)     10
(6)     11
(7)     11
(8)     11
(9)     10
(10)    10
(11)    10
(12)    9
(13)    9
(14)    9
```

Coordinate subscripting

Coordinate subscripting was developed for users who want to access their data and specify index ranges in terms of coordinate variable values. For someone who is very familiar with a particular dataset, it may be more intuitive to select subsets of data by specifying coordinate ranges rather than using standard subscripting.

For example, if a temperature variable is defined over latitude, longitude, and time, one possible naming scheme is to name the variable’s dimensions time, lat, and lon. Coordinate variables could then be assigned to these dimensions and, using coordinate subscripting, a user could select a subset of data by specifying a latitude, a longitude, and a time range, rather than using standard indexing.

Coordinate subscripting uses normal subscripting syntax of parenthesis with commas each dimension's subscript, but also uses curly braces, '{}' between each of the commas to denote the use of coordinate subscripting.

For example, suppose there is a two-dimensional variable, named temperature, with dimensions called lat and lon of size 5 and 6 respectively. Further, let's suppose that a coordinate variable called lat is assigned with values 90, 89, 88, 87, and 86. Also, suppose a coordinate variable lon is assigned with values -20, -10, 0, 10, 20, and 30. Now, coordinate subscripting can be used to make data selections from the temperature data.

The following pairs of statements all select the same data; however, the first statement in each pair uses standard subscripting, and the second uses coordinate subscripting.

```
t = temperature (0,0)
t = temperature ({90}, {20})
;
t = temperature (2:4, :3)
t = temperature ({88:86}, {:10})
```

When using coordinate indexing, it is very important that you know whether your coordinate arrays are monotonically increasing or decreasing to avoid accidentally reordering your data and causing unnecessary losses in efficiency. For example, if users incorrectly assume that the temperature coordinate variables in our example are monotonically increasing, then they might mistakenly make the following reference that reverses the order of the data in the first dimension and causes an unintentional efficiency loss due to the overhead of reordering the data.

```
; The first dimension is reversed because the coordinate array
; for the first dimension was defined to be monotonically decreasing
; ( 90 to 86). However, in the following statement, the first
; dimension subscript is referenced in an increasing order (86 to 89),
; thus reversing the order of the data.

t = temperature ({86:89}, {:10})
```

Standard and coordinate indexing can be mixed in the same statement as in the following example.

```
t = temperature (0,{20:30})
```

There is a simple rule for choosing the index when the coordinate given is not an exact coordinate value. NCL finds the coordinate value above and below the given value and chooses the closest exact value. This is only the case when one coordinate value is given. If a coordinate range is specified, then all coordinate values within the range are selected.

Named subscripting

Named subscripting is an extension of both standard and coordinate indexing. Named subscripting allows the flexibility of specifying subscripts in any order. With standard and coordinate indexing, subscripts have to be specified in the order in which a variable's dimensions are defined. However, with named subscripting, the subscripts can be specified in any sequence to reorder the data.

Named subscripting allows the user to use the dimension name to specify the dimension that the subscript indexes. The rule for braces is the same as the preceding indexing schemes. The difference is that in each subscript, the name of the dimension is listed followed by the pipe symbol '|' and the subscript range.

The following example shows the same data selection first using standard subscripting and the second and third using named subscripting. The variable, temp, is a 5x6 array with coordinate variables named lat and lon respectively. The values of the lat array are (/-90, -89, -87, -86, -85/), and the values in the lon array are (/-20, -10, 0, 10, 20, 30/).

```
; standard subscripting
     t = temp (:4, 0)

     ; named subscripts with coordinate subscripting style
     t = temp ({lat | :-85},{lon | -20})

     ; named subscripts with standard subscripting style
     t = temp (lat | :4,lon | 0)
```

A feature of named subscripting is that it provides a mechanism for reordering the dimensions of data simply by switching the order of the named dimensions. For example, data is reordered in the following example because the first and second dimension names are swapped. So, rather than the longitude dimension varying the fastest, the latitude dimension varies the fastest, and the data is printed in column major order rather than the default row major order.

```
print (temp (lon | 0:5, lat | 0:4))
```

NCL data and file attributes

NCL attributes are the metadata that describe a variable or data file and its contents. The attributes can specify data range, variable sizes, a file title, a file history, and any other user-defined descriptive information. Attributes can have any valid NCL type, but they must be single dimensioned. Common variable attributes are the valid range of the variable, the units, the long name of the variable, and other user-defined information. Common data file attributes are the file title and date. Variables and files can have as many attributes as the user desires. Attributes are limited to be single dimension values. Individual elements of an attribute can be selected using standard array subscripting following the reference to the variable and attribute.

For example, a data file, called file1, may have variables, such as temperature and precipitation, and global attributes, such as the file title, and creation date. The variables may also have attributes, such as the units or the maximum valid data value.

The at sign, '@', is used to specify a variable or a file attribute name. Attributes are created in the following manner:

```
; File attributes called "title" and "date"

file1@title = "South American Climate Data"

file1@date = "November 23, 1994"

; Variable attributes called "units" and "max"

file1->temperature@units = "Degrees C"

file1-> precipitation@units = "centimeters"

file1->temperature@max = 40.87
```

NCL syntax and semantics

NCL can be thought of as a complete programming language. As a programming language, it has a grammar and syntax rules for constructing statements and commands. This module describes the syntax, semantics, and conventions used for constructing NCL statements, identifiers, comments, array subscripts, data, metadata, and strings.

Symbol set

Any printing ACSII character is a legal character in NCL:

> uppercase letters: A B C... Z
> lowercase letters: a b c... z
> digits: 0 1 2... 9
> special characters: + = _ - ( ) * & % # ! | < > . , ; : " / { } ~ \ [ ] ^ @

Also, the three non-printing characters space, newline, and tab are legal NCL characters.

Separators

White space is used to separate tokens. The valid NCL separators are space and tab.

Identifiers

Identifiers are used to give unique names to various syntactic objects (variables, function names, and so forth) in a program. NCL identifiers must begin with an uppercase or lowercase letter or the underscore

'_' character. The rest of the identifier name can contain any mix of these characters and any digits. The maximum identifier name length is 256 characters.

Examples of valid identifiers:

_A_very_long_variable_name_
_
temperature
avg_temp_1988

Examples of invalid identifiers:

temp.june
var$8
a variable name with spaces instead of underscores
7june

Comments

The semicolon ';' character is used to produce comments in an NCL statement. Comments must begin with a semicolon, and anything after the semicolon and before the next newline character is ignored by the NCL interpreter. Comments may appear alone in an NCL statement or after an NCL command. Comments cannot appear on the same line before a statement because everything after the comment character is ignored.

For example, the following are valid NCL comments:

```
; A comment can appear alone on a line. Everything after
; the semicolon is ignored by the interpreter.
temperature = 77.6  ; Comments can also appear with a command.
```

Strings

NCL character strings can be up to 256 characters long, and they may contain any of the characters listed above in the NCL symbol set.

Strings must be enclosed in double quotes, '"'. For example, the following statements create string variables and assign character strings to them.

```
string_var_1 = "This is the contents of the string variable"
string_var_2 = "Special chars (@#$%^&*_+~`;') are valid"
```

The maximum length of a string is 256 characters.

Keywords

NCL keywords are reserved words that have specific meaning to the NCL interpreter. Consequently, keywords cannot be used in other contexts. For a complete listing of the NCL keywords, see the NCL keyword reference page.

Operators

Operators are the special characters, such as the ‘+’,‘*’, ‘-’, and ‘=’ characters, that perform various functions like arithmetic and assignment. Operators are described in detail in the NCL expressions and operators module.

Arrays and subscripts

NCL arrays and subscripts use several special symbols for specifying arrays and subsets of array data. An array expression combines either scalar values or other arrays into a single array result. An array expression is made up of expressions separated by commas (’,’), enclosed in ’(/’ and ’/)’ characters. Arrays can be made up of any of the basic types as well as graphic objects. Arrays of files are not supported. To specify the contents of an entire array, no special symbols are needed. For example, if the array is named temp, then using "temp" with no subscripts specifies the entire contents of the array.

Array subscripts are enclosed in parentheses, ’(’ and ’)’. For example, if temp is a one-dimensional array, then the first element can be specified as temp(0) when using standard subscripting. If temp is a three-dimensional array, then the subscripts are separated by commas. So, the first element in a three-dimensional array is specified as temp(0,0,0) using standard subscripting.

Ranges and strides are specified using the colon ’:’ character. For example, the first through tenth elements in a one-dimensional array are specified as temp(0:9) using standard subscripting. Similarly, this syntax is expanded for variables with multiple dimensions by separating the dimension ranges with commas, as in temp(0:9, 3:7, 2:20). Strides, for thinning data, are appended after the range using another colon. For example, a stride of 3 in a one-dimensional array is specified as temp(0:9:3). Again, this can be expanded to a multi-dimensional array by separating the subscripts with commas, as in temp(0:9:3, 4:7:2, 7:20). This last example uses a stride of 3 in the first dimension, a stride of 2 in the second dimension, and no stride in the third dimension.

Curly braces, ’{’ and ’}’, are used to surround subscripts when using coordinate subscripting. For example, if the coordinates for the temp array range from -200 to 600, then curly braces can be used to select individual elements and ranges. The following NCL statements demonstrate how braces are used.

```
; Select the element that corresponds to coordinate index -150.
; Assume temp is a one-dimensional array.
scalarvar = temp({-150})

; Select a range of data from the beginning to the end.
; Assume temp is a one-dimensional array.
vectorvar = temp({-200:600})

; Select a range of data from a three-dimensional variable
; Assume temp is a three-dimensional array.
var_3D = temp({-200:50}, {45}, {30:100})
```

The vertical bar ’|’ character is used to specify named subscripts. For example, if the coordinate name of the first dimension of the temp variable is called "time", then the vertical bar is used as follows.

```
; Select the first element using named subscripting.
scalarvar = temp((time|0))

; Select ranges using named subscripting. Assume temp is
; a two-dimensional array with dimensions time by lat.

var_2D = temp ((time|0:5), (lat|4:9))

; Select ranges using name subscripting but use names to
; reorder the dimensions of two-dimensional array temp to
; be lat by time

var_2D2 = temp((lat | 4:9),(time | 0:5))
```

Array data

An array expression is made up of expressions separated by commas ',' enclosed in '(/' and '/)' character pairs. Each element separated by commas must have the same dimensionality as the rest of the elements in the array expression.

For example, the following are all NCL statements containing valid array expressions.

```
; Create a one-dimensional array of 5 elements
var1 =  (/ 1, 2, 3, 4, 5 /)

; Create a two-dimensional array of 10 elements
var2 = (/(/ 1, 2, 3, 4, 5 /),(/ 1, 2, 3, 4, 5 /)/)

; Create a two-dimensional array from variable references

var3 = (/var1,var2(0,:),var2(1,:)/)
```

Metadata

Metadata are the named attributes, dimensions, and coordinates that describe data. Several special symbols are used to reference metadata, including the '!', '&', and the '@' symbols.

Dimensions

Variable dimension names are referenced using the '!' symbol. If a variable called temp has three dimensions, then the dimension names are referenced as temp!0, temp!1, and temp!2. Note that the dimension number uses the convention used in C and always starts at 0.

For example, the following NCL statements assign the names "time", "lat", and "lon" to each dimension of the temp variable.

```
temp!0 = "time"
temp!1 = "lat"
temp!2 = "lon"
```

Coordinates

Coordinate variables are referenced using the '&' character. If a variable called temp has three dimensions named time, lat, and lon, respectively, then the coordinate variables along each of its dimensions are referenced as temp&time, temp&lat, and temp&lon.

For example, the following NCL statements assign coordinate vectors (defined elsewhere) to each dimension of the variable called temp.

```
temp&time = vector_of_time_coordinates
temp&lat = vector_of_lat_coordinates
temp&lon = vector_of_lon_coordinates
```

Attributes

NCL attributes are the metadata that describe a variable or data file and its contents. File and variable attributes are referenced using the '@' symbol. If temp is a variable name with attributes called "units" and "max", then the NCL syntax for these references is temp@units and temp@max. File variables can also have attributes. Common file attributes are "title" and "date". The NCL syntax for these references is filevar@title and filevar@date, where filevar is the file variable name.

For example, the following NCL statements demonstrate the assignment of file and variable attributes.

```
; File attributes called "title" and "date"
filevar@title = "South American Climate Data"
filevar@date = "November 23, 1994"

; Variable attributes called "units" and "max"
temp@units = "Degrees C"
temp@max=85.
```

File variable references

The '->' symbol is used to reference variables stored in an external file. For example, if an external file is read into NCL, then a file variable exists that points to descriptive information in a file record. If filevar is the name of the file variable, then references to the external variables and their attributes are made in the following manner.

```
; Print the data from external variable called precip
print (filevar->precip)

; Print the longname of the external precip variable
print (filevar->precip@longname)
```

Expressions and operators

Data types

Numeric data types

There are six numeric data types in NCL: `double, float, long, integer, short, byte`. The float type is the basic type for representing floating point numbers; double will provide the same or more precision than float; integer is the basic type for representing integral values; short will have the same or less precision as integer; byte will almost always be an 8-bit quantity. The algebraic operators operate on the numeric data types.

Non-numeric data types

The following table summarizes the non-numeric data types in NCL:

Data type	Description
character	Used to store a single character
string	A finite sequence of characters
logical	Can have one of two values: True or False
graphic	References an instance of an HLU object
file	References a disk file (returned by addfile)

Coercion of types

For an expression to be evaluated correctly, it is often necessary to convert one of the basic data types into another. This processes is called "coercion" and it is often done automatically. For example, when a float is multiplied by an integer, the integer will be converted to a float, the multiplication performed, and a float calculated for the expression value. The following table shows how the type coercions are performed:

Type	Coercible to
character	string
short	character string integer long logical float double
integer	string long logical float double
long	logical string float double
float	logical string double

Creating data

All numeric data types, strings, characters, logical, and graphic types can be created using the "new" statement. There are two types of new calls: one creates an array of data with a specific missing value assigned to each element; the other creates the array of data using the default missing value. The new function takes, as parameters, an array of dimension sizes, the type keyword to create, and (optionally) a missing value to assign to each element of the new data array. The following creates a three-dimensional float array:

```
a = new((/5,6,7/),float)
```

The following is an example of how to assign a specific missing value.

```
a = new((/5,6,7/),float,-1e12)
```

File data types are created differently. A file type is created only by the addfile function. Arrays of type graphic can be created with new, but none of the elements of the array will actually reference an existing HLU object. Each element will contain a missing value. In addition to new, there are ways of entering data manually. Currently there are four types of literal values. Floating point numbers must be entered with a decimal point. The following show creation of data using literal floating point values:

```
a = 2.
a = 1.2
a = (/ 1., 2., 3. /)
```

Integer values are specified without the decimal point:

```
a = 2
a = (/ 1, 2, 3 /)
```

String values are specified as characters enclosed in double quotes ("):

```
a = "NCAR Graphics"
```

Logical values are specified using the keywords "True" and "False":

```
a = False
```

NCL operators

Summary table

The following table summarizes the primary operators in NCL. The operators are presented in their order of precedence, that is, in the evaluation of an expression, if one operator appears above another in the table, it will be performed first. See the examples of expression evaluation below. If a non-associative operator (such as "^" or "%") occurs successively, then the precedence proceeds left-to-right. For example, 2^3^4 evaluates to 4096.

Operator	Description	Type
->	File variable	metadata
@	Attribute	metadata
&	Coordinate variable	metadata
!	Dimension name	metadata
-	Unary negation	algebraic
^	Exponentiation	algebraic
*	Multiplication	algebraic
/	Division	algebraic
%	Modulus	algebraic
+	Addition	algebraic
-	Subtraction	algebraic
<	Less than selector	algebraic
>	Greater than selector	algebraic
.not.	Logical negation	logical
.le.	Less than or equal to	relational
.lt.	Less than	relational
.ge.	Greater than or equal to	relational
.gt.	Greater than	relational
.ne.	Not equal to	relational
.eq.	Equal to	relational
.and.	And	logical
.or.	Or	logical
.xor.	Exclusive or	logical
=	Assignment	assignment

Detailed descriptions of operators:

File variable (->)

This operator requires two operands. The first operand is a file variable (as created with an addfile command). The second operand is a variable name. The specified variable is either set or retrieved as appropriate. The file variable operator differs from other operators in that the operands are not expressions but identifiers.

Examples:

```
file1->a = 5
file1->a@value = 22.
print(file1->var1)
```

Attribute (@)

This operator takes two operands. The first operand is a variable name, and the second operand is an attribute name for the variable in the first operand. The result can be any type. The attribute operator is similar to the file variable operator in that its operands also must be identifiers and not expressions.

Examples:

```
a@units = "Degrees"
a@value = 3.14159
print(a@units)
```

Coordinate variable (&)

This operator requires two operands. The first operand is a variable name, and the second operand is the name of a dimension of the first operand. The result is a coordinate variable. The values in the coordinate variable must be monotonically increasing or decreasing. If the value assigned is non-monotonic or contains missing values, the assignment still occurs but a WARNING message is generated and attempts to use coordinate subscripting will fail with a FATAL message. The coordinate variable operator only accepts identifiers just like the attribute and file variable operators.

Example:

```
a&Dimension1 = (/.1,.2,.3,.4,.5,.../)
```

Dimension name (!)

This operator takes two operands. The first operand is either a variable or a file variable, and the second operand is an integer. The second operand indicates the dimension number (starting with 0). The result is of type string. This operator is used to retrieve or set names for a variable's dimension(s).

Unary negation (-)

This operator operates on a single operand and that operand must have a numeric type. If the operand is an array, the negation will be applied individually to each element of the array.

Exponentiation (^)

This operator requires two operands. The first operand is raised to the power of the second operand. The operands must be of numeric type. If the first operand is negative, then the second operand must be positive. If either operand is a non-linear array, then the other operand must be a scalar or an array of the same shape. An array raised to a scalar power results in an array of the same size with each element of the original array raised to the scalar power. A scalar raised to an array power will result in array of the same shape as the power where each element of the array is the scalar raised to the power of the array element. An array raised to an array power results in the obvious thing. The result of the exponentiation operator is of type double.

Multiplication (*)

This operator requires two operands. The first operand is multiplied by the second operand. The operands must be of numeric type. Array operands are handled in a manner analogous to that described in exponentiation.

Division (/)

This operator requires two operands. The first operand is divided by the second operand. The operands must be of numeric type. If both operands are of type integer, then the result will be an integer where any decimal remainders are truncated. Array operands are handled in a manner analogous to that described in exponentiation.

Modulus (%)

This operator requires two operands of type integer. The result is an integer type that is the integral remainder of dividing the first operand by the second. Array operands are handled in a manner analogous to that described for exponentiation.

Addition (+)

This operator requires two operands. The first operand is added to the second operand. The operands must both be of numeric type or both of string type. If both operands are of numeric type, then the result will be the sum of the two operands; if both operands are of string type, then the result will be the concatenation of the two strings. Array operands are handled in a manner analogous to that described for exponentiation.

Subtraction (-)

This operator requires two operands. The second operand is subtracted from the first operand. The operands must both be of numeric type. Array operands are handled in a manner analogous to that described for exponentiation.

Less than selector (<)

This operator requires two operands. The result is the smaller of the values in the first operand and the second operand. If one operand is a scalar and the other is an array, then, for each array entry, the smaller of the array element and the scalar is selected for each element of the resultant array. The operands must either both be type numeric, or both type string. For strings string1 and string2, "string1 < string2" yields string1 if the first character of string1 is less than the first character of string2 in the ASCII collating sequence.

Greater than selector (>)

This operator requires two operands. The result is the larger of the values in the first operand and the second operand. If one operand is a scalar and the other is an array, then, for each array entry, the larger of the array element and the scalar is selected for each element of the resultant array. For strings string1 and string2, "string1 > string2" yields string1 if the first character of string1 is greater than the first character of string2 in the ASCII collating sequence.

Logical negation (.ne.)

This operator requires a single logical operand and returns a logical result. If the operand is True, then the result is False; if the operand is False, then the result is True.

Less than or equal to (.le.)

This operator requires two operands of numeric type. The two operands can be a scalar and an array, or two arrays of the same shape. If a scalar is being compared to an array, then an array of logical values is returned as per individual comparisons of the scalar with each array element. If two arrays are being compared, then an array of logical values is returned as per comparing each of the elements in one array with those in the other.

Less than (.lt.)

This operator requires two operands of numeric type. The two operands can be a scalar and an array, or two arrays of the same shape. If a scalar is being compared to an array, then an array of logical values is returned as per individual comparisons of the scalar with each array element. If two arrays are being compared, then an array of logical values is returned as per comparing each of the elements in one array with those in the other.

Greater than or equal to (.ge.)

This operator requires two operands of numeric type. The two operands can be a scalar and an array, or two arrays of the same shape. If a scalar is being compared to an array, then an array of logical values is returned as per individual comparisons of the scalar with each array element. If two arrays are being compared, then an array of logical values is returned as per comparing each of the elements in one array with those in the other.

Greater than (.gt.)

This operator requires two operands of numeric type. The two operands can be a scalar and an array, or two arrays of the same shape. If a scalar is being compared to an array, then an array of logical values is returned as per individual comparisons of the scalar with each array element. If two arrays are being compared, then an array of logical values is returned as per comparing each of the elements in one array with those in the other.

Not equal to (.ne.)
: This operator requires two operands of any type and returns a result of type logical. If a scalar is being compared to an array, then an array of logical values is returned as per individual comparisons of the scalar with each array element. If two arrays are being compared, then an array of logical values is returned as per comparing each of the elements in one array with those in the other.

Equal to (.eq.)
: This operator requires two operands of any type and returns a result of type logical. If a scalar is being compared to an array, then an array of logical values is returned as per individual comparisons of the scalar with each array element. If two arrays are being compared, then an array of logical values is returned as per comparing each of the elements in one array with those in the other.

And (.and.)
: This operator requires two operands of type logical and returns a result of type logical. The result is True if and only if both operands are True.

Or (.or.)
: This operator requires two operands of type logical and returns a result of type logical. The result is True if and only if either of the operands is True.

Exclusive or (.xor.)
: This operator requires two operands of type logical and returns a result of type logical. The result is True if and only if one, but not both, of the operands is True.

Assignment (=)
: The operator requires two operands. The value of the second operand is assigned to the first operand. The type of the second operand must be coercible to the type of the first operand.

NCL Expressions

Definition and examples

An NCL expression is any sequence of operators and operands that results in the computation of a value. In particular, any literal value is an expression, and any variable is an expression. Also, arrays are expressions as well as functions and procedures. Operators applied to expressions are expressions.

Parentheses surrounding an expression result in an expression and can be used to alter the order of evaluation in expressions in the same manner that they are used in most all programming languages. So, for example,

```
a + b / c
```

would be evaluated by dividing b by c and then adding a, whereas

```
(a + b)/ c
```

would be evaluated by adding b and a and then dividing by c.

Here are some examples of expressions:

```
.5
v1
sin(3.1415926)
a(2,3,1)^(b-1) + (file1->a)/2.718
String3 = "String1" + "String2"
(/2,8,11/) % (/2,3,4/)
((v1.lt.v3) .xor. (v2.lt.v4))
```

Missing data in the evaluation of expressions

If a variable is created and no valid value has been assigned to that variable, then that variable will have an attribute named "_FillValue", and the value of the variable will be set to the value of "_FillValue". A pre-defined value of the same type as the variable will be assigned to "_FillValue" unless a value has been specifically assigned by the user. Consider the following NCL session:

```
ncl 0> a = new(2,float)
ncl 1> b = (/ 3., a(0) /)
ncl 2> print(b)
Variable: b
Type: float
Total Size: 8 bytes
            2 values
Number of Dimensions: 1
Dimensions and sizes:   [2]
Coordinates:
(0)     3
(1)     -999
ncl 3> c = (/ 5., 8. /) + b
ncl 4> print(c)
Variable: c
Type: float
Total Size: 8 bytes
            2 values
Number of Dimensions: 1
Dimensions and sizes:   [2]
Coordinates:
(0)     8
(1)     -991
```

The variable "a" is created using the new command and, since a has not been assigned any value, it receives the default _FillValue of -999. This value for a is used in the creation of b, and this becomes the actual value of the second element of b. This value is used in the computation of c.

If the variables making up an expression contain more than one "_FillValue" value, the _FillValue of the left-most variable that contains a missing value is used for the value of the expression.

Expressions and metadata.

The handling of metadata in the computation of expressions presents problems. For example, consider a case where one file named "east" contains a vector variable U and another file named "north" contains a vector variable V. Suppose U has an attribute "long_name" that is set to "east wind component" and V has an attribute called "long_name" that is set to "north wind component". Then, suppose we want to calculate the wind speed and store the result in a variable c:

```
c = sqrt(east->U^2 + north->V^2)
```

Clearly, neither of the values for the attribute "long_name" applies to c. Similarly, attributes like valid_range, valid_min, valid_max, and scale_factor could not be correctly propagated to c. NCL solves this problem simply by not propagating metadata in the evaluation of an expression unless that expression is a single variable. This rule also applies to variables used as parameters to functions.

Examples of expression evaluation.

```
-----------------------------------------------------------------
 Expression                     Evaluates as
-----------------------------------------------------------------
 a < b + c                      a < (b + c)
 a/b*d%c                        ((a/b)*d) % c
 a^b*-c/d+e%f<g                 ((((a^b)*(-c))/d) + (e % f)) < g
 a+b.lt.c*d                     (a+b) .lt. (c*d)
 a.le.c.or.d.gt.e.and.f.ne.g    (a.le.c).or.((d.gt.e).and.(f.ne.g))
 file2->a^3+file2->b*5          ((file2->a)^3) + ((file2->b)*5)
-----------------------------------------------------------------
```

NCL commands

Introduction

This module discusses those statements that are legal to type in--in interactive mode--at the NCL prompt, or those statements that are legal to include in an NCL script. This module also details the functions and procedures that are a part of NCL and how to add to that function set.

NCL statements

A statement is the fundamental element of NCL. Statements are not restricted to being a single line of source code, and statements can be nested within statements. You should keep two important things in mind about how NCL executes its statements:

1. A statement is only executed after it has been parsed and is found to be free of syntax errors.

2. If a runtime error occurs while executing a statement, execution of the statment--and any statements in which the erroneous statement is nested--is immediately terminated.

Flow control

NCL offers several flow control statements that are similar to flow control statements in other computer languages (details are provided by the links):

If statements - These statements, of the form *If-then* and *If-then-else*, implement a decision-making capability in NCL.

Looping - The two looping statements are Do for looping from a starting value to an ending value with a stride and Do while for looping until a condition is met.

break can be used to abort a loop and continue can be used to abort the current iteration of a loop.

Visualization blocks

NCL has three statements for interfacing with the High Level Utilities library:

create - Allows you to create an HLU object and assign values to its resources.
setvalues - Allows you to modify resources in an HLU object after it has been created.
getvalues - Allows you to retrieve the values of the resource setting in an HLU object.

Other statements

block allows you to group statements into a new statement and can be used to require syntax-checking on a block of statements before executing those statements.

An assignment statement allows you to assign a value to any legal NCL construct that can have a value, such as variables, arrays, and so forth.

new is a statement that can be used to initialize variables and arrays and is similar in function to type statements in other languages.

quit is used to exit NCL.

record and stop can be used to start and stop the recording of NCL commands in a file.

Functions and procedures

In NCL, a function or procedure is a name followed by arguments in parentheses with the arguments separated by commas. A function or procedure is invoked by referencing its name and supplying an argument list. A function returns a value, a procedure does not. Arguments can be of any type, but facilities exist in defining a function for imposing restrictions on the arguments, such as numbers of dimensions and dimension sizes. For arguments that must be a certain type, the NCL coercion rules apply. *All arguments are passed by reference in NCL*; this means that a change in the value of an argument within a function or procedure will change the value of the argument in the call. If a parameter is coerced before a function is called, changes within the function *will not* be reflected in the parameter.

General functions and procedures

addfile - Function used to open a new data file (returns a file id).

all - Function that returns True if all of the arguments in the input logical array are True.
any - Function that returns True if any of the arguments in the input logical array is True.
asciiread - Function that reads ASCII files.
asciiwrite - Procedure that writes an ASCII data file from an NCL numeric data type.
cbinread - Function that reads binary files created with a C block I/O function write.
cbinwrite - Procedure that writes a binary file in raw C block I/O format from a numeric data type.
delete - Procedure used to delete variables and free the symbol name from the symbol table.
dimsizes - Function that returns the sizes of the dimensions of any data array.
fbinread - Function that reads binary files created using an unformatted Fortran write.
fbinwrite - Procedure that writes a binary file, using Fortran unformatted writes, of a numeric data type.
getenv - Function that retrieves the values of shell environment variables.
getfilevarnames - Function that retrieves the variable names from a file.
isatt - Function that returns True if a specific attribute name is defined.
isdim - Function that returns True if a specific dimension name is defined.
isfilevar - Function that returns True if a specific file variable name is defined.
isfilevaratt - Function that returns True if a specific file variable attribute name is defined.
isfilevardim - Function that returns True if a specific file variable dimension name is defined.
ismissing - Function that returns True if its argument references a missing value.
list_files - Procedure that lists all of the current variables containing references to files.
list_filevars - Procedure that lists all of the variables associated with a specific file.
list_hlus - Procedure that lists all of the HLU objects currently referenced by NCL.
list_procfuncs - Procedure that lists all the currently defined NCL functions and procedures.
list_vars - Procedure that lists the currently defined variables.
print - Procedure that prints the value of a variable (including file variables) and expressions.
qsort - Procedure that sorts singly-dimensioned numeric arrays in place.
sqsort - Procedure that sorts singly-dimensioned string arrays in place.
rand - Function that generates pseudo random numbers.
sizeof - Function that returns the size, in bytes, of its argument.
sleep - Procedure that pauses execution for a specified number of seconds.
srand - Procedure that establishes a seed for the **rand** function.
system - Procedure that passes UNIX system commands to the shell.

Math functions

Currently supported math functions are: abs , acos , asin , atan , ceil , cos , cosh , fabs , floor , idsfft , log , log10 , sin , sinh , sqrt , tan , tanh .

NCL versions of HLU functions and procedures

NCL contains versions of most of the functions and procedures available to the HLUs. For some of these, such as "draw", "delete" and so forth, the NCL names have been shortened for convenience. The full HLU names are still available in NCL if you want to use them. You may use whichever name you prefer; the arguments will be identical.

clear - Procedure that provides access to the NhlClearWorkstation procedure for clearing workstations.
datatondc - Procedure that provides access to the function NhlDataToNDC for converting from data coordinates to NDC coordinates.
destroy - Procedure that provides access to the NhlDestroy procedure for removing HLU objects.

draw - Procedure providing access to the basic draw function NhlDraw.
frame - Procedure providing access to the HLU picture control function NhlFrame.
ndctodata - Procedure providing access to the function NhlNDCToData for converting from NDC coordinates to data coordinates.
NhlAddAnnotation - Function that adds an annotation to a plot.
NhlAddData - Function for adding one or more additional data items to a plot.
NhlClassName - Function returning the class name of one or more HLU objects.
NhlDataPolyline - Procedure for drawing an immediate mode polyline using data coordinates.
NhlDataPolygon - Procedure for drawing an immediate mode polygon using data coordinates.
NhlDataPolymarker - Procedure for drawing an immediate mode polymarker using data coordinates.
NhlFreeColor - Procedure for removing color index settings from specified workstations.
NhlGetBB - Function for retrieving the bounding boxes of a list of HLU input objects.
NhlIsAllocatedColor - Function for interrogating a workstation to see if a given color index has been allocated.
NhlName - Function that returns the object name of an HLU object, given the HLU object identifier.
NhlNDCPolyline - Procedure for drawing an immediate mode polyline using NDC coordinates.
NhlNDCPolygon - Procedure for drawing an immediate mode polygon using NDC coordinates.
NhlNDCPolymarker - Procedure for drawing an immediate mode polymarker using NDC coordinates.
NhlNewColor - Function used to allocate new workstation color indices.
NhlRemoveAnnotation - Procedure for removing annotations from plots.
NhlRemoveData - Procedure for removing data items from one or more plots.
NhlRemoveOverlay - Procedure for removing one or more plots from an overlay.
NhlSetColor - Procedure for defining workstation color indices.
NhlUpdateData - Procedure that forces an update of the internal state of a **DataComm** object.
overlay - Procedure used to overlay one HLU plot object over a specific style of transformation.
update - Procedure providing access to the HLU update workstation function NhlUpdateWorkstation.

NCL data type conversion functions

Conversion functions exist for every possible data conversion in NCL: chartodouble, chartofloat, chartoint, chartolong, chartoshort, doubletobyte, doubletochar, doubletoint, doubletofloat, doubletolong, doubletoshort, floattobyte, floattochar, floattoint, floattolong, floattoshort, inttobyte, inttochar, inttoshort, longtobyte, longtochar, longtoint, longtoshort, shorttobyte, shorttochar, chartostring, stringtochar.

Intrinsic functions

An intrinsic function is not defined by NCL source; it is a C or Fortran routine that has been added to the NCL function set. Intrinsic functions often perform operations that NCL source does not support. For details on NCL intrinsic functions, see the Intrinsic functions section of the NCL Reference Guide.

Writing your own function or procedure

NCL allows you to define your own functions and procedures. NCL functions and procedures are similar to functions and procedures in most programming languages, but there are some distinct differences. For details on procedures in general, see NCL procedure overview. For specifics on the syntax of defining your own functions and procedures, see NCL functions and procedures definitions.

Adding to NCL's function set

Please see the NCL Reference Manual for a discussion of Extending NCL's function set.

NCL plot creation sequence

This module describes a basic structure for an NCL script used to create and display graphics. Complicated tasks may require a more elaborate structure. After experience with using NCL, you may find a structure that is more suitable to your needs and preferences, but the following will get you started. A very simple example is provided below. For more sophisticated examples, consult the Quick Start Guide.

Basic structure:

1. Create or import any relevant data
2. Create output devices for graphic display
3. Create plot objects
4. Draw plot objects on output devices
5. Modify plot attributes and redraw

Create or import any relevant data

There are several ways that you may create the data needed for your plots in NCL. You may generate the data internally within NCL and store it in an NCL variable--this is illustrated in the simple example below. Also, you may import data from a previously-existing file. Various formats are supported for the imported file. For details on importing and exporting data in NCL, see Importing and exporting data.

After your datasets have been created or imported, you may want to manipulate the data. For example, you may want to convert temperatures from Fahrenheit to Celsius. Or you may want to compute quantities based on the imported data, thereby creating new data fields. NCL has a wide selection of internal functions and procedures to help you with data manipulation. See the module on Data manipulation for details.

Create output devices for graphic display

You need to create workstation objects that specify the output devices on which you want to display your graphics. You can display your view objects in X11 windows, or produce an NCAR CGM file, or produce PostScript output. For more details on workstation objects, see the HLU module on the Workstation class. Also see the module on Managing your output for a discussion of selecting various output devices.

Create plot objects

NCL offers a powerful but simple interface for creating plot objects and for setting and modifying the resource values of such objects. For a discussion of what objects are and how they are used, see the appropriate topics in the High Level Utilities section of the User Guide. For details on creating plot objects in NCL, see the module Visualization specification process.

Draw plot objects on output devices

Once your view objects have been created, it is a simple matter to plot them on any of the output devices you have created. See the module on Visualizing images for details.

Modify plot attributes and redraw

Once you have plotted a view object on a workstation, you may want to modify the values in its resource file and draw it again on the same workstation, or on a different workstation. You may want to scale it or change its position on the output device. For details on how to do these things, see the Manipulating images module.

Delete objects

Before terminating NCL, it is good form to delete all the objects you have created. The NCL procedures delete and destroy can be used for this.

A simple example

Here is an extremely simple NCL script that illustrates the steps mentioned above.

```
;
; Simple NCL script just to illustrate the NCL plot generation structure
; discussed in this module.
;

;
; Draw an X-Y plot with internally-generated data.
;

;
; Step 1 - create linear arrays containing the data to be plotted.
;          This data could have been read in using one of the read
;          functions, or by doing an addfile on a file in a supported
;          data format.
;

x_data = (/ -2., -1., 0., 1., 2., 3. /)
y_data = (/  2.,  1., 0., 1., 2., 3. /)

;
; Step 2 - create an X11 window to use as an output device for displaying
;          the plot. Set the pause option so that a call to "frame" will
;          pause waiting for a mouse click or keyboard entry.
;

xworkid = create "simple" xWorkstationClass defaultapp
    "wkPause" : "True"
end create
```

```
;
; Step 3 - create an XyPlot object using the data in x_data and y_data.
;

;
;           First create the data object.
;

dataid = create "xyData" coordArraysClass defaultapp
    "caXArray": x_data
    "caYArray": y_data
end create

;
;           Next, create the XyPlot object which is a child of the
;           XWorkstation object. Set the appropriate resource
;           to force recomputation of plot limits when the data
;           is changed.
;

plotid = create "xyPlot" xyPlotClass xworkid
    "xyCoordData": dataid
    "xyComputeYMax": "True"
end create

;
; Step 4 - Draw the plot.
;

draw(plotid)
frame(xworkid)

;
; Step 5 - Modify the data and redraw.
;

y_data = y_data^2

;
;           Modify the data object.
;
setvalues dataid
    "caYArray": y_data
end setvalues

;
;           Inform the plot object of the data change.
;
setvalues plotid
    "xyCoordData": dataid
end setvalues

;
;           Redraw
;
draw(plotid)
frame(xworkid)
```

Importing and exporting data

NCL supports import and export for several data types including netCDF and HDF. Here is a list of supported operations:

- netCDF import
- netCDF export
- GRIB import
- HDF import
- HDF export
- binary file import
- binary file export
- ASCII file import
- ASCII file export

Support for importing GRIB files has been add in Version 4.0.1 of the package.

ASCII files

Import

Support for importing ASCII files is provided by the NCL asciiread function. This function returns all data that is read in to an NCL variable. For example, suppose a local file named "float_data" looked like:

```
-1. 0. 1. 2.
```

then executing the NCL commands:

```
input_data = asciiread("float_data",4,"float")
print(input_data)
```

yields:

```
Variable: input_data
Type: float
Total Size: 16 bytes
            4 values
Number of Dimensions: 1
Dimensions and sizes:   [4]
Coordinates:
Number Of Attributes: 1
_FillValue
(0)     -1
(1)     0
(2)     1
(3)     2
```

For numeric data, the format of the data file is free-form and all non-numeric characters are ignored. So, for example, if the file "int_data" looked like:

```
 1 <--First data item, 2 <--Second data item
 3, 4,
 5XXX6YYY
```

then executing the NCL commands:

```
input_data = asciiread("int_data",(/2,3/),"integer")
print(input_data)
```

yields:

```
Variable: input_data
Type: integer
Total Size: 24 bytes
            6 values
Number of Dimensions: 2
Dimensions and sizes:   [2] x [3]
Coordinates:
Number Of Attributes: 1
_FillValue
(0,0)   1
(0,1)   2
(0,2)   3
(1,0)   4
(1,1)   5
(1,2)   6
```

If there are more data values in the file than are requested, then only the requested number of values are used. If there are fewer data values in the file than are requested, then the missing values are assigned the default missing value for the given data type. If the data type in an asciiread is "string", then each line of the input file is returned as a single data value.

Export

Support for exporting ASCII files is provided by the NCL asciiwrite procedure. For example, executing the following NCL commands:

```
a = (/1,2,3/)
asciiwrite("int_file",a)
```

would create a file named "int_file" that looks like:

```
1
2
3
```

Binary files

Import

Binary files created in C blocked binary I/O format (by a call to the C procedure fwrite for example) can be imported directly into NCL by using the NCL cbinread function. For example, if "c_bin_data" is a binary file as described that contains three floating point numbers, then the NCL command

```
fval = cbinread("c_bin_data",3,"float")
```

would store the three values in "c_bin_data" in the NCL array fval. If there are more values in the data file than specified in the second argument to cbinread, then the requested number of data values will be returned; if there are not as many values in the data file as requested, then the missing values are assigned the default missing value for the given data type.

Note: The data file read in by cbinread must have been created on a machine of the same architecture as the one that NCL is being run on. For example, if you created a binary file on a machine with a 64-bit word size and tried to read it back on a 32-bit machine, you will not get what you want.

Binary files created using Fortran unformatted writes can be imported directly into NCL by using the NCL fbinread function. For example, if "f_bin_data" is a binary file as described that contains three floating point numbers, then the NCL command

```
fval = fbinread("f_bin_data",3,"float")
```

would store the three values in "f_bin_data" in the NCL array fval. If there are more values in the data file than specified in the second argument to fbinread, then the requested number of data values will be returned; if there are not as many values in the data file as requested, then an *error* will result.

Note: The data file read in by fbinread must have been created on a machine of the same architecture as the one that NCL is being run on.

Export

The NCL procedure cbinwrite can be used to write a file in C blocked I/O format from any NCL numeric variable. For example:

```
fval = (/ 1., 2., 3. /)
cbinwrite("c_bin_data",fval)
```

would write the binary file "c_bin_data". NCL automatically figures out how many values to write and the data type being written.

Writing binary files using a Fortran unformatted write with fbinwrite is entirely analogous to using cbinwrite.

netCDF files

netCDF (network Common Data Form) is an interface for scientific data access and a library that provides an implementation of the interface. The netCDF library also defines a machine-independent format for representing scientific data. Together, the interface, library, and format support the creation, access, and sharing of scientific data. The netCDF software was developed at the Unidata Program Center in Boulder, Colorado.

Import

netCDF files are imported into NCL via the addfile function. NCL has several operators that apply to netCDF files; these operators are the metadata operators discussed in the operator summary table of the Expressions and operators module. Any netCDF file must have a suffix of ".nc" or ".cdf". NCL uses these suffixes to determine that the file is a netCDF file.

As an example, suppose the file "simple.cdl" (cdl files are ASCII versions of netCDF files that can be converted to netCDF files by using the netCDF library function *ncgen*) contains the following simple file:

```
netcdf simple {  // A very simple file for demo purposes.

dimensions:
    xdim = 5;

variables:

    float        xarray(xdim);
                 xarray:long_name = "A small array";
                 xarray:units = "dimensionless";
                 xarray:valid_range = 0.f, 7.f ;
                 xarray:_FillValue = -9999.0f;

    float        xdim(xdim);
                 xdim:long_name = "X array dimension";
                 xdim:units = "dimensionless";

    :history = "created by SCD for demo purposes";
    :title = "Simple demo";

data:

 xdim = -20, -10,   0,  10,  20;

 xarray = -1., 0., 1., 2., 3.;
}
```

Let simple.cdf be the netCDF file created from the simple.cdl file by using ncgen. The following NCL interactive session illustrates importing simple.cdf into NCL and using some of the metadata operators:

```
ncl 0> file1 = addfile("simple.cdf","r")
ncl 1> print (file1)

filename:       simple
path:   simple.cdf
        dimensions:
                xdim = 5
        variables:
                float xarray(xdim)
```

```
                float xdim(xdim)
ncl 2> print(file1->xarray)


Variable: xarray (file variable)
Type: float
Total Size: 20 bytes
            5 values
Number of Dimensions: 1
Dimensions and sizes:   [xdim | 5]
Coordinates:
            xdim: [xx..xx]
Number Of Attributes: 4
long_name
units
valid_range
_FillValue
(0)     -1
(1)     0
(2)     1
(3)     2
(4)     3
ncl 3> print(file1!0)
(0)     xdim
ncl 4> print(file1@history)
(0)     created by SCD for demo purposes
ncl 5> print(file1->xarray({-10:10}))


Variable: xarray (file variable subsection)
Type: float
Total Size: 12 bytes
            3 values
Number of Dimensions: 1
Dimensions and sizes:   [xdim | 3]
Coordinates:
            xdim: [xx..xx]
Number Of Attributes: 4
long_name
units
valid_range
_FillValue
(0)     0
(1)     1
(2)     2
```

Export

netCDF files can be created directly from NCL by specifying the "c" option in an addfile command. Consider the following NCL session:

```
ncl 0> file1 = addfile("file1.cdf","c")
ncl 1> file1->xarray = (/2.,3.,4.,5.,6./)
ncl 2> file1!0 = "xdim"
ncl 3> file1&xdim = (/-20,-10,0,10,20/)
ncl 4> file1->xarray@units = "millimeters"
ncl 5> file1@title = "NCL-created netCDF"
ncl 6> quit
```

This creates a netCDF file "file1.cdf". A dump of this netCDF file obtained by using the "ngdump" utility in the netCDF library yields:

```
netcdf file1 {
dimensions:
        xdim = 5 ;

variables:
        float xarray(xdim) ;
                xarray:units = "millimeters" ;
        long xdim(xdim) ;

// global attributes:
                :title = "NCL-created netCDF" ;

data:

 xarray = 2 , 3 , 4 , 5 , 6  ;

 xdim = -20, -10, 0, 10, 20 ;
}
```

Existing netCDF files can be modified by importing them using "addfile" with a final argument of "w" and making the desired changes. The imported netCDF file will be overwritten when NCL is terminated.

HDF files

HDF (Hierarchical Data Format) is a multi-object file format, developed at NCSA, that facilitates the transfer of various types of data between machines and operating systems.

Import

Importing HDF files is entirely analogous to importing netCDF files as described above, with some minor exceptions. First, if variable names contain spaces or non-alphanumeric characters these characters are replaced with the '_' character when listed from NCL and are referenced from NCL in this fashion.

Export

Although HDF and netCDF share the same NCL interface and can be accessed through the same types of NCL statements, exporting HDF files from NCL is somewhat limited currently. Only the values of variables and their names are written to files in the HDF format. Attributes and dimension names can not currently be written from NCL. Attributes and dimension names will generate warning messages when attempting to write an NCL variable to an HDF file.

Also there is currently no way to access 8 bit and 24 bit HDF images from NCL.

GRIB import

GRIB was added with release 4.0.1. See Information on Supported Data Formats section of the reference guide for more information.

Data manipulation

Introduction

This module discusses the basic tools that NCL provides for manipulating data. This section is somewhat all-inclusive since a good part of what NCL is designed to do is manipulate data.

Importing and exporting data

NCL contains several functions and procedures for importing and exporting data in binary, ASCII, netCDF and HDF formats. See the module Importing and exporting data for details on this.

Searching and sorting, random numbers

NCL contains a function for sorting singly-dimensioned numeric arrays qsort and one for sorting string arrays, sqsort. Also available are rand and srand for generating random numbers.

Math functions

NCL contains all of the standard trig functions as well as several other useful math functions.

Data and plots

You can add data objects to plots using the NCL function NhlAddData and you can remove data objects using the NCL procedure NhlRemoveData.

Data conversion

In addition to the NCL procedures NhlDataToNDC and NhlNDCToData for converting user data to and from Normalized Device Coordinates, there exists every imaginable function for converting between data types.

NCL operators

Any of the basic NCL metadata, algebraic, logical, relational, or assignment operators can be used to manipulate data.

Array processing

One of the most valuable features of NCL is its ease of processing array data. Most of the basic operators can be applied to arrays. Also, array subsections are easily extracted and processed, and it is easy to rearrange array data and create new arrays from existing arrays.

Flow control and procedures

The NCL flow control statements are useful in manipulating data. Defining your own procedures can be useful as well.

Examples

Almost any of the NCL examples available in the Quick Start Guide will illustrate NCL data manipulation. The example xy14n.ncl is specifically designed to illustrate the basic data manipulation tools.

Visualization specification process

Introduction

This module discusses how to create HLU objects in NCL and how to change values in and retrieve values from those objects. Also discussed here is how to do overlays, that is, how to superimpose one plot on top of another.

Creating HLU objects

The way to create HLU objects in NCL is to use the create expression that is part of an NCL visualization block. The create expression is the NCL analog of the HLU NhlCreate function. Here is a simple example of using the NCL create expression to create a simple TickMark object:

```
    box_id = create "Box" tickMarkClass xworkid
        "tmXBDataLeftF": 0.
        "tmXBDataRightF": 1.
        "tmYLDataTopF": 1.
        "tmYLDataBottomF": 0.
        "tmYLLabelsOn": "False"
    end create
```

In this example, box_id is returned as the object id of the TickMark object that is created; "Box" is the name of the created object; tickMarkClass is the class of object being created; xworkid is an id of an object to be used as the parent of the object being created. In this case, xworkid would be an object identifier for an X11 workstation that would need to have been created before the above create is done. Note how easy it is to specify values for resources in NCL creates.

Changing resource values

If you want to change the value of a resource in an object that has already been created, you can do this with the NCL setvalues expression. For example, suppose that after you did the create above, you wanted to specify a new value for the left X-axis value. You could do this with:

```
setvalues box_id
   "tmXBDataLeftF": -1.
end setvalues
```

Values for HLU resources can also be assigned via a resource file as well as in the NCL visualization blocks. For details on setting values for resources, see the document Understanding resources in the HLU User Guide. To set resource values via a resource file, you must create an App object to supply the base name of the resource file. For example, the following NCL create expression would provide access to a resource file named "my_resources.res":

```
appid = create "my_resources" appClass defaultapp
   "appDefaultParent" : "True"
end create
```

Retrieving resource values

Values for resources can easily be retrieved in NCL by using the getvalues expression. For example,

```
getvalues box_id
   "tmXBDataLeftF": x_left_value
end getvalues
```

would return the current value for tmXBDataLeftF in the NCL variable x_left_value.

Overlays

Overlays allow you to have two or more transformation objects share the same data coordinate space. This allows you to superimpose one plot on another and have them share the same data transformations. Overlays are created starting with a base plot and adding transform objects using the NCL *overlay* procedure. Note that the base plot must be a plot object, that is an object belonging to the transform class that contains a PlotManager instance that is not itself an overlay. The only classes that can be instantiated as plot objects are those with the word plot in the class name. The NCL function NhlRemoveOverlay can be used to remove one or more plots from an overlay.

A thorough description of overlays and annotations is contained in the reference page for PlotManager.

Presented below is a very simple example of how to use the NCL overlay procedure. This example is for illustration only, since you would not want to create an overlay to do what it does (superimpose a curve on a background box). You would just use a single XyPlot object to do this. For more sophisticated examples of using overlays, see cn05n.ncl, xy10n.ncl, and mp03n.ncl. The Quick Start Guide explains how to run these examples.

```
;
; Simple example of using the NCL overlay procedure.  This example
; is for illustration only - it does not reflect real-world usage.
;
```

```
;
; Create an X11 output workstation.
;

xworkid = create "simple" xWorkstationClass defaultapp
    "wkPause" : "True"
end create

;
; Create an XyPlot object that is just a box.  This object will be
; used as the base plot.
;

box_id = create "Box" xyPlotClass xworkid
    "tmXBLabelsOn": "False"
    "tmYLLabelsOn": "False"
    "tmXBOn": "False"
    "tmXTOn": "False"
    "tmYLOn": "False"
    "tmYROn": "False"
;
;   Specify the plot limits to conform with those of the subsequent overlay.
;
    "trXMinF": -1.
    "trXMaxF":  1.
    "trYMinF":  0.
    "trYMaxF":  1.
end create

;
; Create data for another XyPlot object that is simply a curve.  This
; object will be used as an overlay to the base plot.
;

dataid = create "xyData" coordArraysClass defaultapp
    "caXArray": (/ -1., 0., 1./)
    "caYArray": (/  0., 1., 0./)
end create

;
; Create the XyPlot overlay (by default PlotManager will not
; create a TickMark object for this object since pmTickMarkDisplayMode
; is set to "NoCreate").
;

curve_id = create "Curve" xyPlotClass xworkid
    "xyCoordData": dataid
end create

;
; Overlay the curve on the box.
;

overlay(box_id,curve_id)

;
; Draw the plot which is the curve overlaid on the box.
;

draw(box_id)
frame(xworkid)
```

Visualizing images

This module shows you how to create output workstations and how to send view objects to them for viewing. You will also see how you can send the same object to different output workstation.

Creating output workstations

After you have created a view object, in order to view the object you must create an output workstation and send your view objects to the workstation. You can create several workstations and send view objects to any or all of them.

There are three different types of workstation classes you can use for creating workstation objects for sending view objects to:

1. XWorkstation for plotting to an X11 window.
2. NcgmWorkstation for storing graphics in NCGM format.
3. PSWorkstation for storing graphics in various flavors of PostScript.

Each of the above workstation objects has resources for controlling various aspects of the output, such as metafile name for NCGM output, pausing in an X11 window, selecting PS or EPS output for PostScript, and so on. Consult the reference pages for these workstations for details. All of the above classes are subclassed from the Workstation class. The Workstation class has a multitude of resources itself, such as setting color maps. See the reference page for the Workstation class for details.

Only one NcgmWorkstation can be created at a time in an application, but up to 15 independent XWorkstations or PSWorkstations can be created.

Workstations are created in NCL by using the create expression. For example, the following code would create an instance of a PostScript workstation that produces EPS.

```
xworkid = create "simple" psWorkstationClass defaultapp
    "wkPSFormat" : "EPS"
    "wkBackgroundColor"  : (/ 1., 1., 1. /)
end create
```

The workstation name "simple" is used as the base name for the EPS output, so that when view objects are drawn to xworkid, the output file will have the name "simple.eps".

Sending view objects to output workstations

The NCL function used to send a picture represented by a viewable object to an output device is draw. This function takes a single argument that contains an array of one or more view object identifiers. The objects identified in the argument to the NCL draw procedure are sent to the workstation identified as the parent in the create expression that was used to create the object.

Here is a simple example:

```
;
; NCL script that creates an output workstation and sends view
; objects to it.

;
; Create an X11 output workstation.
;

xworkid = create "x_work" xWorkstationClass defaultapp
    "wkPause" : "True"
end create

;
; Create a couple of TextItem objects.
;

text_id1 = create "Text" textItemClass xworkid
    "txPosXF": 0.5
    "txPosYF": 0.4
    "txString": "Text String 1"
end create
text_id2 = create "Text" textItemClass xworkid
    "txPosXF": 0.5
    "txPosYF": 0.6
    "txString": "Text String 2"
end create

;
; Draw one string on the workstation and pause.
;

draw(text_id1)
frame(xworkid)

;
; Draw both strings on the workstation and pause.
;

draw((/text_id1,text_id2/))
frame(xworkid)
```

For most applications, you will call the NCL procedure frame after a draw in order to terminate an image. For more precise control, you may want to use the NCL procedures update and clear.

Sending the same object to one or more output workstations

Sometimes you would like to send the same object to more than one workstation. The NCL draw procedure sends an object identified in its argument to the workstation whose identifier was specified in the create expression that was used to create the object. In order to send an object to a different workstation than the one identified in its creation, it is necessary to specify a new parent for the object. This can be done by using the NhlChangeWorkstation procedure.

Here is a simple example of sending the same object to two different workstations:

```
;
; Create a PS output workstation with a white background color.
;

psworkid = create "simple" psWorkstationClass defaultapp
    "wkBackgroundColor"  : (/ 1., 1., 1. /)
end create

;
; Create an X11 workstation.
;

xworkid = create "X11" xWorkstationClass defaultapp
    "wkPause" : "True"
    "wkBackgroundColor"  : (/ 1., 1., 1. /)
end create

;
; Create data for an XyPlot
;

dataid = create "xyData" coordArraysClass defaultapp
    "caXArray": (/0.0, 0.1, 0.5, 0.9, 1.0, 0.9, 0.5, 0.1, 0.0/)
    "caYArray": (/0.5, 0.9, 1.0, 0.9, 0.5, 0.1, 0.0, 0.1, 0.5/)
end create

;
; Create a simple XyPlot object.
;

plot_id = create "Box" xyPlotClass xworkid
   "xyCoordData": dataid
end create

;
; Send the XyPlot to the X11 workstation.
;

draw(plot_id)
frame(xworkid)

;
; Send the XyPlot to the PS workstation.
;

NhlChangeWorkstation(plot_id,psworkid)
draw(plot_id)
frame(psworkid)
```

Manipulating images

This module discusses how to plot and position your viewable objects on an output workstation. It also discusses how to modify and redraw an object once it has been created.

Positioning images

In order to plot an object on an output workstation, that object has to be a view object. To position a view object on an output workstation, you use the viewClass resources vpXF, vpYF, vpWidthF, and vpHeightF. The resources *vpXF* and *vpYF* specify a coordinate point in NDC space that references the upper left corner of the plot object on an output workstation. The viewClass resource *vpWidthF* specifies how wide the view object will be (again in NDC units), and *vpHeightF* specifies how tall the view object will be. For example, the settings:

```
"vpXF": .4
"vpYF": .6
"vpWidthF": .3
"vpHeightF": .3
```

would specify that the view object would be displayed with its upper left corner at NDC coordinate (0.4, 0.6) in a square of size 0.3 NDC units.

If *vpWidthF* and *vpHeightF* do not specify a square, then the view object will be made to fit into the specified rectangle, unless the viewClass resource *vpKeepAspect* is set to "True", in which case the view object will be constrained to fit in the largest box, with aspect ratio matching the original shape, that can be inscribed within a box of the specified size. *vpKeepAspect* is "False" by default, so that a view object will be made to fit the rectangle specified by *vpWidthF* and *vpHeightF*.

If the view object is positioned so that part (or all) of it would go outside of NDC limits, then the object is scaled in such a manner to make it fit in NDC space. A warning message is issued in this case.

Modifying images

Once an object is created and plotted to an output workstation, its resources can be modified and the object redrawn. In particular, the positioning resources, mentioned in the above paragraphs, can be modified and the object redrawn.

The NCL expression to use to modify resources in an object that has been created is setvalues. For example, suppose that an XyPlot object has been created and you want to reposition it for plotting to an output workstation. You can do this with the following:

```
setvalues id_of_object
  "vpXF": new_xpos
  "vpYF": new_ypos
  "vpWidthF": new_width
  "vpHeightF": new_height
end setvalues
```

Example

The topics in this module are illustrated in the NCL example basic06n. This example also illustrates setting up a color table and changing colors of items in objects and redrawing the objects. The Quick Start Guide explains how to run this example.

Managing your output

This module discusses how to send your view objects to an output workstation for viewing.

Creating output workstations

After you have created a view object, in order to view the object you must create an output workstation and send your view objects to the workstation. You can create several workstations and send view objects to any or all of them.

There are three different types of workstation classes you can use for creating workstation objects for sending view objects to:

1. XWorkstation for plotting to an X11 window.
2. NcgmWorkstation for storing graphics in NCGM format.
3. PSWorkstation for storing graphics in various flavors of PostScript.

Each of the above workstation objects has resources for controlling various aspects of the output, such as metafile name for NCGM output, pausing in an X11 window, selecting PS or EPS output for PostScript, and so on. Consult the reference pages for these workstations for details. All of the above classes are subclassed from the Workstation class. The Workstation class has a multitude of resources itself, such as setting color maps. See the reference page for the Workstation class for details.

Only one NcgmWorkstation can be created at a time in an application, but up to 15 independent XWorkstations or PSWorkstations can be created.

Workstations are created in NCL by using the create expression. For example, the following code would create an instance of a PostScript workstation that produces EPS.

```
xworkid = create "simple" psWorkstationClass defaultapp
    "wkPSFormat" : "EPS"
    "wkBackgroundColor"  : (/ 1., 1., 1. /)
end create
```

The workstation name "simple" is used as the base name for the EPS output, so that when view objects are drawn to xworkid, the output file will have the name "simple.eps".

Sending view objects to output workstations

The basic NCL procedure for sending view objects to an output device is draw. See the module Visualizing images for details.

For most applications, you will call the NCL procedure frame after a draw in order to terminate an image. For more precise control, you may want to use the NCL procedures update and clear.

Sending the same object to one or more output workstations

Sometimes you would like to send the same object to more than one workstation. The NCL draw procedure sends an object identified in its argument to the workstation whose identifier was specified in the create expression that was used to create the object. In order to send an object to a different workstation than the one identified in its creation, it is necessary to specify a new parent for the object. This can be done by using the NhlChangeWorkstation procedure.

Here is a simple example of sending the same object to two different workstations:

```
;
; Create a PS output workstation with a white background color.
;

psworkid = create "simple" psWorkstationClass defaultapp
    "wkBackgroundColor"  : (/ 1., 1., 1. /)
end create

;
; Create an X11 workstation.
;

xworkid = create "X11" xWorkstationClass defaultapp
    "wkPause" : "True"
    "wkBackgroundColor"  : (/ 1., 1., 1. /)
end create

;
; Create data for an XyPlot
;

dataid = create "xyData" coordArraysClass defaultapp
    "caXArray": (/0.0, 0.1, 0.5, 0.9, 1.0, 0.9, 0.5, 0.1, 0.0/)
    "caYArray": (/0.5, 0.9, 1.0, 0.9, 0.5, 0.1, 0.0, 0.1, 0.5/)
end create

;
; Create a simple XyPlot object.
;

plot_id = create "Box" xyPlotClass xworkid
   "xyCoordData": dataid
end create


;
; Send the XyPlot to the X11 workstation.
;

draw(plot_id)
frame(xworkid)

;
; Send the XyPlot to the PS workstation.
;

NhlChangeWorkstation(plot_id,psworkid)
draw(plot_id)
frame(psworkid)
```

Batch mode instructions (NCL)

As described in the How to run NCL section of the NCL user environment module, one of the ways of running NCL is in batch mode. This involves creating and running NCL scripts.

Creating NCL scripts

An NCL script is an ASCII file that contains valid NCL statements and comment lines. You can create NCL scripts using whatever text editor you normally use. All of the sources for the NCL examples in the Quick Start Guide are NCL scripts.

Running NCL scripts

Once you have created an NCL script, say in file *my_ncl*, you run it by passing it to NCL's input stream by issuing the command:

```
ncl < my_ncl
```

The NCL interpreter will proceed to interpret, in sequence, each NCL statement in the input script. If you want NCL to check the entire script for syntax errors before proceeding, make the entire script an NCL block by surrounding it with a *begin* at the start and an *end* at the end. Most of the example NCL scripts illustrate this usage of NCL blocks.

Loading NCL scripts with the NCL load statement

NCL scripts can be inserted into other NCL scripts by using the NCL *load* statement. The NCL *load* statement is similar to the *#include* statement of the C programming language. The form of the *load* statement is:

```
load "file_name"
```

where *file_name* contains an NCL script. This can be any valid NCL script, but frequently the load statement is used to include independently-defined NCL functions and procedures. The file name argument to the load statement must be a string literal.

Suppose *file1* contains the single NCL statement:

```
a = 5
```

and *file2* contains the statements:

```
load "file1"
print(a)
```

then executing

```
ncl < file2
```

produces the result:

```
Variable: a
Type: integer
Total Size: 4 bytes
            1 values
Number of Dimensions: 1
Dimensions and sizes:   [1]
Coordinates:
(0)     5
```

NCL API overview

In addition to functioning as a command line interpreter, NCL has an API for submitting NCL scripts from within an application. The API provides GUI and application writers a uniform way to do data access, manipulation, and visualization from within their application using NCL scripts. This should simplify the implementation of application-specific GUIs by reducing design time spent on the visualization code and the data access code.

This module for the NCL User Guide is currently being developed. For now, consult the NCL API documentation in the NCL Reference Guide.

Compiling and linking NCL programs

How to compile and link an NCL program

Good programming practices

This module gives useful tips for efficient programming with NCL.

Fortran vs. C

This module describes the advantages and disadvantages of using NCL with Fortran programs and with C programs.

Library names

This module tells users what they need to know about NCL API libraries.

Header files

This module tells users what they need to know about NCL API header files.

NCL API examples

This module provides API examples to help users understand APIs and how to use them.

NCL: Putting it all together

It is assumed that you have read most of the other modules in the NCL User Guide by the time you read this. This module takes you through an example that illustrates the main features of the NCL language.

NCL is a powerful tool for data exploration and analysis. It is particularly useful in examining netCDF files. Recall the information on the NCL data model from the data model module.

A typical NCL application will read in a netCDF file for analysis. Datasets in the netCDF file will be plotted. Subsets of datasets are easily extracted (as discussed in the NCL array subscripting module) and plotted.

The cn10n.ncl example is considered in this module. The Quick Start Guide explains how to run this example.

The example begins by reading in a netCDF file using the NCL addfile function. This netCDF file was created from an ASCII ".cdl" file by using the netCDF *ncgen* command. For reference, here is the original ASCII source used to create the netCDF file:

```
netcdf mound {  // A mound to use as a demo for NCL data manipulation
                // and plotting.

dimensions:
    xdim = 15;
    ydim = 18;

variables:
```

```
    float       mound(xdim,ydim);
                mound:long_name = "2-D mound for data demo";
                mound:units = "dimensionless";
                mound:valid_range = 0.f, 7.f ;
                mound:_FillValue = -9999.0f;

    float       xdim(xdim);
                xdim:long_name = "X axis dimension";
                xdim:units = "dimensionless";

    float       ydim(ydim);
                ydim:long_name = "Y axis dimension";
                ydim:units = "dimensionless";

    :history = "created by SCD for demo purposes";
    :title = "Two dimensional mound for data demo";

data:

 xdim = -40, -30, -20, -10,   0,  10,  20,  30,  40,  50,  60,  70,
         80,  90, 100;

 ydim = -60, -50, -40, -30, -20, -10,   0,  10,  20,  30,  40,  50,
         60,  70,  80,  90, 100, 110;

mound =

34.81, 36.60, 38.06, 39.20, 40.01, 40.50, 40.66, 40.50, 40.01,
39.20, 38.06, 36.60, 34.81, 32.70, 30.26, 27.50, 24.41, 21.00,

36.56, 38.35, 39.81, 40.95, 41.76, 42.25, 42.41, 42.25, 41.76,
40.95, 39.81, 38.35, 36.56, 34.45, 32.01, 29.25, 26.16, 22.75,

37.81, 39.60, 41.06, 42.20, 43.01, 43.50, 43.66, 43.50, 43.01,
42.20, 41.06, 39.60, 37.81, 35.70, 33.26, 30.50, 27.41, 24.00,

38.56, 40.35, 41.81, 42.95, 43.76, 44.25, 44.41, 44.25, 43.76,
42.95, 41.81, 40.35, 38.56, 36.45, 34.01, 31.25, 28.16, 24.75,

38.81, 40.60, 42.06, 43.20, 44.01, 44.50, 44.66, 44.50, 44.01,
43.20, 42.06, 40.60, 38.81, 36.70, 34.26, 31.50, 28.41, 25.00,

38.56, 40.35, 41.81, 42.95, 43.76, 44.25, 44.41, 44.25, 43.76,
42.95, 41.81, 40.35, 38.56, 36.45, 34.01, 31.25, 28.16, 24.75,

37.81, 39.60, 41.06, 42.20, 43.01, 43.50, 43.66, 43.50, 43.01,
42.20, 41.06, 39.60, 37.81, 35.70, 33.26, 30.50, 27.41, 24.00,

36.56, 38.35, 39.81, 40.95, 41.76, 42.25, 42.41, 42.25, 41.76,
40.95, 39.81, 38.35, 36.56, 34.45, 32.01, 29.25, 26.16, 22.75,

34.81, 36.60, 38.06, 39.20, 40.01, 40.50, 40.66, 40.50, 40.01,
39.20, 38.06, 36.60, 34.81, 32.70, 30.26, 27.50, 24.41, 21.00,

32.56, 34.35, 35.81, 36.95, 37.76, 38.25, 38.41, 38.25, 37.76,
36.95, 35.81, 34.35, 32.56, 30.45, 28.01, 25.25, 22.16, 18.75,

29.81, 31.60, 33.06, 34.20, 35.01, 35.50, 35.66, 35.50, 35.01,
34.20, 33.06, 31.60, 29.81, 27.70, 25.26, 22.50, 19.41, 16.00,

26.56, 28.35, 29.81, 30.95, 31.76, 32.25, 32.41, 32.25, 31.76,
```

```
30.95, 29.81, 28.35, 26.56, 24.45, 22.01, 19.25, 16.16, 12.75,

22.81, 24.60, 26.06, 27.20, 28.01, 28.50, 28.66, 28.50, 28.01,
27.20, 26.06, 24.60, 22.81, 20.70, 18.26, 15.50, 12.41,  9.00,

18.56, 20.35, 21.81, 22.95, 23.76, 24.25, 24.41, 24.25, 23.76,
22.95, 21.81, 20.35, 18.56, 16.45, 14.01, 11.25,  8.16,  4.75,

13.81, 15.60, 17.06, 18.20, 19.01, 19.50, 19.66, 19.50, 19.01,
18.20, 17.06, 15.60, 13.81, 11.70,  9.26,  6.50,  3.41,  0.00;

}
```

This file has a simple 2-D dataset called "mound" that represents a smooth mound. It should be instructive to refer to this file as you read through example cn10n.ncl.

After the netCDF file is read in, a quick-and-dirty contour plot is drawn of the mound. After this plot, an improved contour plot is drawn by using color fill and modifying the label fonts, and so forth. In the third picture, a subregion of the original data is selected and plotted. In the next plot, a slice is taken through the data with X held constant, and a curve is drawn by creating an XyPlot object and plotting it. In the final plot, the original 2-D array is contoured and overlayed on top of the state of Colorado using MapPlot to draw a Lambert Conformal projection of an appropriate region of the United States.

This example illustrates most of the important features of NCL as well as how NCL interacts with plot objects of the HLU library.

Troubleshooting NCL

This module discusses various tools for figuring out what went wrong with an NCL program execution.

Syntax errors

Most syntax error messages should be self-explanatory. Since NCL is an interpreted language, syntax errors will not be reported until they are encountered. However, if you want to have an entire NCL script checked for syntax errors before execution, you can enclose the entire script in a *begin* and *end* statement. That is, put a *begin* at the beginning of the script and an *end* at the end. This will make the entire script into an NCL block. Since blocks are not executed until they are checked for syntax, the entire NCL script will be checked for syntax before proceeding.

Using the NCL print procedure

There is no high-level debugger for NCL like *dbx*. Most of the time, the error messages issued by NCL will be enough to figure out what the problem is. If not, you may want to use the NCL print procedure.

The procedure *print* can be used to print, on the standard output unit, values of any NCL variable or data structure. When printing arrays, you may want to take advantage of the ability to easily select subsets of the array. This is discussed in the NCL array subscripting module.

If you just want to print an informational message, you can pass a string literal to *print*. For example:

```
print("Execution got to this point")
```

Changed values not reflected in a plot

If you assign a value to an HLU resource in a resource file and that change is not reflected in your plot, make sure that you have not assigned a value to the resource in an NCL create statement, or in an NCL setvalues statement. Values set in a *create* or *setvalues* statement will take precedence over those set in a resource file.

Really stumped?

The NCAR Graphics consulting staff is available via e-mail at graphuse@ncar.ucar.edu. Installation consulting is available at ncargfx@ncar.ucar.edu. You can also send comments to us by simply clicking on the Feedback icon (the one with the ear) in the control panel that appears in every documentation module.

Low Level Utilities overview

The Low Level Utilities (LLUs) are a library of programs that produce graphical output in response to user-supplied programs and data.

These modules will help you learn more about the LLUs. Note that many of the following documents are now available in PostScript *and* HTML format:

- *NCAR Graphics Fundamentals*
- *NCAR Graphics Contouring and Mapping Tutorial*
- User's Guide for NCAR GKS-0A Graphics
- *Areas, A Package that Solves Certain Practical Problems in Topology*
- *Autograph, A Package of Routines to Draw X/Y Plots*
- *CGM, Graphcap, and Fontcap Supplement*
- *Conpack, A Contouring Package*
- *Dashpack, A Software Package for Drawing Dashed Lines*
- *Ezmap, A Map-Drawing Package*
- *Gflash, A Graphics Instruction Manipulation Package*
- *Gridal, A Package to Draw Backgrounds for X/Y Plots*
- *Isosurface, A Package of Routines used to Draw Isosurfaces*
- *Labelbar, A Package of Routines to Draw Labelled Bars*
- *Plotchar, A Character- Plotting Package*
- *Polypack, A Package of Routines to Manipulate Polygons*
- *Stitle, A Package to Create Scrolled Movie Titles*
- *Seter, A Centralized Error-Handling Package for NCAR Graphics*
- *Softfill, A Software Fill Package*
- *Streamlines, A Field Flow Visualization Utility*
- *Tdpack, A Three-Dimensional Plotting Utility*

- *Vectors, A Vector Field Plotting Utility*
- *Wmap, A Package for Producing Daily Weather Maps and Plotting Station Model Data*