GRASS Development Team

GRASS 5.0 Programmer's Manual

Geographic Resources Analysis Support System

30th January 2004

Edited by

Markus Neteler

Member of **GRASS Development Team**

ITC-irst

Istituto per la Ricerca Scientifica e Tecnologica

Via Sommarive, 18

38050 Povo (Trento), Italy

*

GMS Laboratory

University of Illinois-Champaign, Urbana, Illinois

Center of Applied Spatial Research

Baylor University, Waco, Texas

*

30th January 2004, Draft Version

*

Based on preliminary programming notes on GRASS 5

written by Olga Waupotitsch and Michael Shapiro (CERL),
Bill Brown (GMSL) and Darrel McCauley (Purdue)

and the former

GRASS 4.2 Programmer's Manual

edited by Steve Clamons, Bruce Byars (Baylor University)
and basically written by
Michael Shapiro, James Westervelt, Dave Gerdes, Majorie Larson, and
Kenneth R. Brownfield (CERL)

ABSTRACT

GRASS (Geographical Resources Analysis Support System) is a comprehensive GIS with raster, topological vector, image processing, and graphics production functionality. This manual introduces the reader to the *Geographic Resources Analysis Support System* version 5.0 from the programming perspective. Design theory, system support libraries, system maintenance, and system enhancement are all presented. Standard GRASS 4.x conventions are still used in much of the code. This work is part of ongoing research being performed by the GRASS Development Team coordinated at ITC-irst, Trento, Italy), a worldwide programmer's team (see below), the GMS Laboratory at University of Illinois-Champaign (U.S.A.) and the Center of Applied Geographic and Spatial Research at Baylor University (U.S.A.). GRASS module authors are cited within their module's source code and the contributed manual pages.

30th January 2004

The development of GRASS software and this manual is kindly supported by Intevation GmbH, Osnabrück, Germany, who provide the GRASS CVS repository.

European Headquarters: http://grass.itc.it

Unites States Headquarters: http://www3.baylor.edu/grass/

Foreword

This manual represents documentation for the third revision to the Geographic Resources Analysis Support System (GRASS) Geographic Information System (GIS) with version 4.x being replaced with 5.0.

This work was originally performed by the Environmental Division (EN) of the U.S. Army Construction Engineering Research Laboratory (USACERL). In August, 1997, GRASS development was taken up by the GRASS Development Team at Baylor University. From Summer 1999 to Summer 2001 the GRASS project was coordinated at University of Hannover, Germany, since Summer 2001 at ITC-irst, Trento, Italy.

Original Authors of the GRASS 4.x Programmer's Manual are Michael Shapiro, James Westervelt, Dave Gerdes, Majorie Larson, and Kenneth R. Brownfield. It is upon their work that this is based, and we wish for full acknowledgement to go to them for their efforts. Dr. James Westervelt has provided valuable insight into GRASS for this project. Dr. Robert Lozar of USA-CERL has also been instrumental in the release of the GRASS 4.2 manual.

The upgrade to GRASS 5 programming API is based on comprehensive notes written in 1995 by Olga Waupotitsch and Michael Shapiro (CERL), Bill Brown (GMSL) and Darrel McCauley (Purdue University). Their documents have been written in HTML and were merged into this manual. Further core designers of GRASS 5 libraries and modules have been Roman Waupotitsch, James Westervelt, David Gerdes, Helena Mitasova, Jaro Hofierka and Lubos Mitas.

A rather complete list of GRASS programmers can be found online:

http://grass.itc.it/grasscredits.html

http://www3.baylor.edu/grass/grasscredits.html

Please notify us in case of contributors missing in this list.

Other Related Materials in the GRASS 5.0 Series

GRASS 5.0 Command Reference

NOTE: This manual is far from being completely updated. Please send your useful comments to Markus Neteler (neteler@itc.it).

GRASS 5 Core Team Members

(Status: 10/2000)

Roger Bivand (Norway), Radim Blazek (Czechia), Bill Brown (U.S.A.), Huidae Cho (South Korea), David D. Gray (U.K.), Jaro Hofierka (Slovak), Justin Hickey (Thailand), John Huddleston (U.S.A.), Bill Hughes (U.S.A.), Andreas Lange (Germany), Pierre de Mouveaux (France), Lubos Mitas (U.S.A.), Helena Mitasova (U.S.A.), Eric G. Miller (U.S.A.), Eric Mitchell (U.S.A.), Markus Neteler (Germany), Bernhard Reiter (Germany), Alexander Shevlakov (Russia), Frank Warmerdam (U.S.A.), Michel Wurtz (France), Lisa Zygo (U.S.A.)

Get latest list here:

http://freegis.org/cgi-bin/viewcvs.cgi/~checkout~/grass/AUTHORS

Book status and History

Note: This page will disappear when the book is finished.

This page reflects the current status of the "GRASS 5.0 Programmer's Manual" and needs to be updated regularly by book authors.

```
 [$Id: status.tex,v 1.23 2001/04/05 17:20:32 markus Exp $]

Current status:

Markus Neteler 4/2001:
* added HTML web pages, converted to Latex:
  - FP added, but needs to be merged furtherly within existing text
  - datetime added, but function definitions missing (?, ask Bill Brown)
  - sites API added
  - g3d API added
* comments are in [] like [GRASS 5: ...] Here further updates
  are required.
* added small PROJ4/GRASS API

Contributions from
 David D. Gray  (vector)
 Eric G. Miller (sites)


File list:

progmangrass50.tex -> head of document
progmangrass50.sty -> layout parameters, \Gfunc and Gprog environment are
                      defined here

chapter1.tex: "Introduction"
 - latex updated

chapter2.tex: "Development Guidelines"
 - latex updated

chapter3.tex: "Multilevel"
 - latex updated

chapter4.tex: "Database Structure"
 - latex updated
 - added 121 projections

chapter5.tex: "Raster Maps"
 - latex updated

chapter6.tex: "Vector Maps"
 - latex updated
 - 64 bit support needs to be included (Bill Hughes?)

chapter7.tex: "Point Data: Site List Files"
 - latex updated
```

```
chapter8.tex: "Image Data: Groups"
 - latex updated

chapter9.tex: "Region and Mask"
 - latex updated
 - is window/region terminology clear?

chapter10.tex: "Environment Variables"
 - latex updated
 - several new variables missing

chapter11.tex: "Compiling and Installing GRASS Modules"
 - Auto-conf is not yet explained

chapter12a.tex: "GIS Library 1"
 - latex updated
   Reference for function definition in Latex!!

chapter12b.tex: "GIS Library 2"
 - latex updated for new 5.x functions

chapter12c.tex: "GIS Library 3"
 - latex updated
 - added new parser functionality (implemented by Huidae Cho
                                        <hdcho@geni.knu.ac.kr>)
 - added unix sockets (Eric G. Miller)

chapter13.tex: "Vector Library"
 - latex updated

chapter14.tex: "Imagery Library"
 - latex updated

chapter15.tex: "Raster Graphics Library"
 - latex updated

chapter16.tex: "Display Graphics Library"
 - latex updated
 - added new functions:
    * D_set_dig_name(name)
    *     sets the name of the dig file currently displayed
    *
    * D_get_dig_name(name)
    *     returns the name of the dig file currently displayed

    implemented by Huidae Cho <hdcho@geni.knu.ac.kr>

chapter17.tex: "Lock Library"
 - latex updated

chapter18.tex: "Rowio Library"
 - latex updated

chapter19.tex: "Segment Library"
 - latex updated

chapter20.tex: "Vask Library"
 - latex updated
```

```
proj_datum.tex: "coordinate conversion library"
  - latex updated
  - added supported proj list
  - added PROJ4/GRASS API

grid3d.tex: "GRID3D voxel format"
 - latex updated

chapter21.tex: "DateTime Library"
 - latex updated
  %% Original Chapter 21 is outdated and belongs to GRASS 4.0/v.digit2.
  % replaced by DateTime description

gsurf.tex: "gsurf Library for OpenGL programming"
- latex updated

gui.tex: "tcltkgrass and XML/Python GUI programming"
- added

chapter22.tex: "Digitizer/Mouse/Trackball Files (.dgt)"

gmath.tex: "Numerical math interface to LAPACK/BLAS"
- added, latex o.k.

chapter23.tex: "Writing a Graphics Driver"
chapter24.tex: "Writing a Paint Driver"
chapter25.tex: "Writing GRASS Shell Scripts"
chapter26.tex: "GRASS CVS repository"
appendix.tex:
 - index available now, generated by makeindex

---------------
Contributions:

 - David D Gray <ddgray@armadce.demon.co.uk>:
    G__write_cats returns 1 on successful completion
      - affects at least: G_write_cats, G_write_vector_cats
   Fixed.

   Numerical math interface to LAPACK/BLAS
   added

 - Andreas Lange <Andreas.Lange@Rhein-Main.de>:
    CC-doc.tex completely written

 - Eric G. Miller:
    G_readsites_xyz
```

TODO list - Errata

G_tokenize() G_number_of_tokens() and G_free_tokens() are missing
(see libes/gis/token.c)

From: Glynn Clements <glynn.clements@virgin.net>
Date: Mon, 30 Apr 2001 00:16:23 +0100

1. XDRIVER now supports the RGB_RASTER operation natively; if it's
using a TrueColor or DirectColor visual, it uses logical operations to
convert the data (tested on a 5:6:5 display). No Colormaps,
lookup-tables or similar.

2. libdisplay contains some new functions for RGB raster operations:

```
D_draw_raster_RGB
D_draw_d_raster_RGB
D_draw_f_raster_RGB
D_draw_c_raster_RGB
D_draw_cell_RGB
D_cell_draw_setup_RGB
D_raster_of_type_RGB
D_set_colors_RGB
```

These are all more or less analogous to the corresponding functions
without the _RGB suffix, the main difference being that they take
three sets of raster data instead of one.

These functions all use the RGB_RASTER operation, so there are no
colour tables involved. The CELL/FCELL/DCELL values are converted to
bytes using the appropriate channel[1] from the specified colour table
(typically the one from the layer).

[1] I.e. the red components are used for the red layer, etc. If the
layers have suitable grey-scale colour tables, that'll work.

Date: Sat, 16 Dec 2000 14:03:43 -0700
From: "William L. Baker" <BakerWL@uwyo.edu>
Subject: [GRASS5] Programming manual or GIS library corrections?

Hello,

Am working on revision of r.le and not very knowledgeable about
the grass5 revisions in general, so please excuse if I miss something
obvious.

I think the following are just little mistakes in the programming manual:
1. G_zero_raster_row is listed on p. 143 of latest (Nov.) revision of
programming manual, but this does not seem to be the correct name.
It seems that it is G_zero_raster_buf.

2. G_read_fp_range on p. 156. I think the correct order of the parameters

```
is: (char *name, char *mapset, struct FPRange).  The manual lists
struct FPRange first.

3. p. 160 lists G_quant_truncate twice, but I think the second case
should be G_quant_round as that one works OK in my program, and
the description mentions rounding.

The following may be either errors in the programming manual or maybe
the GIS library?
1. G_read_raster_range is listed on p. 156, but does not work at all.
The old G_read_range works fine for CELL, and G_read_fp_range
works for float and double.

2. G_get_colors_min_max() is listed on p. 153, but does not work at all.
This one seems useful and I don't know what the substitute would be?

Bill Baker
Univ. of Wyoming

----------------------------------------

Modified Files:
        parser.c
Log Message:
added xml output of command parameters when issuing flag
--interface-description

chapter12c.tex:\subsection{Parser Routines}

add to gui.tex chapter
-------------------------------------------------
Add:

"description"
 see examples in all raster modules:

[...]
        module = G_define_module();
        module->description =
                "Finds the average of values in a cover map within "
                "areas assigned the same category value in a "
                "user-specified base map.";
[... parm definition]

-------------------------------------------------
Markus/David mailed:
> Maybe we
> can change this to common GIS vocabulary for GRASS 5.1?
> There was a sort of discussion on this recently.
> Maybe
>     5.0              5.1
>  category number -> index
>  category label  -> attribute

Yes. I think that hits the nail on the head.

-------------------------------------------------
R_pad_list();
R_pad_error();
```

```
R_pad_select();
R_pad_get_item();
R_pad_freelist();
R_pad_list_items();
R_pad_append_item();
R_pad_delete_item();
R_pad_set_item();
R_pad_create();
R_pad_current();
R_pad_delete();
R_pad_invent();

see d.save
diese Funktionen sind in src/libes/raster unter item_*.c, pad_*.c,
lists.c und perror.c.

-----------------------------------------------------

Add info about
XDriver/fifos etc.

(compare html/drivers.html)

-----------------------------------------------------

Add in grid3d.tex:
G_find_grid3 (name, mapset)

src/libes/g3d/find_grid3.c
```

Contents

1 Introduction

1.1 Background

The Geographic Resources Analysis Support System (GRASS) is a geographic information system (GIS) originally designed and developed by researchers at the U.S. Army Construction Engineering Research Laboratory (USACERL) and now supported and enhanced by the GRASS Development Team headquartered at ITC-irst, Trento (Italy) and Baylor University, Waco (U.S.A.). GRASS provides software capabilities suitable for organizing, portraying and analyzing digital spatial data.

Since the first release of GRASS software in 1985, the number of users and applications has rapidly grown. Because GRASS is distributed with source code und GNU General Public License, user sites (including many government organizations, educational institutions, and private firms) are able to customize and enhance GRASS to meet their own requirements. While researchers at ITC-irst, University of Illinois-Champaign and Baylor University maintain and support GRASS with worldwide contributions, as well as develop and organize new versions of GRASS for release, programmers at numerous sites work directly with GRASS source code.

The release of GRASS 5 under GNU General Public License (GPL) in October 1999 protects the various authors from misuse of their developments, especially in other proprietary systems. For the general user the open source model offers full insights into the system. Users can analyse the methods internally used, understand their functionality, modify methods to their purpose, error check and, in case required, correct or update methods. The speed to fix problems is usually much higher than in commercial systems. GRASS 5 is quite stable now and offers many new features comparing to GRASS 4.x. Another general purpose of the open-source release under GPL is the opportunity for users to implement their own ideas or to suggest modifications which could be implemented by everyone familar with programming. Currently GRASS 5 is in the top-ten list of biggest open-source programs available (http://www.codecatalog.com).

1.2 Objective

Those who work with GRASS source code need detailed information on the structure and organization of the software, and on procedures and standards for programming and documentation. The objective of this manual is to provide the necessary information for programmers to understand and enhance GRASS software.

1.3 Approach

GRASS software is continuously updated and improved. In the past, software enhancements have been developed at various sites, and submitted to USACERL to be shared with other sites and included in future releases of GRASS. Since CERL announced that it would not develop any more GRASS releases, the GRASS Development Team at ITC-irst and Baylor University have taken over development, support, and enhancement of the current GRASS version. Version 5.0 is currently the latest release, and is built largely on the GRASS 4.x source, with the major enhancement in raster floating point support, the new sites format, the datetime functions and the new GRID3D raster volume format being incorporation of contributed modules and codes.

With each new release of GRASS, more and more sites have begun working directly with GRASS source code. Sites are encouraged to use standard procedures in development of new GRASS capabilities. Sites that develop GRASS software are encouraged to learn and use GRASS programming libraries, and to use standard procedures for coding, commenting and documenting software. The use of GRASS libraries and conventions will:

1. Eliminate duplication of functions that already exist in GRASS libraries;
2. Increase the capability of multiple sites to share enhancements;
3. Reduce problems in adapting contributed GRASS capabilities to new data structures and new versions of GRASS software;
4. Provide some common elements (such as documentation and user interfaces) for users who use code contributed from multiple sites, and reduce the learning curve associated with each contributed capability.

The first GRASS Programmer's Manual was developed for GRASS 2.0 (released in 1987). The GRASS Programmer's Reference Manual for GRASS 3.0 (released in 1988) was completely rewritten due to the numerous and fundamental changes made in GRASS 3.0. The GRASS 4.1 Programmer's Manual was published in 1993 to reflect further code changes. The GRASS 4.2 Programmer's Manual was an update of the 4.1 manual and published in 1998.

Because much of GRASS has remained consistent from 3.0 to 4.0 and 4.1 USACERL researchers elected to upgrade the 3.0 Programmer's Manual to reflect the changes that have turned GRASS 3.0 into GRASS 4.0. This GRASS 5.0 manual is build on top of GRASS 4.2 manual incorporating the fundamental changes and new functions from GRASS 5.0.

The approach used in the development of this manual involves a systematic effort to describe GRASS development guidelines, user interfaces, data structures, programming libraries and peripheral drivers. Since it is based on the GRASS 4.x Programmer's Manuals, users should already be familiar with the conventions used here.

1.4 Scope

Information in this manual is valid for GRASS version 5.0, first time released in Spring 1999. As changes are made to GRASS libraries, data structures, and user interfaces, elements in this manual will require updating. Plans to perform updates, and the availability of these updates, will be announced on the GRASS web sites, in the GRASS mailing list and other GRASS information forums.

1.5 Mode of Technology Transfer

Federal organizations provide distribution and support services for GRASS within their own agencies, and several educational institutions and private firms also provide distribution, training and support services for GRASS. Current information on the status and availability of services for GRASS can be obtained from the GRASS Development Team located at either the ITC-irst, Italy, or the Baylor University, Waco, Texas[1] .

This manual should prove to be a valuable resource facilitating GRASS software development efforts at the numerous government agency, educational institutions and private firms that now use GRASS and plan to modify, enhance or customize the software. Sites that develop new analytical capabilities or peripheral drivers for GRASS are encouraged to share their products with others in the GRASS/GIS user community. To facilitate this sharing process among user, support and development sites, several forums have been established. These include the following:

The GRASS Information Center,

An annual GRASS/GIS User Group Meeting,

GRASS Internet sites with an electronic mailing lists and software retrieval forum.

The **GRASS Information Center** maintains: (1) a set of publications on GRASS and GRASS-related items, (2) updated information on locations that distribute and support GRASS software and on training courses for GRASS, (3) the mailing list for user discussion, and (4) updated information on the status of GRASS user group meetings and software releases.

The annual **GRASS / GIS User Group Meeting** is hosted by one of the member agencies of the Coordinating Committee. Papers, demonstrations, and discussion panels present GRASS applications and software development issues. The meeting provides opportunities for current and potential users to share and demonstrate new GRASS software.

GRASSNET is an electronic mail forum that provides a mechanism through which GRASS user and development sites can exchange messages. It can be reached via Internet.[2]

[1] See *1.6 GRASS Information Center (p. 6)* for phone numbers and mail addresses
[2] http://grass.itc.it/support.html

1.6 GRASS Information Center

Sites wishing to contribute code to GRASS GIS, or wanting to participate in any of these GRASS/GIS user community forums, should contact one of the GRASS Information Centers, either at ITC-irst or Baylor University at:

GRASS Development Team Institute of Physical Geography and Landscape Ecology ITC-irst MPA/SSI Via Sommarive, 18 38050 Trento (Povo), Italy email: neteler@itc.it http://grass.itc.it	GRASS Development Team Center for Applied Geographic and Spatial Research Baylor University P.O. Box 97351 Waco, Texas 76798-7351 email: grass@baylor.edu http://www.baylor.edu/~grass

2 Development Guidelines

GRASS continues its development with several key objectives as a guide. The programmer should be aware of these and strive to write code that blends well with existing capabilities. All objectives are based on an understanding of the needs of the end users of GRASS.

2.1 Intended GRASS Audience

GRASS is a general purpose geographic information system. Its intended users are regional land planners, ecologists, geologists, geographers, archeologists, and landscape architects. Used to evaluate broad land use suitability, it is ideal for siting large projects, managing parks, forest, and range land, and evaluating impacts over wide areas. These users are generally NOT equipped to write modules or design a system. In many cases they have never used a computer or even a keyboard.

REGIONAL PLANNING TOOL – GRASS is designed for planning at the county, park, forest, or range level. It is suitable for planning at a macro scale where the land uses are larger than 30 meters (or so, depending on the database resolution). As yet, no GRASS tools exist for the modeling and simulation of traffic, electrical, water, and sewage infrastructure loads, or for the precise positioning of urban structures.

UTM REFERENCED – To facilitate area calculations, a planimetric projection was desired for initial GRASS development. Funding was provided through Army military installations which were familiar with the Universal Transverse Mercator (UTM) projection. Due to these factors, GRASS developed around the UTM coordinate system. The UTM projection allows GRASS to assume equal area cells anywhere in the database. It also makes distance calculations simple and straightforward.

LATITUDE-LONGITUDE REFERENCING – It has been recognized that the UTM projection has limitations that make it awkward if not impossible to use for regions that span two (or more) UTM zones. Significant capabilities have been added to support latitude-longitude referenced data bases that will support analyses over large regions as well global analysis. However, the development is incomplete, especially on the vector side [GRASS 5 ????]. The programmer will find some routines in the libraries which are specifically designed to support this projection.

OTHER PROJECTIONS – Due to the spreading of GRASS usage around the globe it was required to introduce further projections and coordinate systems into GRASS GIS. GRASS 5.0 is coming along with 121 projections and supporting all important systems worldwide. [see Appendix of supported projections]

INTERACTIVE – GRASS has a strong interactive component. Its multilevel design allows users to work either at a very user friendly level, at a more flexible command level, or at a programming level. Beside the GRASS command line several graphical user interfaces (GUI) are available.

GRAPHIC ORIENTED – Many of the functions can be accompanied by graphic output results.

FOR NONPROGRAMMER – Users of GRASS are often first-time users of a GIS. To this end, it is important that the programmer take the extra time to provide on-line help, clear prompts, and user tutorials.

INEXPENSIVE – GRASS can run on personalcomputers in the under-$2,000 range. Higher-cost equipment should be necessary only for providing faster analyses, and more disk and memory space. It might be required for data-intense analysis. The software itself is freely available under GNU General Public License.

PORTABLE – This system is intended to be as portable as possible. Groups interested in GRASS resoundingly stated that portability is the number one concern, ranking firmly above speed and user friendliness. GRASS code must run on a wide variety of hardware configurations. GRASS 5.0 was tested in 32bit environment as well as on 64bit platforms. The "autoconfigure" tool allows to guess system specific parameters automatically for the compilation of the GRASS source code package.

2.2 Programming Standards

Programming is done within the following guidelines.

UNIX ORIENTED – Primarily for the purpose of portability, GRASS will continue its development under the UNIX operating system environment. Programmers should write code in ANSI-C style [GRASS 5: is that correct??]. Optionally it is intended to fully compile GRASS on WINDOWS-based platforms.

C LANGUAGE – All code is written in the C programming language. Some Fortran 77 code has occasionally been adopted into the system, but problems with portability, efficiency, and

legibility have resulted in most Fortran modules being rewritten in C.

FUNCTION LEVELS – GRASS is designed within a functional level scheme. Each level is designed to perform particular functions. Programming must be done within this scheme.

Briefly, these levels are as follows:

Specialized Interface Level – The new and occasional user would work at this level. It is expected that specialized models, natural language interfaces, graphic pop-up menu front-ends, and fancier menus will be developed in the future. GRASS modules developed at this level may be specifically designed for one hardware arrangement. [GRASS 5: ??]

Command Level – This is the level most used. Using the user's login shell, GRASS commands are made available through internal modification of the **PATH** variable. Help and on line manual commands are available as well as the graphical user interface "tcltkgrass".

In version 2.0, GRASS modules included both user interface and module function capabilities and were highly interactive. GRASS 3.0 introduced complementary command-line versions of these functions in which the information required by the module was provided by the user on the command line or in the standard input stream (with no prompting). This provided the advanced user greater flexibility and the system analyst a high-level GIS programming language in concert with other UNIX utilities. However, this resulted in a doubling of the number of commands: one for the interactive form, another for the command-line form.

Since GRASS 4.1 the interactive and command-line versions of a module have been "merged" into a single module (as far as the user is concerned). This merging should be understood by programmers developing new code. It is described in *11 Compiling and Installing GRASS Modules (p. 69)*. A standard command-line interface has been developed to complement the existing interactive interface, and an attempt has been made to standardize the command names.

Programming Level – For even greater flexibility in the application of GRASS, a user has the opportunity to module GRASS functions in the C language. The main restrictions here are that the programmer is to use the existing GRASS function libraries to the greatest extent possible.

Library Level – Work at the library level should be done with the cooperation and approval of one group. At this writing, that group is the GRASS programming staff from GRASS Development Team and worldwide contributors. The most critical functions are those that manipulate data. It is believed that these functions will be more permanent than the database structure. Though the database structure may change, these functions (and the programming environment) will not. Code management is centralized in CVS system (see *31 GRASS CVS repository (p. 431)*).

2.3 Documentation Standards

GRASS is a system under terms of GNU General Public License. While such systems are inexpensive to new sites wishing to adopt them, costs incurred in putting up the system, modifying the code, and understanding the product can be very high. To minimize these costs, GRASS modules shall be thoroughly documented at several levels.

Source code – The source code for the functions should be accompanied by liberal amounts of descriptive variables, algorithm explanations, and function descriptions.

On-line help – Brief help/information will be available for the new user of a module.

Online manual – Manual entries in the style of the UNIX manual entries will also be available to the user. Additionally these manual entries are published online in HTML format.

Tutorial – The tools that are more involved or difficult to use shall be accompanied by tutorial documents which teach a user how to use the code. These have been written in nroff/troff using the *ms* macro package [1] or in LaTeX format. Final documents have been kept separate from the GRASS directories, though it is suggested that they appear with appropriate "makefiles" under $GISBASE/tutorials.[2] [GRASS 5: ???]

[1] This package, invoked with the -ms option to nroff, is documented in section 7 of its UNIX manual.
[2] $GISBASE is the directory where GRASS is installed. See *10.1 UNIX Environment (p. 65)* for details.

3 Multilevel

As introduced in the previous section, the overall GRASS design incorporates several levels:

- Specialized Interfaces
- Command Level
- Programming Level
- Library Level

Each level is associated with a different type of user interface.

3.1 General User

The general GRASS user is someone with a skill in some resource area (e.g., planning, biology, agronomy, forestry, etc.) in which GRASS can be used to support spatial analysis. Such users have no significant computer skills, know nothing of UNIX, and may struggle with the learning curve for GRASS. Such users should select a **Specialized Interface**, if available, where they are guided through the GRASS system or a specific application in a friendly way. Programs written at this level may take many forms in the future. The promise of a natural language capability may take form here. Current success with graphic menu systems in other applications will lead to pleasant graphic screens with pull-down menus. Interfaces developed at this level (and this level only) may be hardware specific. GRASS may take the form of a voice-activated system with fancy AI capabilities on one machine, while it is driven by a pull-down menu system which is also tightly interfaced to an RDBMS on another [GRASS 5 ???]. All versions, however, will rely heavily on the consistent commands available at the **Command Level**. It is anticipated that specialized analysis models using little or no user input will be developed shortly, making use of UNIX shell scripts and **Command Level** programs. These models will be written by system analysts and will require no knowledge of C programming. Until improvements in speed and cost of hardware and flexibility of software are made available, most general users of GRASS will interface the system through the **Command Level**.

The **Command Level** requires some knowledge of UNIX. The user starts up the GRASS tools individually through the UNIX shell (commonly Bash, Bourne or Csh). Once a GRASS tool is started, the user either enters a very friendly and interactive environment or provides information to the tool in the form of arguments on the command line. Users are **not** prompted through graphics. Prompting is restricted to written interaction.

3.2 GRASS Programmer

The GRASS programmer, using an array of programming libraries, writes interactive tools and command line tools. Programmers must keep in mind that **Special Interfaces** tools will be:

a. Written for the occasional user;

b. Verbose in their prompting;

c. Accompanied by plenty of help; and

d. Give the user few options.

The programmer also writes **Command Level** tools. These:

a. Can run in batch (background) mode;

b. Take input from the command line, standard input, or a file;

c. Can run from a shell; and

d. Operate with a standard interface.

GRASS programmers should keep the following design goals in mind:

a. Consistent user interface;

b. Consistent database interface;

c. Functional consistency;

d. Installation consistency; and

e. Code portability.

As much as possible, interaction with the user (e.g., prompting for database files, or full screen input prompting) must not vary in style from module to module. All GRASS modules must access the database in a standard manner. Functional mechanisms (such as automatic resampling into the current region and masking of raster data) which are independent of the particular algorithm must be incorporated in most GRASS programs. Users must be able to install GRASS (data, programs, and source code) in a consistent manner. Finally, GRASS modules must compile and run on most (if not all) versions of UNIX. To achieve these goals, all programming must adhere to the following guidelines:

Use C language – This language is quite standard, ensuring very good portability. All of the GRASS system libraries are written in C. With very few exceptions, GRASS modules are also

written in C. While UNIX machines offer a Fortran 77 compiler, experience has shown that F77 code is not as portable or predictable when moved between machines. Existing Fortran code has occasionally been adopted, but programmers often prefer to rewrite the code in C.

Use Bourne shell – GRASS also makes use of the UNIX command interpreter to implement various function scripts, such as menu front-ends to a suite of related functions, or application macros combining GRASS command level tools and UNIX utilities. Portability requires that these scripts be written using the Bourne Shell (/bin/sh) and *no other*. See *30 Writing GRASS Shell Scripts (p. 427)*.

Do not access data directly – The GRASS database is **NOT** guaranteed to retain its existing organization and structure. These have changed in the past; however, the library function calls to the data have remained more consistent over time. Plans do exist to significantly change the data organization. While the programmer should be aware of the data capabilities and limitations, it should not be necessary to open and read data files directly.

Use GRASS Compilation Procedures – GRASS code is compiled using a special procedure 1 which is a front-end to the UNIX *make* utility. This procedure allows the programmer to construct a file with *make* rules containing instructions for making the binary executables, manual and help entries, and other items from the directory's contents. However, there are no hardcoded references to other GRASS programs, libraries, or directories. Variables defining these items are provided by the procedure and are used instead. This allows the compilation and installation process to remain identical from system to system. This procedure is described in detail in *11 Compiling and Installing GRASS Modules (p. 69)*.

Use GRASS libraries – Use of the existing GRASS programming libraries speeds up programming efforts. While user and data interface may make up a large part of a new module, the programmer, using existing library functions, can concentrate primarily on the analysis algorithms of the new tool. Such modules will maintain a consistency in data access and (more importantly) a degree of consistency in the user interface. The libraries are listed briefly below.

GIS Library. This library contains all of the routines necessary to read and write the GRASS raster data layers and their support files. General GRASS database access routines are also part of this library. A standardized method to prompt the user for map names is available. The library also provides some general purpose tools like memory allocation, string analysis, etc. Nearly all GRASS modules use routines from this library. See *12 GIS Library (p. 79)*.

Vector Library. GRASS was developed primarily as a raster map analysis and display system, but got vector capabilities. The principal uses of GRASS vector files are to generate raster maps and to plot base maps on top of raster map displays. Further developments like vector based network analysis are in progress.

However, it is anticipated that additional analysis and data import capabilities will be added to the vector database. Many vector formats exist in the GIS world, but GRASS

has chosen to implement its own internal vector format. The format is a variant of arc-node. The **Vector Library** provides access to the GRASS vector *database*. See *13 Vector Library (p. 219)*.

Segment Library. For modules that need random access to an entire map layer, the segment library provides an efficient paging scheme for raster maps. While virtual memory operating systems perform paging, this library sometimes provides better control and efficiency of paging for raster maps. See *19 Segment Library (p. 277)*.

Vask Library. This screen-oriented user interface is widely used in the GRASS programs. It provides the programmer with a simple means for displaying a particular screen layout, with defined fields where the user is prompted for answers. The user, using the carriage return (or line-feed), Cursor keys and Ctrl-k keys, moves from prompt to prompt, filling an answer into each field. When the ESC-RETURN keys are struck, the answers are provided to the module for analysis. Users have found this interface pleasant and consistent. See *20 Vask Library (p. 283)*.

Graphics Libraries. Graphics design has been a difficult issue in GRASS development. To ensure portability and competitive bidding, GRASS has been designed with graphics flexibility in mind. This has meant restricting graphics to a minimal set of graphics primitives, which generally do not make full use of the graphics capabilities on all GRASS machines. Two libraries, **displaylib** and **rasterlib**, are involved in generating graphics. The **rasterlib** contains the primitive graphics commands used by GRASS. At run time, modules using this library communicate (through fifo files) with another module which translates the graphics commands into graphics on the desired device. Each time the module runs, it may be talking to a different graphics device. Functions available in the **rasterlib** include color setting and choosing, line drawing, mouse access (with three types of cursor), raster drawing operations, and text drawing. Generally, this library is used in conjunction with the **displaylib.** The **displaylib** provides graphics frame management routines, coordinate conversion capabilities, and raster data to raster *graphic conversions*. See *16 Display Graphics Library (p. 255)* and *15 Raster Graphics Library (p. 243)*.

3.3 Driver Programmer

GRASS modules are written to be portable. To this end, a tremendous amount of modularity is designed into the system. Throughout its development, GRASS modules have become increasingly specialized. The original monolithic approach continues to fragment into ever smaller pieces. Smaller pieces will allow future developers and users ev er more variability in the mixing of the tools. This modularity has been manifested in the graphics design. A graphics-oriented tool connects, at run time, to a graphics driver (or translator) module. This separate process understands the standard graphics commands generated by the GRASS tool, and makes the appropriate graphics calls to a particular graphics device. Each graphics device available to a user is accompanied by a driver module, and each module understands the graphics calls of the application module. Porting of GRASS to a new system primarily means the development of one new graphics driver. See *28 Writing a Graphics Driver (p. 409)*.

Those sites using the digitizing software of GRASS must also provide driver routines for their digitizer. These routines, unlike the above graphics calls, are compiled directly into *the digitizing modules*. See *27 Digitizer/Mouse/Trackball Files (.dgt) (p. 395)*. Similarly, GRASS sites may wish to write code to support different hardcopy color printers (inkjet, thermal, etc.). See *29 Writing a Paint Driver (p. 415)*.

3.4 GRASS System Designer

GRASS system design has mostly been done at one location: USACERL. However, in August, 1997, the GRASS Development Team at Baylor University took over development of GRASS. From 1998 onwards the University of Hannover and worldwide programmers and groups joined the team. One, and only one group must be responsible for the design of the system at the database and fundamental library level. As the software is released under terms of GNU General Public License, sites are free to do their own work. However, the strength of future GRASS releases depends on cooperation and sharing of software. Therefore, it is strongly encouraged that **database design and database library development be fully coordinated with GRASS Development Team headquartered at University of Hannover and Baylor University.** The team members can be reached over internet, mail, FAX and phone.

4 Database Structure

This chapter presents the programmer interested in developing new applications with an explanation of the structure of the GRASS databases, as implemented under the UNIX operating system.

4.1 Programming Interface

GRASS Programmers are provided with the *GIS Library*, which interfaces with the GRASS database. It is described in detail in *12 GIS Library (p. 79)*. Programmers should use this library to the fullest extent possible. In fact, a programmer will find that use of the library will make knowledge of the database structure almost unnecessary. GRASS modules are not written with specific database names or directories hardcoded into them. The user is allowed to select the database or change it at will. The database name, its location within the UNIX file system, and other related database information are stored as variables in a hidden file in the user's home directory. GRASS modules access this information via routines in the *12 GIS Library (p. 79)*. The variables that specify the database are described briefly below; see *10 Environment Variables (p. 65)* for more details about these and other environment variables.

Note. These GRASS environment variables may also be cast into the UNIX environment to make them accessible for shell scripts.[1] In the discussion below, these variables will appear preceded by a dollar sign ($). However, C programs should not access the GRASS environment variables using the UNIX getenv( ) since they do not originate in the UNIX environment. GIS Library routines, such as *G_getenv*, must be used instead.

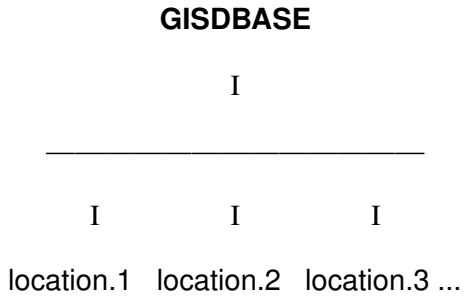
4.2 GISDBASE

The database for GRASS makes use of the UNIX hierarchical directory structure. The top level directory is known as GISDBASE. Users specify this directory when entering GRASS. The full name of this directory is contained in the UNIX environment variable $GISDBASE, and is returned by library routine *G_gisdbase.*

[1] using g.gisenv; see *30 Writing GRASS Shell Scripts (p. 427)*

4.3 Locations

Subdirectories under the GISDBASE are known as locations. Locations are independent databases. Users select a location when entering GRASS. All database queries and modifications are made to this location only. It is not possible to simultaneously access multiple locations. The currently selected location is contained in the environment variable $LOCATION_NAME, and is returned by the library routine *G_location.*



When users select a location, they are actually selecting one of the location directories.

Note. GISDBASE may be changed to the parent directory of other sets of locations, notably on other system hard disks for database management purposes. Note that GRASS modules will only work within one location under one GISDBASE directory in a given GRASS session.

4.4 Mapsets

Subdirectories under any location are known as mapsets. Users select a mapset when entering GRASS. New mapsets can be created during the selection step. The selected mapset is known as the current mapset. It is named in the environment variable $MAPSET and returned by *G_mapset.*

Modifications to the database can only be made in the current mapset. Users may only select (and thus modify) a mapset that they own (i.e., have created). However, data in all mapsets for a given location can be read by anyone (unless prevented by UNIX file *permissions*). See 4.7

Database Access Rules (p. 22) for more details. When users select a mapset, they are actually selecting one of the mapset directories.

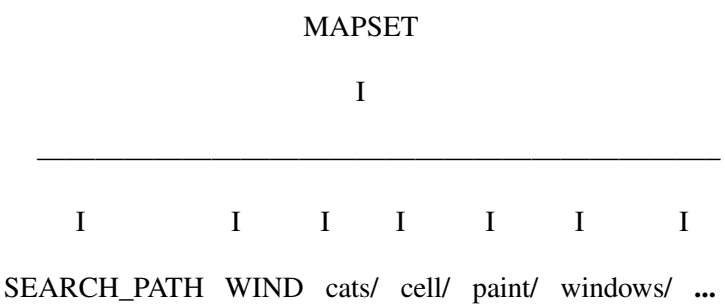
Note. The full UNIX directory name for the current mapset is

$GISDBASE/$LOCATION_NAME/$MAPSET and is returned by the library routine G_location_path.

Note. Each location will have a special mapset called PERMANENT that contains non volatile data for the location that all users will use. However, it also contains some information about the location itself that is not found in other mapsets. See *4.6 Permanent Mapset (p. 21)*.

4.5 Mapset Structure

Mapsets will contain *files* and subdirectories, known as database *elements.* In the diagram below, the elements are indicated by a trailing /.



4.5.1 Mapset Files

The following is a list of some of the mapset files used by GRASS programs:

files	function
GROUP	current imagery group
SEARCH_PATH	mapset search path
WIND	current region

This list may grow as GRASS grows. The GROUP file records the current imagery group selected by the user, and is used only by imagery functions. The other two files are fundamental to all of GRASS. These are WIND and SEARCH_PATH.

WIND is the current region.[2] This file is created when the mapset is created and is modified by the *g.region* command. The contents of WIND are returned by *G_get_window*. See *9.1 Region (p. 61)* for a discussion of the GRASS region.

SEARCH_PATH contains the *mapset search path*. This file is created and modified by the *g.mapsets* command. It contains a list of mapsets to be used for finding database files. When users enter a database file name without specifying a specific mapset, the mapsets in this search path are searched to find the file. Library routines that look for database files follow and use the mapset search path. See *4.7.1 Mapset Search Path (p. 22)* for more information about the mapset search path.

4.5.2 Elements

Subdirectories under a mapset are the database *elements*. Elements are not created when the mapset is created, but are created dynamically when referenced by the application programs.[3] Mapset data reside in files under these elements.

The dynamic creation of database elements makes adding new database elements simple since no reconfiguration of existing mapsets is required. However, the programmer must be aware of the database elements already used by currently existing modules when creating new elements. Furthermore, as development occurs outside USACERL, guidelines must be developed for introducing new element names to avoid using the same element for two div erse purposes.

Programmers using shell scripts must exercise care. It is not safe to assume that a mapset has all, or any, database elements (especially brand new mapsets). Certain GRASS commands automatically create the element when it is referenced (e.g., *g.ask*). In general, however, elements are only created when a new file is to be created in the element. It is wise to explicitly check for the existence of database elements.

Here is the list of some of the elements used by GRASS modules written at USACERL:

element	function
cell	binary raster file (INT)
fcell	binary raster file (FLOAT/DOUBLE)
cellhd	header files for raster maps
cats	category information for raster maps
colr	color table for raster maps
colr2	secondary color tables for raster maps
cell_misc	miscellaneous raster map support files (projection etc.)
hist	history information for raster maps
dig	binary vector data
dig_ascii	ascii vector data
dig_att	vector attribute support

[2] Under GRASS 3.0 this was called the database ”window”. However, the term ”window” has many meanings. For clarity this term has been replaced by the term ”region”. The database files and programming interfaces, however, hav e not been renamed. Thus WIND now contains the current region.

[3] See *12.5.7 Creating and Opening a New Database File (p. 89)*.

dig_cats	vector category label support
dig_plus	vector topology support
reg	digitizer point registration
arc	ascii ARC/INFO ungenerate files (for data exchange)
bdlg	binary dlg files (for data exchange)
dlg	ascii dlg files (for data exchange)
dxf	ascii DXF files (for data exchange)
camera	camera specification files used by *i.ortho.photo*
icons	icon files used by *p.map*
paint	label and comment files used by *p.map*
group	imagery group support data
site_lists	site lists for *sites* related modules
windows	predefined regions
COMBINE	*r.combine* scripts
WEIGHT	*r.weight* scripts

Note. The mapset database elements can be simple directory names (e.g., cats, colr) or multilevel directory names (e.g., paint/labels, group/xyz/subgroup/abc). The library rou-tines that create the element will create the top level directory and all subdirectories as well.

4.6 Permanent Mapset

Each location must have a PERMANENT mapset. This mapset not only contains original raster and vector files that must not be modified, but also several special files that are only found in this mapset. The files MYNAME and DEFAULT_WIND and are never modified by GRASS software. The

MYNAME contains a single line descriptive name for the location. This name is returned by the routine *G_myname.*

DEFAULT_WIND contains the default region for the location. The contents of this file are returned by *G_get_default_window.* This file is used to initialize the WIND file when GRASS creates a new mapset. and can be used by the user as a reference region at any time.

PROJ_INFO contains detailed projection parameters.

PROJ_UNITS contains the projection units information.

4.7 Database Access Rules

GRASS database access is controlled at the mapset level. There are three simple rules:

1. A user can select a mapset as the *current* mapset only if the user is the owner *of the mapset directory* (see *4.4 Mapsets (p. 18)*).

2. GRASS will create or modify files only in the current mapset.

3. Files in all mapsets may be read by anyone (see *4.7.1 Mapset Search Path (p. 22)*) unless prohibited by normal UNIX file permissions (see *4.7.2 UNIX File Permissions (p. 22)*).

4.7.1 Mapset Search Path

When users specify a new data file, there is no ambiguity about the mapset in which to create the file: it is created in the current mapset. However, when users specify an existing data file, the database must be searched to find the file. For example, if the user wants to display the ”soils” raster map, the system looks in the various database mapsets for a raster file named ”soils.” The user controls which mapsets are searched by setting the *mapset search path*, which is simply a list of mapsets. Each mapset is examined in turn, and the first ”soils” raster file found is the one that is displayed. Thus users can access data from other users’ mapsets through the choice of the search path.

Users set the search path using the *g.mapsets* command.

Note. If there were more than one ”soils” file, the mapset search mechanism returns the first one found. If the user wishes to override the search path, then a specific mapset could be specified along with the file name. For example, the user could request that ”soils@PERMANENT” be displayed.

4.7.2 UNIX File Permissions

GRASS creates all files with read/write permission enabled for the owner and read only for everyone else; directories are created with read/write/search permission enabled for the owner and read/search only for everyone else.[4] This implies that all users can read anyone else’s data files. Read access to all files in a mapset can be controlled by removing (or adding) the read and search permissions on the mapset directory itself using the GRASS *g.access* command, without adversely affecting GRASS programs. If read and search permissions are removed, then no other user will be able to read any file in your mapset.

[4] This means -rw-r–r–for files, and drwxr-xr-x for directories. It is accomplished by setting the umask to 022 in all GRASS programs.

Warning. Since the PERMANENT mapset contains global database information, all users must have read and search access to the PERMANENT mapset directory[5]. Do not remove the read and search permissions from PERMANENT.

4.8 Supported Projections

GRASS Projection software is based on PROJ4 from USGS.

New site since 2000: http://www.remotesensing.org/proj/

```
ll -- Lat/Lon
utm -- Universe Transverse Mercator
stp -- State Plane
aea -- Albers Equal Area
lcc -- Lambert Conformal Conic
merc -- Mercator
tmerc -- Transverse Mercator
leac -- Lambert Equal Area Conic
laea -- Lambert Azimuthal Equal Area
aeqd -- Azimuthal Equidistant
airy -- Airy
aitoff -- Aitoff
alsk -- Mod. Stererographics of Alaska
apian -- Apian Globular I
august -- August Epicycloidal
bacon -- Bacon Globular
bipc -- Bipolar conic of western hemisphere
boggs -- Boggs Eumorphic
bonne -- Bonne (Werner lat_1=90)
cass -- Cassini
cc -- Central Cylindrical
cea -- Equal Area Cylindrical
chamb -- Chamberlin Trimetric
collg -- Collignon
crast -- Craster Parabolic (Putnins P4)
denoy -- Denoyer Semi-Elliptical
eck1 -- Eckert I
eck2 -- Eckert II
eck3 -- Eckert III
eck4 -- Eckert IV
eck5 -- Eckert V
eck6 -- Eckert VI
eqc -- Equidistant Cylindrical (Plate Caree)
eqdc -- Equidistant Conic
euler -- Euler
fahey -- Fahey
fouc -- Foucaut
fouc_s -- Foucaut Sinusoidal
gall -- Gall (Gall Stereographic)
gins8 -- Ginsburg VIII (TsNIIGAiK)
```

[5] PERMANENT has the DEFAULT_WIND and MYNAME files. This is a minor design flaw. Global database information should be kept in the database, but not in any of the mapsets. All mapsets could then be treated equally.

```
gn_sinu -- General Sinusoidal Series
gnom -- Gnomonic
goode -- Goode Homolosine
gs48 -- Mod. Stererographics of 48 U.S.
gs50 -- Mod. Stererographics of 50 U.S.
hammer -- Hammer & Eckert-Greifendorff
hatano -- Hatano Asymmetrical Equal Area
imw_p -- International Map of the World Polyconic
kav5 -- Kavraisky V
kav7 -- Kavraisky VII
labrd -- Laborde
lagrng -- Lagrange
larr -- Larrivee
lask -- Laskowski
lee_os -- Lee Oblated Stereographic
loxim -- Loximuthal
lsat -- Space oblique for LANDSAT
mbt_s -- McBryde-Thomas Flat-Polar Sine (No. 1)
mbt_fps -- McBryde-Thomas Flat-Pole Sine (No. 2)
mbtfpp -- McBride-Thomas Flat-Polar Parabolic
mbtfpq -- McBryde-Thomas Flat-Polar Quartic
mbtfps -- McBryde-Thomas Flat-Polar Sinusoidal
mil_os -- Miller Oblated Stereographic
mill -- Miller Cylindrical
mpoly -- Modified Polyconic
moll -- Mollweide
murd1 -- Murdoch I
murd2 -- Murdoch II
murd3 -- Murdoch III
nell -- Nell
nell_h -- Nell-Hammer
nicol -- Nicolosi Globular
nsper -- Near-sided perspective
nzmg -- New Zealand Map Grid
ob_tran -- General Oblique Transformation
ocea -- Oblique Cylindrical Equal Area
oea -- Oblated Equal Area
omerc -- Oblique Mercator
ortel -- Ortelius Oval
ortho -- Orthographic
pconic -- Perspective Conic
poly -- Polyconic (American)
putp1 -- Putnins P1
putp2 -- Putnins P2
putp3 -- Putnins P3
putp3p -- Putnins P3'
putp4p -- Putnins P4'
putp5 -- Putnins P5
putp5p -- Putnins P5'
putp6 -- Putnins P6
putp6p -- Putnins P6'
qua_aut -- Quartic Authalic
robin -- Robinson
rpoly -- Rectangular Polyconic
sinu -- Sinusoidal (Sanson-Flamsteed)
somerc -- Swiss. Obl. Mercator
stere -- Stereographic
tcc -- Transverse Central Cylindrical
tcea -- Transverse Cylindrical Equal Area
```

```
tissot -- Tissot
tpeqd -- Two Point Equidistant
tpers -- Tilted perspective
ups -- Universal Polar Stereographic
urm5 -- Urmaev V
urmfps -- Urmaev Flat-Polar Sinusoidal
vandg -- van der Grinten (I)
vandg2 -- van der Grinten II
vandg3 -- van der Grinten III
vandg4 -- van der Grinten IV
vitk1 -- Vitkovsky I
wag1 -- Wagner I (Kavraisky VI)
wag2 -- Wagner II
wag3 -- Wagner III
wag4 -- Wagner IV
wag5 -- Wagner V
wag6 -- Wagner VI
wag7 -- Wagner VII
weren -- Werenskiold I
wink1 -- Winkel I
wink2 -- Winkel II
wintri -- Winkel Tripel
```

5 Raster Maps

This chapter provides an explanation of how raster map layers are accommodated in the GRASS database[1].

5.1 What is a Raster Map Layer?

GRASS raster map layers can be conceptualized, by the GRASS programmer as well as the user, as representing information from a paper map, a satellite image, or a map resulting from the interpretation of other maps. Usually the information in a map layer is related by a common theme (e.g., soils, or landcover, or roads, etc.). GRASS raster data are stored as a matrix of *grid cells*. Each grid cell covers a known, rectangular (generally square) patch of land. Each raster cell is assigned a single integer attribute value called the *category* number. For example, assume the land cover map covers a state park. The grid cell in the upper-left corner of the map is category 2 (which may represent prairie); the next grid cell to the east is category 3 (for forest); and so on.

land cover

2	3	3	3	4	4
2	2	3	3	4	4
2	2	3	3	4	4
1	2	3	3	3	4
1	1	1	3	3	4
1	1	3	3	4	4

1 = urban 3 = forest

2 = prairie 4 = wetlands

In addition to the raster file itself, there are a number of support files for each raster map layer. The files which comprise a raster map layer all have the same name, but each resides in a different database directory under the mapset. These database directories are:

[1] The descriptions given here are for GRASS 5.x data formats only. Previous formats, still supported by GRASS but no longer generated, are described in documents from earlier releases of GRASS.

directory	function
cell	binary raster (cell) files (int-format)
fcell	binary raster (cell) files (FP-format)
cellhd	raster header files
cats	raster map category information
colr	raster map color tables
colr2	alternate raster map color tables
hist	raster map history information
cell_misc	miscellaneous raster map support information

For example, a raster map named *soils* would have the files *cell/soils, cellhd/soils, colr/soils, cats/soils, etc.*

Note. Database directories are also known as database *elements.* See *4.4 Mapsets (p. 18)* for a description of database elements.

Note. *GIS Library* routines which read and write raster files are described in *12.9 Raster File Processing (p. 105).*

5.2 Raster File Format

The programmer should think of the raster data file as a two-dimensional matrix (i.e., an array of rows and columns) of integer values. Each grid cell is stored in the file as one to four 8-bit bytes of data. An N*x*M raster file will contain N rows, each row containing M columns of cells.

The physical structure of a raster file can take one of 3 formats: uncompressed, compressed, or reclassed.

Uncompressed format. The uncompressed raster file actually looks like an N*x*M matrix. Each byte (or set of bytes for multibyte data) represents a cell of the raster map layer. The physical size of the file, in bytes, will be *rows*cols*bytes-per-cell.*

Compressed format. The compressed format uses a run-length encoding schema to reduce the amount of disk required to store the raster file. Run-length encoding means that sequences of the same data value are stored as a single byte repeat count followed by a data value. If the data is single byte data, then each pair is 2 bytes. If the data is 2 byte data, then each pair is 3 bytes, etc. (see Multibyte data format 29 below). The rows are encoded independently; the number of bytes per cell is constant within a row, but may vary from row to row. Also if run-length encoding results in a larger row, then the row is stored non-run-length encoded. And finally, since each row may have a different length, there is an index to each row stored at the beginning of the file.

Reclass layers. Reclass map layers do not contain any data, but are references to another map layer along with a schema to reclassify the categories of the referenced map layer. The reclass file itself contains no useful information. The reclass information is stored in the raster header file.

Multibyte data format. When the data values in the raster file require more than one byte, they are stored in *big-endian* format[2], which is to say as a base 256 number with the most significant digit first.

Examples:

cell value	base 256	stored as
868	= 3*256 + 100	3 | 100
137,304	= 2*256^2 + 24*256 + 88	2 | 24 | 88
174,058,106 =	10*256^3 + 95*256^2 + 234*256 + 122	10 | 95 | 234 | 122

Negative values are stored as a signed quantity, i.e., with the highest bit set to 1[3]:

cell value	base 256	stored as
1	= -(1)	1 | 0 | 0 | 0 | 1
868	= -(3*256 + 100)	1 | 0 | 0 | 3 | 100
137,304	= -(2*256^2 + 24*256 + 88)	1 | 0 | 2 | 24 | 88
174,058,106	= -(10*256^3 + 95*256^2 + 234*256 + 122)	1 | 10 | 95 | 234 | 122

All data values in a given row are stored using the same number of bytes. This means that if the value 868, which uses 2 bytes, occurred in a row that uses 3 bytes to represent the largest data value, 868 would be stored as 0 | 3 | 100.

Also, one row may only require 2 bytes to store its data values, another 4 bytes, and yet another 1 byte. The rows are stored independently and would be stored using 2 bytes, 4 bytes, and 1 byte respectively.

File portability. The multibyte format described above is (except possibly for negative values) machine independent. If raster files are to be moved to a machine with a different cpu, or accessed using a heterogeneous network file system (NFS), the following guidelines should be kept in mind. All *5.0 format* [4] raster files will transfer between machines, with two restrictions: (1) if the file contains negative values, the size of an integer on the two machines must be the same; and (2) the size of the file must be within the seek capability of the lseek() call[5].

5.3 Raster Header Format

The raster file itself has no information about how many rows and columns of data it contains, or which part of the earth the layer covers. This information is in the raster header file. The

[2]The fact that the values are stored *big-endian* should not be construed to mean that the machine architecture must also be *big-endian.* The programs which read raster files perform the necessary arithmetic to construct the value. They do NOT assume anything about byte ordering in the cpu.

[3]This means that the value is stored using as many bytes as required by an integer on the machine (usually 4).

[4]The raster file format did not change from 3.0 to 4.x.

[5]This usually means that the size of a long integer on the two machines is the same.

format of the raster header depends on whether the map layer is a regular map layer or a reclass layer.

Note. *GIS Library* routines which read and write the raster header file are described in *12.10.1 Raster Header File (p. 113)*.

5.3.1 Regular Format

The regular raster header contains the information describing the physical characteristics of the raster file. The raster header has the following fields:

raster header

proj:	1
zone:	18
north:	4660000
south:	4570000
east :	770000
west :	710000
e-w resol:	50
n-s resol:	100
rows:	900
cols:	1200
format :	0
compressed:	0

proj, zone The projection field specifies the type of cartographic projection[6]:

- 0 is unreferenced x,y (imagery data)
- 1 is UTM
- 2 is State Plane
- 3 is Latitude-Longitude
- 99 other (see the PERMANENT/PROJ_INFO file for detailed definition)

Others may be added in the future. The *zone* field is the projection zone. In the example above, the projection is UTM, the zone is 18.

north, south, east, west The geographic boundaries of the raster file are described by the *north, south, east*, and *west* fields. These values describe the lines which bound the map at its edges. **These lines do NOT pass through the center of the grid cells at the edge of the map, but along the edge of the map itself.**

[6] State Plane is not yet fully supported in GRASS and Latitude-Longitude is still under development [GRASS 5:???]

n-s resol, e-w resol The fields *e-w resol* and *n-s resol* describe the size of each grid cell in the map layer in physical measurement units (e.g., meters in a UTM database). They are also called the grid cell resolution. The *n-s resol* is the length of a grid cell from north to south. The *e-w resol* is the length of a grid cell from east to west. As can be noted, cells need not be square.

rows, cols The fields *rows* and *cols* describe the number of rows and columns in the raster matrix.

format The *format* field describes how many bytes per cell are required to represent the raster data. 0 means 1 byte, 1 means 2 bytes, etc. The value -1 indicates that it is a floating point raster map which is stored in fcell/ subdirectory instead if cell/ subdirectory.

compressed The *compressed* field indicates whether the raster file is in compressed format or not: 1 means it is compressed and 0 means it is not. If this field is missing, then the raster file was produced prior to GRASS 3.0 and the compression indication is encoded in the raster file itself.

Note. If the rows and columns of the raster matrix are not stored in the raster header, they are computed from the geographic boundaries as follows:

rows = (north - south) / (ns resol)

cols = (east - west) / (ew resol)

If the rows and columns of the raster matrix are stored in the raster header, the resolution values are computed from the geographic boundaries as follows:

ns resol = (north - south) / (rows)
ew resol = (east - west) / (cols)

5.3.2 Reclass Format

If the raster file is a reclass file, the raster header does not have the information mentioned above. It will have the name of the referenced raster file and the category reclassification table.

reclass header

reclass	
name:	county
mapset:	PERMANENT
#5	*first category in reclass*
1	*5 is reclassified to 1*
0	*6 is reclassified to 0*
1	*7 is reclassified to 1*
0	*8 is reclassified to 0*
2	*9 is reclassified to 2*

In this case, the library routines will use this information to open the referenced raster file in place of the reclass file and convert the raster data according to the reclass scheme. Also, the referenced raster header is used as the raster header.

5.4 Raster Category File Format

The category file contains the largest category value which occurs in the data, a title for the map layer, an automatic label generation capability, and a one line label for each category.

category file

```
# 5 categories
title for map layer
<automatic label format>
<automatic label parameters>
0:no data
1:description for category 1
2:description for category 2
3:description for category 3
5:description for category 5
```

The number which follows the # on the first line is the largest category value in the raster file. The next line is a title for the map layer. The next two lines are used for automatic label generation. They are used to create labels for categories which do not have explicit labels. (The automatic label capability is not normally used in most map layers, in which case the *format* line is a blank line and the *parameters* line is: 0.0 0.0 0.0 0.0.) Category labels follow on the remaining lines. The format is *cat : label.*

The first four lines of the file are required. The remaining lines need only appear if categories are to be labeled.

Note. *GIS Library* routines which read and write the raster category file are described in *12.10.2 Raster Category File (p. 114)*.

GRASS 5.x category table file

- The Categories table file is now upgraded to support floating-point ranges of values for fp maps. So instead of

  ```
  cat1:description
  ```

  the format is the same for integer maps, but in addition the new format for floating point map is supported.

  ```
  val1:val2:description
  ```

  or

  ```
  val1:description
  ```

  where `value1` and `value2` are floating-point numbers.

- All support functions (e.g. range, cell_stats, colors) should assume that the data has embedded NULL values.

5.5 Raster Color Table Format

The GRASS raster color tables and associated programming interface have undergone a fairly major revision to resolve problems presented by raster maps that have a large range of data values. The previous design[7] used arrays to store a color for each data value between the minimum and maximum values in the raster map. This array structure was also reflected in the format of the color table file–each color stored as a single line in the color file. Because GRASS raster maps can have data values in the range ± 2147483647[8] this method of storing color information is clearly untenable.

Since GRASS 4.x the above problem is solved by representing color tables as linear ramps for intervals of data values. Colors are specified (and stored) for the endpoints of each interval. Colors for values between endpoints are not stored but are computed using a linear interpolation scheme.

The following is an example 4.x color file:

4.x color table file

```
% 1387 1801
```

[7] GRASS 3.x
[8] These values are for 32-bit architectures.

1387:255:85:85 **1456:170:170:0**	colors for categories 1387-1456
1456:170:170:0 **1525:85:255:85**	colors for categories 1456-1525
1525:85:255:85 **1594:0:170:170**	colors for categories 15251594
1594:0:170:170 **1663:85:85:255**	colors for categories 1594-1663
1663:85:85:255 **1732:170:0:170**	colors for categories 1663-1732
1732:170:0:170 **1801:255:85:85**	colors for categories 1732-1801

The first line is a % character (to indicate that this is a 4.x format color file) and two numbers indicating the minimum and maximum data values which have colors. The rest of the file are the color descriptors. In this example, the minimum and maximum values are 1387 and 1801. Looking at the first color line, the color for category 1387 is red=255, green=85, blue=85; the color for category 1456 is red=170, green=170, blue=0.[9] The color for category 1400 is calculated from the colors for categories 1387 and 1456:

red= interpolate(255,170) = 239

green = interpolate(85,170) = 101

blue = interpolate(85,0) = 69

There are other formats which are simply variants of this format. For example, if the red, green, and blue intensities are all the same, then only the ”red” value appears. This next example defines a gray scale color table:

4.x color table file

% 1387 1801

1387:0 1801:255

Also, if the starting and ending categories are the same, only the first appears:

[9] The colors are represented as levels of red, green, and blue, where 0 represents the lowest intensity and 255 represents the highest intensity

4.x color table file

%1 6

1:34:179:112

2:233:110:15

3:127

4:43:135:33

5:70:7:52

6:93:210:163

Note. *GIS Library* routines which read and write the raster color table are described in *12.10.3 Raster Color Table (p. 116).*

GRASS 5.x color table file

- All support functions (e.g. range, cell_stats, colors) should assume that the data has embedded NULL values.

- The color table file is now upgraded to support floating-point ranges of values. So instead of

  ```
  cat1:red:grn:blu  cat2:red:grn:blu
  ```

  the format is now

  ```
  value1:red:grn:blu  value2:red:grn:blu
  ```

  where `value1` and `value2` are floating-point numbers. Also now color table file can contain entries of the form

  ```
  *:red:grn:blu
  ```

  default color (this sets the rgb for all values for which no explicit rgb values were defined in the color table)

  ```
  nv:red:grn:blu
  ```

  (this sets color for drawing `"no data"` cells

5.6 Raster History File Format

The history file contains historical information about the raster map: creator, date of creation, comments, etc. It is generated automatically along with the raster file. In most applications,

the programmer need not be concerned with the history file. Occasionally a module might put information into this file not known or readily available to the user, such as information about a satellite image: sun angles, dates, etc. The GRASS *r.info module* allows the user to view this information, and the *r.support module* allows the user to update it. It is the user's responsibility to maintain this file.

Note. *GIS Library* routines which read and write the raster history file are described in *12.10.3.4.1 Raster History File (p. 121).*

5.7 Raster Range File Format

The range file contains the minimum and maximum values which occur in a raster file. It is generated automatically for all new raster files. This file lives in the *cell_misc* element as "cell_misc/*name*/range" where *name* is the related raster file name. It contains one line with four integer values. These represent the minimum and maximum negative values, and the minimum and maximum positive values in the raster file. If there are no negative values, then the first pair of numbers will be zero. If there are no positive values, then the second pair of numbers will be zero.

Note. *GIS Library* routines which read and write the raster range file are described in *12.10.4 Raster Range File (p. 122).*

5.8 Raster Maps: Floating-Point / NULL support (draft, needs to be merged into tutorial!)

FP-AUTHORS:

Michael Shapiro
Olga Waupotitsch

5.8.1 Objectives

- Provide support for floating point values in raster maps in a cpu-independent format with little or no loss of precision.
- Provide an explicit NULL value, for both integer and floating point raster maps, that is distinct from any number (e.g. different than zero).

5.8.2 Design decisions

This subsection describes some of the design decisions that have been made so far.

5.8.2.1 Floating-point maps

- Sun's eXternal Data Representation (XDR) format and the vendor supplied XDR C-API will be used to read and write floating point numbers to disk.
- Floating-point data will be written to a file in a new `fcell` directory and an empty file created in the `cell` directory.
- for each operation on raster data that there will be one generic routine with 3 specific routines: G_something_raster (..., void *raster, RASTER_MAP_TYPE type, ...) G_something_c_raster (..., CELL *cell, ...) G_something_f_raster (..., FCELL *fcell, ...) G_something_d_raster (..., DCELL *dcell, ...) For null-related operations on raster data the functions will be: G_something_null_value (..., void *raster, RASTER_MAP_TYPE type, ...) G_something_c_null_value (..., CELL *cell, ...) G_something_f_null_value (..., FCELL *fcell, ...) G_something_d_null_value (..., DCELL *dcell, ...) For null-related operations on raster data the functions will
- When programs write floating-point maps and they do not explicitly specify the storage type, the storage type defaults to `float`, unless the user sets the Unix environment variable GRASS_FP_DOUBLE, which changes the default to `double`.
- Any map can be read either as floating-point or as integer, independently of the actual storage type, with the following rules for type conversion:
  - Conversion from `int` to `float` or to `double` will be by C-language promotion.
  - Conversion from `double` to `float` will be by C-language demotion (with subsequent loss of precision)
  - Conversion from `float` or `double` to `int` will be by a flexible on-the-fly *quantization*.
- Any map can be written using floating-point or integer buffers. Any of the C-language number types (`int`, `float`, `double`) can be used to write to the raster map, but if the type doesn't match the resultant storage type (which is determined when the map is opened), the following rules apply:
  - Conversion from `int` to `float` or to `double` will be by C-language promotion.
  - Conversion from `double` to `float` will be by by C-language demotion (with subsequent loss of precision)
  - Conversion from `float` or `double` to `int` will be by C-language truncation to integer.
- Allow storing floating-point either as 4-byte `floats` or 8-byte `doubles`. The user should have some control as to whether floating-point values are be written as `doubles` (greater precision at higher disk use) or `floats` (less precision as lower disk use).
- Provide an interface to determine the storage type (`int`, `float`, `double`)
- Provide two methodologies for accessing NULL values when reading maps:
  - Embedding the NULL value in the raster data buffers
  - Reading a buffer of NULL value booleans flags.

- All support functions (e.g. range, cell_stats, colors) should assume that the data has embedded NULL values.
- The color table file is now upgraded to support floating-point ranges of values. So instead of

  ```
  cat1:red:grn:blu  cat2:red:grn:blu
  ```

  the format is now

  ```
  value1:red:grn:blu  value2:red:grn:blu
  ```

  where `value1` and `value2` are floating-point numbers. Also now color table file can contain entries of the form

  ```
  *:red:grn:blu
  ```

  default color (this sets the rgb for all values for which no explicit rgb values were defined in the color table)

  ```
  nv:red:grn:blu
  ```

  (this sets color for drawing `"no data"` cells
- The Categories table file is now upgraded to support floating-point ranges of values for fp maps. So instead of

  ```
  cat1:description
  ```

  the format is the same for integer maps, but in addition the new format for floating point map is supported.

  ```
  val1:val2:description
  ```

  or

  ```
  val1:description
  ```

  where `value1` and `value2` are floating-point numbers.
- Support files needed for floating-point maps will be under the `cell_misc` directory:

  `cell_misc/<map>/f_format` This ascii file contains information about the format of the floating point data: float/double, compression, XDR indicator. For example:

  ```
  type: float
  byte_order: xdr
  lzw_compression_bits: 9
  ```

  This file is read and written using the Key_Value functions in the `GISLIB`. The *byte_order* and *lzw_compression* are write-only and are for documentation (and future upgrades). The *type* is one of `float` or `double` and indicates the storage type for the floating-point values in the map.

  `cell_misc/<map>/f_range` This binary file contains the range (minimum and maximum) of the floating point data written to the map.

  `cell_misc/<map>/f_quant` This ascii file contains rules specifying how to quantize the floating point values into integers. The format is, one rule per line:

```
value1:value2:cat1:cat2
```

where *value1* and *value2* are a range of floating point values, with * indicating (positive or negative) infinity; *cat1* and *cat2* are a range of integer values, with *cat2* optional. The rule is a linear interpolation from a floating-point value in the range *[value1,value2]* to an integer in the range *[cat1,cat2]*.

5.8.2.2 NULL-values

- The NULL values will not be written to the raster map itself; a zero will be written as a place holder. NULL value flags will be written as a separate file under the `cell_misc` directory as `cell_misc/<map>/null`. This file will be a bitmap with ones indicating that the corresponding cell contains valid data, and zeros indicating that the corresponding cell contains a NULL value.
- Internally within programs, the NULL value can either be processed as a special bit pattern embedded in the raster data, or as a separate char array of flags. The bit pattern used for integer is the largest positive integer. The bit pattern for floating point number is one (of many) NaN bit patterns. This bit pattern will never be written to the raster data file.
- For maps which do not have a NULL bitmap file, zeros are translated to NULL when the map is read. (NOTE: is this behaviour is not desirable, use G_get_map_row() the old `"read row"` function. It reads all 0's and nulls as 0's) See REF description for G_get_map_row() for more info.
- In addition to NULL values within the raster file, NULL values are be generated automatically by both the MASK and the region (if the region extends beyond the extent of the raster map).

6 Vector Maps

This chapter provides an explanation of how vector map layers are accommodated in the GRASS database.

6.1 What is a Vector Map Layer?

GRASS vector maps are stored in an *arc-node* representation, consisting of curves called *arcs.* An arc is stored as a series of x,y coordinate pairs.[1] The two endpoints of an arc are called *nodes.* Tw o consecutive x,y pairs define an arc *segment.*[2]

The arcs, either singly, or in combination with others, form higher level map features: *lines*[3] (e.g., roads or streams) or *areas* [4] (e.g., farms or forest stands). Arcs that form linear features are sometimes called *lines,* and arcs that outline areas are called *area edges or area lines.*[5]

Each map feature is assigned a single integer attribute value called the *category* number. For example, assume a vector file contains land cover information for a state park. One area may be assigned category 2 (perhaps representing prairie); another is assigned category 3 (for forest); and so on. Another vector file which contains road information may have some roads assigned category 1 (for paved roads); other roads may be assigned category 2 (for gravel roads); etc. See *5.1 What is a Raster Map Layer? (p. 27)* for more information about GRASS category values.

A vector map layer is stored in a number of data files. The files which comprise a single vector map layer all have the same name, but each resides in a different database directory under the mapset.[6] These database directories are:

directory	function
dig	binary arc file

[1] For this reason *arcs* are also called *vectors.*

[2] Arc *segments* are sometimes called *line-segments.*

[3] A *line* here does not mean a straight line between two points. It only means a linear feature.

[4] *Areas* are also called *polygons.* The GRASS vector format does not store the polygons explicitly. They are constructed by finding the particular *arcs* which form the polygon perimeter.

[5] Obviously, there is some confusion in the GIS vector terminology. This is partly due to use of terms that have a common meaning as well as a mathematical meaning. Vector terminology is a subject for much debate in the GIS world.

[6] Database directories are also called *elements.* See *4.4 Mapsets (p. 18)* for a description of database elements.

dig_ascii	ascii arc file
dig_att	vector category attribute file
dig_cats	vector category labels
dig_plus	vector index/pointer file
reg	digitizer registration points

For example, a map layer named *soils* would have the files *dig/soils, dig_att/soils, dig_plus/soils, dig_ascii/soils, dig_cats/soils, reg/soils, etc.*

paste vect.xfig diagram here [GRASS 5: missing since 1993]

Note. Vector files are also called *digit* files, since they are created and modified by the *GRASS digitizing module v.digit.*

Note. When referring to one of the vector map layer files, the directory name is used. For example, the file under the *dig* directory is called the *dig* file.

Note. Library routines which read and write vector files are described in *13 Vector Library (p. 219)*.

6.2 Ascii Arc File Format

The arc information is stored in a binary format in the *dig* file. The format of this file is reflected in the ascii representation stored in the *dig_ascii* file. It is the ascii version which is described here.[7]

The *dig_ascii* file has two sections: a header section, and a section containing the arcs.

6.2.1 Header Section

The header contains historical information, a description of the map, and its location in the universe. It consists of fourteen entries. Each entry has a label identifying the type of information, followed by the information. The format of the header is:

label	format	description

[7] The programs *v.import, v.in.ascii*, and *v.out.ascii* convert between the ascii and binary formats.

ORGANIZATION:	text (max 29 characters)*	organization that digitized the data
DIGIT DATE:	text (max 19 characters)*	date the data was digitized
DIGIT NAME:	text (max 19 characters)*	person who digitized the data
MAP NAME:	text (max 40 characters)*	title of the original source map
MAP DATE:	text (max 10 characters)*	date of the original source map
OTHER INFO:	text (max 72 characters)*	other comments about the map
MAP SCALE:	integer	scale of the original source map
ZONE:	integer	zone of the map (e.g., UTM zone)
WEST EDGE:	real number (double)	western edge of the entire map
EAST EDGE:	real number (double)	eastern edge of the entire map
SOUTH EDGE:	real number (double)	southern edge of the entire map
NORTH EDGE:	real number (double)	northern edge of the entire map
MAP THRESH:	real number (double)	digitizing resolution
VERTI:	(no data)	marks the end of the header section

** Currently, GRASS modules which read the header information are not tolerant of text fields which exceed these limits. If the limits are exceeded, the ascii to binary conversion will probably fail.*

++The edges of the map describe a region which should encompass all the data in the vector file.

+++The MAP THRESH is set by the v.digit module. If the data comes from outside GRASS, this field can be set to 0.0.

The labels start in column 1 and continue through column 14. Labels are uppercase, left justified, end with a colon, and blank padded to column 14. The information starts in column 15. For example:

ORGANIZATION: GRASS Development Team

DIGIT DATE: 03/18/99

DIGIT NAME: grass

MAP NAME: Urbana, IL.

MAP DATE: 1975

OTHER INFO: USGS sw/4 urbana 15' quad. N4000-W8807.5/7.5

MAP SCALE: 24000

ZONE: 16

WEST EDGE: 383000.00

EAST EDGE: 404000.00

SOUTH EDGE: 4429000.00

NORTH EDGE: 4456000.00

MAP THRESH: 0.00

VERTI:

6.2.2 Arc Section

The arc information appears in the second section of the *dig_ascii* file (following *VERTI:* which marks the end of the header section). Each arc consists of a description entry, followed by a series of coordinate pairs. The description specifies both the type of arc (*A* for area edge, or *L* for line[8]), and the number of points (coordinate pairs) in the arc. Then the points follow.

For example:

A 5

4434456.04 388142.16

4434446.65 388202.64

4434407.49 390524.38

4434107.06 390523.59

4433326.51 390526.48

L 3

4434862.31 392043.33

4434872.42 394662.14

4434871.44 398094.75

A 3

4454747.38 396579.60

4454722.69 393539.73

4454703.68 390786.90

In this example, the first arc is an area edge and has 5 points. The second arc is part of a linear feature and has 3 points. The third arc is another area edge and has 3 points.

[8] Other types may be added in the future.

The arc description has the letter *A* or *L* in the first column, followed by at least one space, and followed by the number of points.[9]

Point entries start with a space, and have at least one space between the two coordinate values.[10]

Note. The points are stored as **y,x** (i.e., north, east), which is the reverse of the way GRASS usually represents geographic coordinates.

Note. If the *v.digit module* has deleted an arc, the arc type will be represented using a lower case letter (i.e., *l* instead of *L, a* instead of *A*). Of course, this will only be manifest when a binary *dig* file with a deleted arc is converted to the ascii *dig_ascii* file.

6.3 Vector Category Attribute File

As was mentioned in *6.1 What is a Vector Map Layer? (p. 41)*, each feature in the vector map layer has a *category* number assigned to it. The category number for each map feature is not stored in the *dig* file itself, but in the *dig_att* file. The *dig_att* file is an ascii file that has multiple entries, each with the same format. Each entry refers to one map feature, and specifies the feature type (area or line), an x,y marker, and a category number.

For example:

A 389668.32 4433900.99 7

L 395103.96 4434881.19 2

In this example, an area feature is assigned category 7, and a linear feature is assigned category 2.

The x,y marker is used to find the map feature in the *dig* file. It must be located so that it uniquely identifies its related map feature. In particular, an area marker must be inside the area, and a line marker must be closer to its related line than to any other line (preferably on the line) and not at a node.

If multiple entries identify the same map feature, only one will be used (currently thee last entry).

A map feature which has no entry in this file is considered to be unlabeled. This means that during the vector to raster conversion (i.e., *v.to.rast*), unlabeled areas will convert as category zero, and unlabeled lines will be ignored.

The format of this file is rather strict, and is described in the following table:

columns	Data
1	Type of map feature (*A* or *L*)*

[9] This can be written with the Fortran format: *A1,1X,I4.*
[10] These can be written with the Fortran format: 2(1X,F12.2).

2-3	Spaces
4-15	Easting (x) of the marker, right justified
16-17	Spaces
18-29	Northing (y) of the marker, right justified
30-31	Spaces
32-39	Category number, right justified
40-49	Spaces
50	Newline †

* Other types, such as *point*, may be allowed in the future.

† UNIX text files are terminated with a newline. Therefore, each entry will appear as 49 characters. The entire file size should be a multiple of 50.

This format is required by modules which modify the vector map (e.g., *v.digit*). Programs which only read the vector map accept a looser format: the feature type must start in column 1; the items must be separated by at least one space; and the entries must be less than 50 characters. Also, the module *v.support* will convert the looser format to this stricter format.

Note. The marker is specified as **x,y** (i.e., east, north), which is the way GRASS usually represents geographic coordinates, but which is reverse of the way the arcs are stored in the dig_ascii file.

6.4 Vector Category Label File

Each category in the vector map layer may have a one-line description. These category labels are stored in the *dig_cats* file. The format of this file is identical to the raster category file described in *12.10.2 Raster Category File (p. 114)*, and the reader is referred to that section for details.

Note. The module *v.support* allows the user to enter and modify the vector category labels. The module *v.to.rast* copies the *dig_cats* file to the raster category file during the vector to raster conversion.

Note. Library routines which read and write the *dig_cats* file are described under *12.12.6 Vector Category File (p. 165)*.

6.5 Vector Index and Pointer File

The *dig_plus* file contains information that accelerates vector queries. It is created by the module *build.vect* (which is run by *v.digit* when a vector file is created or modified, and by *v.support*

at user request) from the data in the *dig* and *dig_att* files. For this reason, and since the internal structure of the *dig_plus* file is complex, the format of this file will not be described.

6.6 Digitizer Registration Points File

The *reg* file is an ascii file used by the *v.digit module* to store map registration control points. Each map registration point has one entry with the easting and northing of the map control point. For example:

383000.000000 4429000.000000

383000.000000 4456000.000000

404000.000000 4456000.000000

404000.000000 4429000.000000

Note. This file is used by *v.digit* only. It is *not* used by any other module in GRASS.

6.7 Vector Topology Rules

The following rules apply to the vector data:

1. Area edges should not cross each other (i.e., arcs which would cross must be split at their intersection to form distinct arcs), otherwise topology for the map may not be build correctly.

2. Arcs which share nodes must end at exactly the same points (i.e., must be *snapped* together). This is particularly important since nodes are not explicitly represented in the arc file, but only implicitly as endpoints of arcs.

3. Common boundaries should appear only once (i.e., should not be double digitized).

4. Areas must be explicitly closed. This means that it must be possible to complete each area by following one or more area edges that are connected by common nodes, and that such tracings result in closed areas.

5. It is recommended that area features and linear features be placed in separate layers. However if area features and linear features must appear in one layer, common boundaries should be digitized only once. An area edge that is also a line (e.g., a road which is also a field boundary), should be digitized as an area edge (i.e., arc type *A*) to complete the area. The area feature should be labeled as an area (i.e., feature type *A* in the *dig_att* file). Additionally, the common boundary arc itself (i.e., the area edge which is also a line) should be labeled as a line (i.e., feature type *L* in the *dig_att* file) to identify it as a linear feature.

Note that planar enforcement is not present on vector library level, and both line and area edge feature may be written to the vector so that cross themselves in the same file. To get correct topology for area edges, required above, cleaning modules must be applied on the vector file.

6.8 Importing Vector Files Into GRASS

The following files are required or recommended for importing vector files from other systems into GRASS:

dig_ascii

The *dig_ascii file*, described in *6.2 Ascii Arc File Format (p. 42)*, is required.

dig_att

The *dig_att* file, described in *6.3 Vector Category Attribute File (p. 45)*, is essentially required. While the *dig_ascii* file alone is sufficient for simple vector display, the *dig_att* file is required for vector to raster conversion, as well as more sophisticated vector query.

dig_cats

The dig_cats file, described in *6.4 Vector Category Label File (p. 46)*, while not required, allows map feature descriptions to be imported as well.

Note. The *dig_plus file*, described in *6.5 Vector Index and Pointer File (p. 46)*, is created by the GRASS module *v.import* when converting the *dig_ascii* file to the binary *dig* file.

7 Point Data: Site List Files

Chapter status: Needs further updates!

This chapter describes how point data is currently accommodated in the GRASS database.

7.1 What is a Site List?

Point data is currently stored in ascii files called *site lists* or *site files.* These files are used by the *sites* modules. The *site list* files were designed for use by these modules, but have since become the principal data structure for point data.

7.2 GRASS 5 Site File Format

Site files are ascii files stored under the **site_lists** database element.[1] The format of a site file is best explained by two examples:

name | sample

desc | sample site list

728220 | 5182440 | #27 %1.34 %23.13 @"pH 7.1"

727060 | 5181710 | #28 %2.32 %22.21 @"pH 6.8"

725500 | 5184000 | #29 %4.73 %17.34 @"pH 5.5"

719800 | 5187200 | #30 %3.65 %27.79 @"pH 6.2"

name | sample

desc | sample site list

728220 | 5182440 | 10| #27 %1.34 %23.13 @"string 1a" @"string 2a"

728220 | 5182440 | 20| #28 %1.52 %32.81 @"string 1b" @"string 2b"

[1] See *4.5.2 Elements (p. 20)* for an explanation of database elements.

728200 | 5182440 | 30| #29 %0.23 %43.54 @"string 1c" @"string 2c"

717060 | 5181710 | 10| #30 %2.32 %22.21 @"string 1d" @"string 2d"

[GRASS 5: date field!!!]

name This line contains the name of the site list file, and is printed on all the reports generated by the *s.menu* module. The word **name** must be all lower case letters.

It is permissible for this line to be missing, since the *s.menu module* will add a name record using the name of the site list file itself.

desc This line contains a description of the site list file, and is printed on all the reports generated by the *s.menu* module. The word **desc** must be all lower case letters.

It is also permissible for this line to be missing, in which case the site list will have no description.

points The remaining lines are *point* records. Each site is described by a *point* record.

The format for this record is:[2]

```
east|north[|dim]...|#cat %double [%double] @string [@string]
```

or specifying a date/time record:

```
east|north[|dim]...|time %double [%double]
```

The *east* and *north* fields represent the geographic coordinates (easting and northing) of the site.

The *description* field holds a single integer and is compulsory. Please read *12.13 Site List Processing (GRASS 5 Sites API) (p. 166)* for further details. The new sites format in GRASS 5.0 allows multiple dimensions and multiple attributes with strings and decimals support.

Examples:

```
name|soil
desc|
form|||%
labels|Easting|Northing|%No Label
3566177.5|5763062.5|%161.19 @clay
3566182.5|5763062.5|%161.19 @"sandy soil"
3566187.5|5763062.5|%160.53 @clay

name|time
desc|Example of using time as an attribute
time|Mon Apr 17 14:24:06 EST 1995
10.8|0|9.8|Fri Sep 13 10:00:00 1986 %31.4
11|5.5|9.9|Fri Sep 14 00:20:00 1985 %36.4
5.1|3.9|10|Fri Sep 15 00:00:30 1984 %28.4
```

[2] The pipe character ("|") is used to separate the dimension fields in the records, blank spaces are used to separate decimal and string descriptions.

This data has three dimensions (assume easting, northing, and elevation), five string attributes, and one decimal attribute.

comments Blank lines, and lines beginning with #, are accepted (and ignored).

7.3 Programming Interface to Site Files

The programming interface to the site list files is described in *12.13 Site List Processing (GRASS 5 Sites API) (p. 166)* and the programmer should refer to that section for details.

8 Image Data: Groups

Chapter status: Needs further updates!

This chapter provides an explanation of how imagery data are accommodated in the GRASS database.

8.1 Introduction

Remotely sensed images are captured for computer processing by satellite or airborne sensors by filtering radiation emanating from the image into various electromagnetic wavelength bands, converting the overall intensity for each band to digital format, and storing the values on computer compatible media such as magnetic tape. Color and color infrared photographs are optically scanned to convert the red, green, and blue wavelength bands in the photograph into a digital format as well.

The digital format used by image data is basically a raster format. GRASS imagery modules[1] which extract image data from magnetic tape extract the band data into raster cell files in a GRASS database. Each band becomes a separate cell file, with standard GRASS data layer support, and can be displayed and analyzed just like any other cell file. However, since the band files are extracted as individual cell files, it is necessary to have a mechanism to maintain a relationship between band files from the same image as well as cell files derived from the band files. The GRASS *group* database structure accomplishes this goal.

8.2 What is a Group?

The group is a database mechanism which provides the following:

1. A list of related cell files,
2. A place to store control points for image registration and rectification, and
3. A place to store spectral signatures, image statistics, etc., which are needed by image classification procedures.

[1] See *8.4 Imagery Modules (p. 58)* for a list of the major GRASS imagery modules.

8.2.1 A List of Cell Files

The essential feature of a group is that it has a list of cell files that belong in the group. These can be band data extracted from the same data tape, or cell files derived from the original band files.[2] Therefore, the group provides a convenient "handle" for related image data; i.e., referring to the *group* is equivalent to referring to all the band files at once.

8.2.2 Image Registration and Rectification

The group also provides a database mechanism for image registration and rectification. The band data extracted from tapes are usually unregistered data. This means that the GRASS software does not know the Earth coordinates for pixels in the image. The only coordinates known at the time of extraction are the columns and the rows relative to the way the data was stored on the tape.

Image *registration* is the process of associating Earth coordinates with pixels on the image. Image *rectification* is the process of converting the image files to the new coordinate system based on the registration.

Image registration is applied to a group, rather than to individual cell files. The user displays any of the cell files in a group on the graphics monitor and then marks control points on the image, assigning Earth coordinates to each control point. The control points are stored in the group, allowing all related group files to be registered in one step rather than individually.

Image rectification is applied to individual cell files, with the control points for the group used to control the rectification. The rectified cell files are placed into another database[3] known as the *target* database. Rectification can be applied to any or all of the cell files associated with a group.

8.2.3 Image Classification

Image classification methods process all or a subset of the band files as a unit. For example, a clustering algorithm generates spectral signatures which are then used by a maximum likelihood classifier or other algorithm to produce a landcover map.

Sometimes only a subset of the band files are used during image classification. The signatures must be associated only with the cell files actually used in the analysis. Therefore, within a

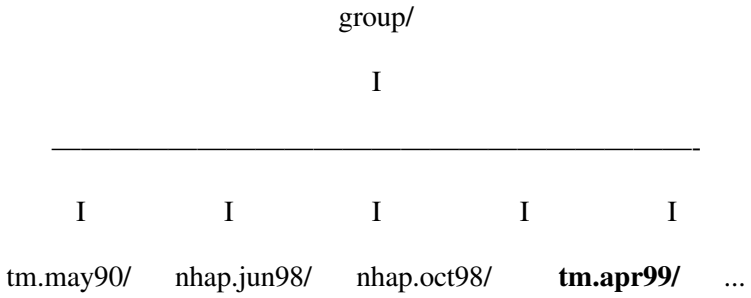
[2] Derived cell files can be the results of image classification procedures such as clustering and maximum likelihood, or band ratios formed using *r.mapcalc*, etc.

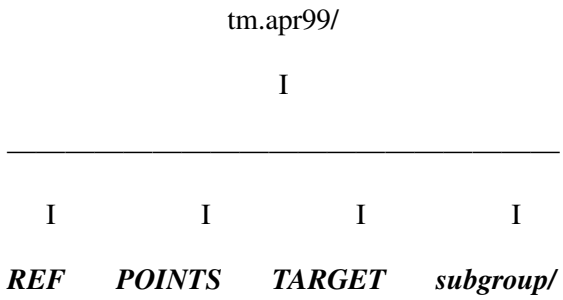
[3] Either a projected database, such as UTM, or an unregistered database (xy), if the image is being registered to another image.

group, *subgroups* can be formed which list only the band files to be "subgrouped" for classification purposes. The signatures are stored with the subgroup. Multiple subgroups can be created within a group, which allows different classifications to be run with different combinations of band files.

8.3 The Group Structure

Groups live in the GRASS database under the **group** database element.[4] The structure of a group can be seen in the following diagram. A trailing / indicates a directory.





In this example, the groups are named *tm.may90* , *nhap.jun98* , etc.[5] Note that each group is itself a directory. Each group contains some files (*REF* , *POINTS* ,and *TARGET*), and a subdirectory (*subgroup*).

8.3.1 The REF File

The REF file contains the list of cell files associated with the group. The format is illustrated below:

[4] See *4.5.2 Elements (p. 20)* for an explanation of database elements.
[5] The group names are chosen by the user.

```
tm.apr99.1 grass
tm.apr99.2 grass
tm.apr99.3 grass
tm.apr99.4 grass
tm.apr99.5 grass
tm.apr99.7 grass
```

Each line of this file contains the name and mapset of a cell file. In this case, there are six *cell files in the group: tm.apr99.1 , tm.apr99.2 , tm.apr99.3 , tm.apr99.4 , tm.apr99.5 and tm.apr99.7 without thermal tm.apr99.6* in mapset *grass.* (Presumably these are bands 1-5 and 7 from an April 99 Landsat Thematic Mapper image.)

8.3.2 The POINTS File

The POINTS file contains the image registration control points. This file is created and modified by the *i.points* module. Its format is illustrated below:

# image		target		status
# east	north	East	north	(1=ok)
#				
504.00	-2705.00	379145.30	4448504.56	1
458.00	-2713.00	378272.67	4448511.67	1
2285.80	-2296.00	415610.08	4450456.17	1
2397.00	-2564.00	417043.22	4444757.65	0
2158.00	-2944.00	411037.79	4438210.97	1
2148.00	-2913.00	410834.61	4438656.18	0
2288.80	-2336.20	415497.19	4449671.77	1

The lines which begin with # are comment lines. The first two columns of data (under *image*) are the column (i.e., *east*) and row (i.e., *north*[6]) of the registration control points as marked on the image. The next two columns (under *target*) are the *east* and *north* of the marked points in the target database coordinate system (in this case, a UTM database). The last column (under *status*) indicates whether or not the control point is well placed.[7] (If it is ok, then it will be used as a valid registration point. Otherwise, it is simply retained in the file, but not used.)

[6] Note that the row values are negative. This is because GRASS requires the northings to *increase* from south to north. Negative values accomplish this while preserving the row value. The true image row is the absolute value. [GRASS 5: ?????????]

[7] The user makes this decision in *i.points.*

8.3.3 The TARGET File

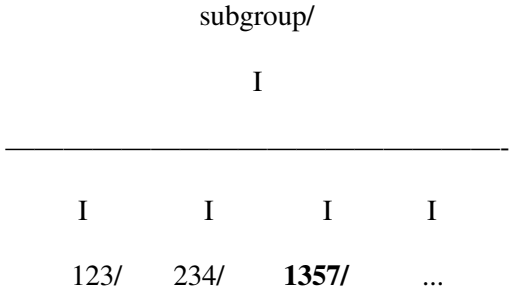
The TARGET file contains the name of the *target* database; i.e., the GRASS database mapset into which rectified cell files will be created. The TARGET file is written by *i.target* and has two lines:

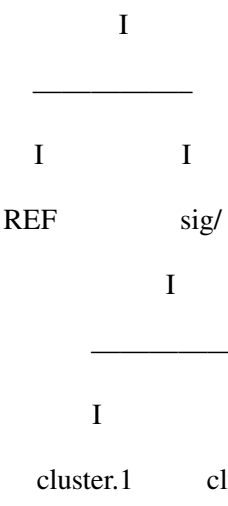
```
spearfish
grass
```

The first line is the GRASS location (in this case *spearfish*), and the second is a mapset within the location (in this case *grass*).

8.3.4 Subgroups

The subgroup directory under a group has the following structure:





In this example, the subgroups are named *123*, *234*, *1357*, etc.[8] Within each subgroup, there is a REF file and a *sig* directory. The REF file would list a subset of the cell files from the group. In this example, it could look like:

tm.apr99.1	grass
tm.apr99.3	grass
tm.apr99.5	grass
tm.apr99.7	grass

indicating that the subgroup is composed of bands 1, 3, 5, and 7 from the April 1999 TM scene. The files *cluster.1* and *cluster.2* [9] under the *sig* directory contain *spectral signature* information (i.e., statistics) for this combination of band files. The files were generated by different runs of the clustering module *i.cluster*.

8.4 Imagery Modules

The following is a list of some of the imagery modules in GRASS, with a brief description of what they do. Refer to the *GRASS 5 User's Reference Manual* for more details.

image extraction

i.tape.mss Landsat Multispectral Scanner data

i.tape.tm Landsat Thematic Mapper data

i.tape.spot SPOT Satellite data

i.tape.other other formats, such as scanned aerial photography or SPOT satellite data

i.in.pri ERS1/2 CEOS RADAR reader for PRI products

i.in.gtc ERS1/2 CEOS RADAR reader for GTC products

image rectification

i.ortho.photo ortho photo generation

i.points image registration (assign control points) to raster reference

i.vpoints image registration (assign control points) to vector reference

i.rectify image rectification (linear)

i.rectify2 image rectification (polynomial)

[8] The subgroup names are chosen by the user (hopefully reflecting the contents of the subgroup).
[9] Again, these file names are chosen by the user.

image classification

i.cluster unsupervised clustering

i.maxlik maximum likelihood classifier

i.smap sequential maximum a posteriori classifier

other

i.group image group management

i.target establish target database for the group

8.5 Programming Interface for Groups

The programming interface to the group data is described in *14 Imagery Library (p. 233)* and the reader is referred to that chapter for details.

9 Region and Mask

Chapter status: Needs further updates!

GRASS users are provided with two mechanisms for specifying the area of the earth in which to view and analyze their data. These are known in GRASS as the *region* and the *mask.* The user is allowed to set a *region* which defines a rectangular area of coverage on the earth, and optionally further limit the coverage by specifying a "cookie cutter" *mask.* The region and mask are stored in the database under the user's current mapset. GRASS modules automatically retrieve only data that fall within the region. Furthermore, if there is a mask, only data that fall within the mask are retained. modules determine the region and mask from the database rather than asking the user.

9.1 Region

The user's current database region is set by the user using the GRASS *g.region*, or *d.zoom* commands. It is stored in the WIND file in the mapset. This file not only specifies the geographic boundaries of the region rectangle, but also the region resolution which implicitly grids the region into rectangular "cells" of equal dimension.

Users expect map layers to be resampled into the current region. This implies that raster maps must be extended with no data for portions of the region which do not cover the map layer, and that the raster map data be resampled to the region resolution if the raster map resolution is different. Users also expect new map layers to be created with exactly the same boundaries and resolution as the current region.

The WIND file contains the following fields:

WIND

north:	4660000.00
south:	4570000.00
east :	770000.00
west :	710000.00
e-w resol:	50.00
n-s resol:	100.00
rows:	900
cols:	1200
proj:	1
zone:	18

north, south, east, west

The geographic boundaries of the region are given by the *north, south, east*, and *west* fields. Note: these values describe the lines which bound the region at its edges. These lines do NOT pass through the center of the grid cells which form the region edge, but rather along the edge of the region itself.

rows, cols

These values describe the number of rows and columns in the region.

e-w resol, n-s resol

The fields *e-w resol* and *n-s resol* (which stand for east-west resolution and north-south resolution respectively) describe the size of each grid cell in the region in physical measurement units (e.g., meters in a UTM database). The *e-w resol* is the length of a grid cell from east to west. The *n-s resol* is the length of a grid cell from north to south. Note that since the *e-w resol* may differ from the *n-s resol*, region grid cells need not be square.

proj, zone

The *proj*ection field specifies the type of cartographic projection: 0 is unreferenced x,y (imagery data), 1 is UTM, 2 is State Plane,[1] 3 is Latitude Longitude.[2] Others may be added in the future. The *zone* field is the projection zone. In the example above, the projection is UTM, the zone 18.

Note. The format for the region file "WIND" is very similar to the format for the raster header files. See *5.3 Raster Header Format (p. 29)* for details about raster header files.

[1] State Plane is not yet fully supported in GRASS.
[2] Latitude Longitude is a nonplanimetric projection and is only partially supported in GRASS.

9.2 Mask

[GRASS 5] Note: Floating point masks are not supported...

In addition to the region, the user may set a mask using the *r.mask* command. The mask is stored in the user's current mapset as a raster file with the name MASK.[3] The mask acts like an opaque filter when reading other raster files. No-data values in the mask (i.e., category zero) will cause corresponding values in other raster files to be read as no data (irrespective of the actual value in the raster file).

The following diagram gives a visual idea of how the mask works:

input		
3	4	4
3	3	4
2	3	3

+

MASK		
0	1	1
1	1	0
1	0	0

=

output		
0	4	4
3	3	0
2	0	0

9.3 Variations

If a GRASS module does not obey either the *region* or the *mask*, the variation must be noted in the user documentation for the module, and the reason for the variation given.

[3] The *r.mask module* creates MASK as a reclass file because the reclass function is fast and uses less disk space, but it does not actually matter that MASK is a reclass file. A regular raster file can be used. The only thing that really matters is that the raster file be called MASK.

10 Environment Variables

Chapter status: Needs further updates!

GRASS modules are written to be independent of which database the user is using, where the database resides on the disk, or where the modules themselves reside. When modules need this information, they get some of it from UNIX environment variables, and the rest from GRASS environment variables.

10.1 UNIX Environment

The GRASS start-up command *grass5.0* sets the following UNIX environment variables:[1]

GISBASE top level directory for the GRASS modules

GIS_LOCK process id of the start-up shell script

GISRC name of the GRASS environment file

GISBASE is the top level directory for the GRASS programs. For example, if GRASS were installed under */opt/grass*, then GISBASE would be set to */opt/grass.* The command directory would be */opt/grass/bin*, the command support directory would be */opt/grass/etc*, the source code directory would be */opt/grass/src,* the on-line manual would live in */opt/grass/man*, etc.

GISBASE, while set in the UNIX environment, is given special handling in GRASS code. This variable must be accessed using the *GIS Library* routine *G_gisbase.*

GIS_LOCK is used for various locking mechanisms in GRASS. It is set to the process id of the start-up shell so that locking mechanisms can detect orphaned locks (e.g., locks that were left behind during a system crash).

[1] Any interface to GRASS must set these variables.

GIS_LOCK may be accessed using the UNIX getenv() routine.

GISRC is set to the name of the GRASS environment file where all other GRASS variables are stored. This file is **.grassrc5** in the user's home directory.

10.2 GRASS Environment

All GRASS users will have a file in their home directory named **.grassrc5**[2] which is used to store the variables that comprise the environment of all GRASS programs. This file will always include the following variables that define the database in which the user is working:

GISDBASE toplevel database directory

LOCATION_NAME location directory

MAPSET mapset directory

The user sets these variables during GRASS start-up. While the value of GISDBASE will be relatively constant, the others may change each time the user runs GRASS. GRASS modules access these variables using the *G_gisdbase*, *G_location*, *and G_mapset* routines in the GIS Library. See *4.2 GISDBASE (p. 17)* for details about GISDBASE, *4.3 Locations (p. 18)* for details about database locations, and *4.4 Mapsets (p. 18)* for details about mapsets.

Other variables may appear in this file. Some of these are:

MONITOR currently selected graphics monitor

PAINTER currently selected paint output device

DIGITIZER currently selected digitizer

These variables are accessed and set from C programs using the general purpose routines *G_getenv* and *G_setenv*. The GRASS module *g.gisenv* provides a command level interface to these variables.

[2] GRASS modules do not have this file name built into them. They look it up from the UNIX environment variable GISRC. Note the similarity in naming convention to the .cshrc and .exrc files.

10.3 Difference Between GRASS and UNIX Environments

The GRASS environment is similar to the UNIX environment in that modules can access information stored in "environment" variables. However, since the GRASS environment variables are stored in a disk file, it offers two capabilities not available with UNIX environment variables. First, variables may be set by one module for later use by other programs. For example, the GRASS start-up sets these variables for use by all other GRASS application programs. Second, since the variables remain in the file unless explicitly removed, they are available from session to session. Also, several GRASS environment variables are used as defaults each time a GRASS session is initiated.

11 Compiling and Installing GRASS Modules

GRASS modules are compiled and installed using the GRASS *gmake5* front-end to the UNIX *make* command: *gmake5* reads a file named *Gmakefile* to construct a *make.rules* file (see *11.4.1 Multiple-Architecture Conventions (p. 75)* for more information,) and then runs *make*. The GRASS compilation process allows for multiple-architecture compilation from a single copy of the source code (for instance, if the source code is NFS mounted to various machines with differing architectures.) This chapter assumes that the programmer is familiar with *make* and its accompanying *makefiles.*

Explain ”auto-conf”....

To compile enter following:

```
./configure
make install
```

Then the code will be compiled into "/usr/local/grass-5.0b" directory. The start script "grass5.0beta" and the module compilation scripts "gmake5" and "gmakelinks5" will be placed into "/usr/local/bin".

Optionally other target directories can be specified while "configuring":

```
./configure --prefix=/opt/grass5.0 --with-bindir=/usr/bin
make install
```

This will store the GRASS binaries into the directory "/opt/grass5.0" and the script mentioned above into "/usr/bin".

The script "gmake5" is required to compile single modules, the user has to run "gmakelinks5" afterwards to set internal links for this new module. The compilation process and requirements are discussed in more detail now.

11.1 gmake5

The GRASS *gmake5* utility allows *make* compilation rules to be developed without having to specify machine and installation dependent information. *gmake5* combines predefined variables that specify the machine and installation dependent information with the *Gmakefile,* to create a

makefile. (The predefined variables and the construction of a *Gmakefile* are described in *11.2 Gmakefile Variables (p. 70)*).

gmake5 is invoked as follows:[1]

gmake5 [source directory] [target]

If run without arguments, *gmake5* will run in the current directory, build a *makefile* from the *Gmakefile* found there, and then run *make.* If run with a source directory argument, *gmake5* will change into this directory and then proceed as above. If run with a target argument as well, then *make* will be run on the specified target.

11.2 Gmakefile Variables

The predefined Gmakefile variables which the GRASS programmer must use when writing a *Gmakefile* specify libraries, source and binary directories, compiler and loader flags, etc. The most commonly used variables will be defined here. Examples of how to use them follow in *11.3 Constructing a Gmakefile (p. 72)*. The full set of variables can be seen in *A.1 Appendix A: Annotated Gmakefile Predefined Variables (p. 433)*. Variables marked with (-) are not commonly used.

GRASS Directories: The following variables tell *gmake5* where source code and module directories are:

SRC (-) This is the directory where GRASS source code lives.

BIN This is the directory where user-accessible GRASS modules live.

ETC This is the directory where support files and modules live. These support files and modules are used by the $(BIN) programs, and are not known to, or run by the user.

LIBDIR (-) This is the directory where most of the GRASS libraries are kept.

INCLUDE_DIR (-) This is where include and header files live. For example, ”gis.h” can be found here. gmake5 automatically specifies this directory to the C compiler as a place to find include files.

GRASS Libraries. The following variables name the various GRASS libraries:

GISLIB This names the *GIS Library*, which is the principal GRASS library. See *12 GIS Library (p. 79)* for details about this library, and *12.22 Loading the GIS Library (p. 214)* for a sample Gmakefile which loads this library.

VASKLIB This names the *Vask Library*, which does full screen user input.

[1] When GRASS is installed, *gmake5* is placed into a directory which is in your $PATH (e.g. /usr/local/bin). You should be able to run *gmake5* without having to specify its full path name. This path can be defined differently with parameter to "configure".

VASK This specifies the *Vask Library* plus the UNIX curses and termcap libraries needed to use the *Vask Library* routines. See *20 Vask Library (p. 283)* for details about this library, and *20.4 Loading the Vask Library (p. 287)* for a sample Gmakefile which loads this library.

SEGMENTLIB This names the *Segment Library*, which manages large matrix data. See *19 Segment Library (p. 277)* for details about this library, and *20.4 Loading the Vask Library (p. 287)* for a sample *Gmakefile* which loads this library.

RASTERLIB This names the *Raster Graphics Library*, which communicates with GRASS graphics drivers. See *15 Raster Graphics Library (p. 243)* for details about this library, and *15.10 Loading the Raster Graphics Library (p. 253)* for a sample *Gmakefile* which loads this library.

DISPLAYLIB This names the *Display Graphics Library*, which provides a higher level graphics interface to $(RASTERLIB). See *16 Display Graphics Library (p. 255)* for details about this library, and *16.10 Loading the Display Graphics Library (p. 267)* for a sample *Gmakefile* which loads this library.

UNIX Libraries: The following variables name some useful UNIX system libraries:

MATHLIB This names the math library. It should be used instead of the -lm loader option.

CURSES This names both the curses and termcap libraries. It should be used instead of the -lcurses/-lncurses and -ltermcap loader options. Do not use $(CURSES) if you use $(VASK).

TERMLIB This names the termcap library. It should be used instead of the -ltermcap or -ltermlib loader options. Do not use $(TERMLIB) if you use $(VASK) or $(CURSES).

Compiler and loader variables. The following variables are related to compiling and loading C programs:

CC This variable specifies what compiler/loader to use. This should always be referenced, as opposed to ”cc”. See *11.3.1 Building modules from source (.c) files (p. 72)* for the proper use of the CC variable.

AR This variable specifies the rule that must be used to build object libraries. See *11.3.3 Building object libraries (p. 74)* for details.

CFLAGS (-) This variable specifies all the C compiler options. It should never be necessary to use this variable - *gmake5* automatically supplies this variable to the C compiler.

EXTRA_CFLAGS This variable can be used to add additional options to $(CFLAGS). It has no predefined values. It is usually used to specify additional -I include directories, or -D preprocessor defines.

GMAKE This is the full name of the *gmake5* command. It can be used to drive compilation in subdirectories.

LDFLAGS This specifies the loader flags. The programmer must use this variable when loading GRASS modules since there is no way to automatically supply these flags to the loader.

MAKEALL This defines a command which runs *gmake5* in all subdirectories that have a *Gmakefile* in them.

11.3 Constructing a Gmakefile

A *Gmakefile* is constructed like a *makefile.* The complete syntax for a *makefile* is discussed in the UNIX documentation for *make* and will not be repeated here. The essential idea is that a target (e.g. a GRASS module) is to be built from a list of dependencies (e.g. object files, libraries, etc.). The relationship between the target, its dependencies, and the rules for constructing the target is expressed according to the following syntax:

target : dependencies

actions

more actions

If the target does not exist, or if any of the dependencies have a newer date than the target (i.e., have changed), the actions will be executed to build the target. The actions must be indented using a TAB. *Make* is picky about this. It does not like spaces in place of the TAB.

11.3.1 Building modules from source (.c) files

To build a module from C source code files, it is only necessary to specify the compiled object (.o) files as dependencies for the target module, and then specify an action to load the object files together to form the module. The *make* utility builds .o files from .c files without being instructed to do so.

For example, the following *Gmakefile* builds the module *xyz* and puts it in the GRASS module directory.

OBJ = main.o sub1.o sub2.o sub3.o

$(BIN)/xyz: $(OBJ) $(GISLIB)

$(CC) $(LDFLAGS) -o $@ $(OBJ) $(GISLIB) $(XDRLIB)

$(GISLIB): # in case of library changes

The target *xyz* depends on the object files listed in the variable $(OBJ), the $(GISLIB) and the $(XDRLIB) library. The action runs the C compiler to load xyz from the $(OBJ) files, the $(GISLIB) and the $(XDRLIB).

$@ is a *make* shorthand which stands for the target, in this case *xyz*. Its use should be encouraged, since the target name can be changed without having to edit the action as well.

$(CC) is the C compiler. It is used as the interface to the loader. It should be specified as $(CC) instead of cc. *Make* may define $(CC) as cc, but using $(CC) will allow other C-like compilers to be used instead.

$(BIN) is a *gmake5* variable which names the UNIX directory where GRASS commands live. Specifying the target as $(BIN)/xyz will cause *gmake5* to build *xyz* directly into the $(BIN) directory.

$(LDFLAGS) specify loader flags which must be passed to the loader in this manner.

$(GISLIB) is the *GIS Library*. $(GISLIB) is specified on the action line so that it is included during the load step. It is also specified in the dependency list so that changes in $(GISLIB) will also cause the module to be reloaded. Note that no rules were given for building the .o files from their related .c files. In fact, the GRASS programmer should never giv e an explicit rule for compiling .c files. It is sufficient to list all the .o files as dependencies of the target. The .c files will be automatically compiled to build up-to-date .o files before the .o files are loaded to build the target module.

Also note that since $(GISLIB) is specified as a dependency it must also be specified as a target. *Make* must be told how to build all dependencies as well as targets. In this case a dummy rule is given to satisfy *make*.

11.3.2 Include files

Often C code uses the # include directive to include header files in the source during compilation. Header files that are included into C source code should be specified as dependencies as well. It is the .o files which depend on them:

```
OBJ = main.o sub1.o sub2.o

$(BIN)/xyz: $(OBJ) $(GISLIB)

$(CC) $(LDFLAGS) -o $@ $(OBJ) $(GISLIB)

$(OBJ): myheader.h

$(GISLIB): # in case library changes
```

In this case, it is assumed that ”myheader.h” lives in the current directory and is included in each source code file. If ”myheader.h” changes, then all .c files will be compiled even though they may not have changed. And then the target module *xyz* will be reloaded.

If the header file ”myheader.h” is in a different directory, then a different formulation can be used:

EXTRA_CFLAGS = -I..

OBJ = main.o sub1.o sub2.o

$(BIN)/xyz: $(OBJ) $(GISLIB)

$(CC) $(LDFLAGS) -o $@ $(OBJ) $(GISLIB) $(XDRLIB)

$(GISLIB): # in case of library changes

$(EXTRA_CFLAGS) will add the flag -I.. to the rules that compile .c files into .o files. This flag indicates that #include files (i.e., ”myheader.h”) can also be found in the parent (..) directory.

Note that this example does not specify that ”myheader.h” is a dependency. If ”myheader.h” were to change, this would not cause recompilation here. The following rule could be added:

$(OBJ): ../myheader.h

11.3.3 Building object libraries

Sometimes it is desirable to build libraries of subroutines which can be used in many programs. *gmake5* requires that these libraries be built using the $(AR) rule as follows:

OBJ = sub1.o sub2.o sub3.o

lib.a: $(OBJ)

$(AR)

All the object files listed in $(OBJ) will be compiled and archived into the target library *lib.a.* The $(OBJ) variable must be used. The $(AR) assumes that all object files are listed in $(OBJ).

Note that due to the way the $(AR) rule is designed, it is not possible to construct more than one library in a single source code directory. Each library must have its own directory and related *Gmakefile.*

11.3.4 Building more than one target

Many *target : dependency* lines many be giv en. However, it is the first one in the *Gmakefile* which is built by *gmake5.* If there are more targets to be built, the first target must explicitly or implicitly cause *gmake5* to build the others.

The following builds two programs, *abc* and *xyz* directly into the $(BIN) directory:

```
ABC = abc.o sub1.o sub2.o

XYZ = xyz.o sub1.o sub3.o

all: $(BIN)/abc $(BIN)/xyz

$(BIN)/abc: $(ABC) $(GISLIB)

$(CC) $(LDFLAGS) -o $@ $(ABC) $(GISLIB) $(XDRLIB)

$(BIN)/xyz: $(XYZ) $(GISLIB)

$(CC) $(LDFLAGS) -o $@ $(XYZ) $(GISLIB) $(XDRLIB)

$(GISLIB): # in case library changes
```

If it is desired to run the compilation in various subdirectories, a *Gmakefile* could be constructed which simply runs *gmake5* in each subdirectory. For example:

```
all:

$(GMAKE) subdir.1

$(GMAKE) subdir.2

$(GMAKE) subdir.3
```

11.4 Compilation Results

This section describes the results of the GRASS compilation process for two separate subjects.

11.4.1 Multiple-Architecture Conventions

The following conventions allow for multiple architecture compilation on a machine that uses a common or networked GRASS source code directory tree.

Object files and library archives are compiled into subdirectories that represent the architecture that they were compiled on. These subdirectories are created in the $(SRC) directory as OBJ.*arch* and LIB.*arch*, where *arch* represents the architecture of the compiling machine. Thus, for example, $(SRC)/OBJ.sun4 would contain the object files for Sun/4 and SPARC architectures, and $(SRC)/LIB.Linux would contain library archives for Linux architectures. Likewise, $(SRC)/OBJ.Linux would contain the object files for Linux architectures, and $(SRC)/LIB.Linux would contain library archives for Linux architectures.

Note that 'arch' is defined for a specific architecture during setup and compilation of GRASS, it is not limited to sun4 or any specific string.

gmake5 produces a make.rules file in the $(SRC)/OBJ.arch directory instead of a makefile to allow for multiple-architecture compilation.

11.4.2 Compiled Command Destinations

GRASS V.5 merges the command-line and interactive versions of a function under the same name. This merging happens in one of two methods:

1. The programmer writes a single module which uses the new parser capability (see *12.16 Command Line Parsing (p. 186)*). The parser has both a command-line and a rudimentary prompt-based interactive interface.

2. The programmer writes a command-line version using the parser, but also provides an interactive version as a separate module to override the parser's interactive interface.

The second method requires that both the command-line module and the interactive module be somehow merged into one module. This is accomplished by placing both modules in separate directories under $(GISBASE)/etc/bin and creating a link (as described below) in $(BIN).

There are six directories where modules are placed. These, along with their respective Gmakefile variables, are:

etc/bin/main/inter $(BIN_INTER)

Interactive versions of the primary GRASS commands.

etc/bin/main/cmd $(BIN_CMD)

Command-line versions of the primary GRASS commands.

To merge the command-line and interactive versions of a command, the compilation process creates a link in $(BIN) to $(GISBASE)/etc/front.end. This link has the same name as the command, and causes execution of the command to be passed to a front-end. The behavior of the front.end command is shown in the figure below using the command *r.reclass* as an example.

paste front.end.xfig diagram here (not existing.. 2/2000)

The front.end module will call the interactive version of the command if there were no command-line arguments entered by the user. Otherwise, it will run the command-line version. If only one version of the specific command exists (for example, there is only a command-line version available,) that one existing command is executed.

11.5 Notes

11.5.1 Bypassing the creation of .o files

If a module has only one .c source file, it is tempting to compile the module directly from the .c file without creating the .o file. Please do not do this. There have been problems on some systems specifying both compiler and loader flags at the same time. The .o files must be built first. Once all the .o files are built, they are loaded with any required libraries to build the module.

11.5.2 Simultaneous compilation

The compilation process may be run on only one machine at a time. If you try to compile the same source directory on two machines simultaneously, things will not turn out properly. This is your responsibility–gmake5 cannot detect simultaneous compilations.

12 GIS Library

12.1 Introduction to GIS Library

The *GIS Library* is the primary programming library provided with the GRASS system. **Programs must use this libary to access the database.** It contains the routines which locate, create, open, rename, and remove GRASS database files. It contains the routines which read and write raster files. It contains routines which interface the user to the database, including prompting the user, listing available files, validating user access, etc. It also has some general purpose routines (string manipulation, user information, etc.) which are not tied directly to database processing.

It is assumed that the reader has read *4 Database Structure (p. 17)* for a general description of GRASS databases, *5 Raster Maps (p. 27)* for details about raster map layers in GRASS, and *9 Region and Mask (p. 61)* which discusses regions and masks. The routines in the *GIS Library* are presented in functional groupings, rather than in alphabetical order. The order of presentation will, it is hoped, provide a better understanding of how the library is to be used, as well as show the interrelationships among the various routines. Note that a good way to understand how to use these routines is to look at the source code for GRASS modules which use them. Most routines in this library require that the header file "gis.h" be included in any code using these routines.[1] Therefore, programmers should always include this file when writing code using routines from this library:

```
#include "gis.h"
```

Note. All routines and global variables in this library, documented or undocumented, start with the prefix **G_.** To avoid name conflicts, programmers should not create variables or routines in their own modules which use this prefix.

An alphabetic index is provided in *A.3 Appendix C: Index to GIS Library (p. 438)*.

12.2 Library Initialization

It is **mandatory** that the system be initialized before any other library routines are called.

[1] The GRASS compilation process, described in *11 Compiling and Installing GRASS Modules (p. 69)*, automatically tells the C compiler how to find this and other GRASS header files.

initialize gis library

int
G_gisinit (char *program_name)

This routine reads the user's GRASS environment file into memory and makes sure that the user has selected a valid database and mapset. It also initializes hidden variables used by other routines. If the user's database information is invalid, an error message is printed and the module exits. The **program_name** is stored for later recall by *G_program_name.* It is recommended that argv[0] be used for the **program_name:**

```
int main (int argc, char **argv)
{
G_gisinit(argv[0]);
}
```

12.3 Diagnostic Messages

The following routines are used by other routines in the library to report warning and error messages. They may also be used directly by GRASS programs.

print error message and exit

int
G_fatal_error (char *message, ...)

print warning message and continue

int
G_warning (char *message, ...)

These routines report errors to the user. The normal mode is to write the **message** to the screen (on the standard error output) and wait a few seconds. G_warning( ) will return and G_fatal_error( ) will exit.
If the standard error output is not a tty device, then the message is mailed to the user instead.
If the file GIS_ERROR_LOG exists (with write permission), in either the user's home directory or in the $GISBASE[2] directory, the messages will also be logged to this file.

While most applications will find the normal error reporting quite adequate, there will be times when different handling is needed. For example, graphics modules may want the messages displayed graphically instead of on the standard error output. If the programmer wants to handle the error messages differently, the following routines can be used to modify the error handling:

change error handling

int

[2] $GISBASE is the directory where GRASS is installed. See *10.1 UNIX Environment (p. 65)* for details.

G_set_error_routine (int (*handler)( ))

This routine provides a different error handler for G_fatal_error( ) and G_warning( ). The **handler** routine must be defined as follows:

```
int handler (char *message, int fatal)
```

where **message** is the message to be handled and **fatal** indicates the type of error: 1 (fatal error) or 0 (warning).
Note. The handler only provides a way to send the message somewhere other than to the error output. If the error is fatal, the module will exit after the handler returns.

int
G_unset_error_routine (void)

reset normal error handling

This routine resets the error handling for *G_fatal_error* and *G_warning* back to the default action.

int
G_sleep_on_error (int flag)

sleep on error?

If **flag** is 0, then no pause will occur after printing an error or warning message. Otherwise the pause will occur.

int
G_suppress_warnings (int flag)

suppress warnings?

If **flag** is 0, then *G_warning* will no longer print warning messages. If **flag** is 1, then G_warning( ) will print warning messages.

Note. This routine has no effect on *G_fatal_error.*

12.4 Environment and Database Information

The following routines return information about the current database selected by the user. Some of this information is retrieved from the user's GRASS environment file. Some of it comes from files in the database itself. See *10 Environment Variables (p. 65)* for a discussion of the GRASS environment.

The following four routines can be used freely by the programmer:

char *
G_location (void)

current location name

Returns the name of the current database location. This routine should be used by modules that need to display the current location to the user. See *4.3 Locations (p. 18)* for an explanation of locations.

current mapset name

char *
G_mapset (void)

Returns the name of the current mapset in the current location. This routine is often used when accessing files in the current mapset. See *4.4 Mapsets (p. 18)* for an explanation of mapsets.

location title

char *
G_myname (void)

Returns a one line title for the database location. This title is read from the file MYNAME in the PERMANENT mapset. See also *4.6 Permanent Mapset (p. 21)* for a discussion of the PERMANENT mapset.

top level module directory

char *
G_gisbase (void)

Returns the full path name of the top level directory for GRASS programs. This directory will have subdirectories which will contain modules and files required for the running of the system. Some of these directories are:

```
  bin     commands run by the user
  etc     modules and data files used by GRASS
commands
  txt     help files
  menu    files used by the grass3 menu interface
```

The use of G_gisbase( ) to find these subdirectories enables GRASS modules to be written independently of where the GRASS system is actually installed on the machine. For example, to run the module *sroff* in the GRASS *etc* directory:

```
char command[200];

sprintf (command, "%s/etc/sroff", G_gisbase( ) );
system (command);
```

The following two routines return full path UNIX directory names. They should be used only in special cases. They are used by other routines in the library to build full UNIX file names for database files. **The programmer should not use the next two routines to bypass the normal database access routines.**

char *
G_gisdbase (void)

top level database directory

Returns the full UNIX path name of the directory which holds the database locations. See *4.2 GISDBASE (p. 17)* for a full explanation of this directory.

char *
G_location_path (void)

current location directory

Returns the full UNIX path name of the current database location. For example, if the user is working in location *spearfish* in the */usr/grass5/data* database directory, this routine will return a string which looks like */home/user/grassdata/spearfish*.

These next routines provide the low-level management of the information in the user's GRASS environment file. **They should not be used in place of the higher level interface routines described above.**

int
G_getenv (char *name)

query GRASS environment variable

int
G_ _getenv (char *name)

query GRASS environment variable

These routines look up the variable **name** in the GRASS environment and return its value (which is a character string). If **name** is not set, G_getenv( ) issues an error message and calls exit( ). G_ _setenv( ) just returns the NULL pointer.

int
G_setenv (char *name, char *value)

set GRASS environment variable

int
G_ _setenv (char *name, char *value)

set GRASS environment variable

These routines set the the GRASS environment variable **name** to **value.** If **value** is NULL, the **name** is unset.

Both routines set the value in module memory, but only G_setenv( ) writes the new value to the user's GRASS environment file.

12.5 Fundamental Database Access Routines

The routines described in this section provide the low-level interface to the GRASS database. They search the database for files, prompt the user for file names, open files for reading or writing, etc. The programmer should never bypass this level of database interface. These routines must be used to access the GRASS database unless there **are other higher level library routines which perform the same function.** For example, routines to process raster files (*12.9 Raster File Processing (p. 105)*), vector files (*12.12 Vector File Processing (p. 161)*), or site files (*12.13 Site List Processing (GRASS 5 Sites API) (p. 166)*), etc., should be used instead.

In the descriptions below, the term database *element* is used. Elements are subdirectories within a mapset and are associated with a specific GRASS data type. For example, raster files live in the "cell" and "fcell" element. See *4.5.2 Elements (p. 20)* for more details.

12.5.1 Prompting for Database Files

The following routines interactively prompt the user for a file name from a specific database **element.** (See *4.5.2 Elements (p. 20)* for an explanation of elements.) In each, the **prompt** string will be printed as the first line of the full prompt which asks the user to enter a file name. If **prompt** is the empty string "" then an appropriate prompt will be substituted. The name that the user enters is copied into the **name** buffer.[3] The short (one or two word) **label** describing the **element** is used as part of a title when listing the files **in element.**

The user is required to enter a valid file name, or else hit the RETURN key to cancel the request. If the user enters an invalid response, a message is printed, and the user is prompted again. If the user cancels the request, the NULL pointer is returned. Otherwise the mapset where the file lives or is to be created is returned. Both the name and the mapset are used in other routines to refer to the file.

An example will be given here. The G_ask_old( ) routine used in the example is described a bit later. The user is asked to enter a file from the "paint/labels" element:

```
char name[50];
char *mapset;
mapset = G_ask_old ("", name, "paint/labels", "labels");
if (mapset = = NULL)
exit(0); /* user canceled the request */
```

[3] The size of **name** should be large enough to hold any GRASS file name. Most systems allow file names to be quite long. It is recommended that name be declared *char name[50].*

The user will see the following:

```
Enter the name of an existing labels file
Enter 'list' for a list of existing labels files
Hit RETURN to cancel request [4]
>
```

char *
G_ask_old (char *prompt, char *name, char *element, char *label)

prompt for existing database file

The user is asked to enter the name of an existing database file.
Note. This routine looks for the file in the current mapset as well as other mapsets. The mapsets that are searched are determined from the user's mapset search path. See *4.7.1 Mapset Search Path (p. 22)* for some more details about the search path.

char *
G_ask_new (char *prompt, char *name, char *element, char *label)

prompt for new database file

The user is asked to enter the name of a new file which does not exist in the current mapset.
Note. The file chosen by the user may exist in other mapsets. This routine does not look in other mapsets, since the assumption is that **name** will be used to create a new file. New files are always created in the current mapset.

char *
G_ask_in_mapset (char *prompt, char *name, char *element, char *label)

prompt for existing database file

The user is asked to enter the name of an file which exists in the current mapset.
Note. The file chosen by the user may or may not exist in other mapsets. This routine does not look in other mapsets, since the assumption is that **name** will be used to modify a file. GRASS only permits users to modify files in the current mapset.

char *
G_ask_any (char *prompt, char *name, char *element, char *label, int warn)

prompt for any valid file name

[4] This line of the prompt can be modified using *G_set_ask_return_msg*.

The user is asked to enter any leg al file name. If **warn** is 1 and the file chosen exists in the current mapset, then the user is asked if it is ok to overwrite the file. If **warn** is 0, then any leg al name is accepted and no warning is issued to the user if the file exists.

set Hit RETURN msg int **G_set_ask_return_msg (char *msg)**

The "Hit RETURN to cancel request" part of the prompt in the prompting routines described above, is modified to "Hit RETURN **msg.**"

get Hit RETURN msg char * **G_get_ask_return_msg (void)**

The current *msg* (as set by *G_set_ask_return_msg*) is returned.

12.5.2 Fully Qualified File Names

All GRASS routines which access database files must be given both the file name and the mapset where the file resides. Often the name and the mapset are 2 distinct character strings. However, there is a need for a single character string which contains both the name and the mapset (e.g., for interactive interfacing to command-line programs). This form of the name is known as the *fully qualified file name* and is built by the following routine:

fully qualified file name char * **G_fully_qualified_name (char *name, char *mapset)**

Returns a fully qualified name for the file **name** in **mapset.** Currently this string is in the form *name@mapset*, but the programmer should pretend not to know this and always call this routine to get the fully qualified name.
The following example shows how an interactive version of *d.rast* interfaces with the command-line version of *d.rast*:

```
#include "gis.h"
int main(char *argc, char **argv)
{
char name[100], *mapset, *fqn;;
char command[1024];
G_gisinit(argv[0]);
mapset = G_ask_cell_old ("", name, "");
if (mapset = = NULL) exit(0);
fqn = G_fully_qualified_name (name, mapset);
sprintf (command, "d.rast map='%s'", fqn);
system(command);
}
```

12.5.3 Finding Files in the Database

Noninteractive modules cannot make use of the interactive prompting routines described above. For example, a command line driven module may require a database file name as one of the command arguments. In this case, the programmer must search the database to find the mapset where the file resides.

The following routines search the database for files:

char *
G_find_file (char *element, char *name, char *mapset)

find a database file

Look for the file **name** under the specified **element** in the database. The **mapset** parameter can either be the empty string "", which means search all the mapsets in the user's current mapset search path,[5] or it can be a specific mapset, which means . look for the file only in this one mapset (for example, in the current mapset).
If found, the mapset where the file lives is returned. If not found, the NULL pointer is returned.
If the user specifies a fully qualified file name, (i.e, a name that also contains the mapset; see *12.5.2 Fully Qualified File Names (p. 86)*) then *G_find_file( )* modifies **name** by eliminating the mapset from the **name**
For example, to find a "paint/labels" file anywhere in the database:

```
char name[50];
char *mapset;
if ((mapset = G_find_file("paint/labels",name,""))
== NULL)
/* not found */
```

To check that the file exists in the current mapset:

```
char name[50];
if (G_find_file("paint/labels",name,G_mapset( ))
== NULL)
/* not found */
```

12.5.4 Legal File Names

Not all names that a user may enter will be legal files for the GRASS databases. The routines which create new files require that the new file have a leg al name. The routines which prompt the user for file names (e.g., *G_ask_new*) guarantee that the name entered by the user will be legal. If the name is obtained from the command line, for example, the programmer must check that the name is legal. The following routine checks for legal file names:

int

check for legal database file names

[5] See *4.7.1 Mapset Search Path (p. 22)* for more details about the search path

G_legal_filename (char *name)

Returns 1 if **name** is ok, -1 otherwise.

12.5.5 Opening an Existing Database File for Reading

The following routines open the file **name** in **mapset** from the specified database **element** for reading (but not for writing). The file **name** and **mapset** can be obtained interactively *using G_ask_old, and noninteractively using G_find_file.*

open a database file for reading

int
G_open_old (char *element, char *name, char *mapset)

The database file **name** under the **element** in the specified **mapset** is opened for reading (but not for writing).
The UNIX open( ) routine is used to open the file. If the file does not exist, -1 is returned. Otherwise the file descriptor from the open( ) is returned.

open a database file for reading

FILE *
G_fopen_old (char *element, char *name, char *mapset)

The database file **name** under the **element** in the specified **mapset** is opened for reading (but not for writing).
The UNIX fopen( ) routine, with "r" read mode, is used to open the file. If the file does not exist, the NULL pointer is returned. Otherwise the file descriptor from the fopen( ) is returned.

12.5.6 Opening an Existing Database File for Update

The following routines open the file **name** in the current mapset from the specified database **element** for writing. The file must exist. Its **name** can be obtained interactively *using G_ask_in_mapset, and noninteractively using G_find_file.*

open a database file for update

int
G_open_update (char *element, char *name)

The database file **name** under the **element** in the current mapset is opened for reading and writing.
The UNIX open( ) routine is used to open the file. If the file does not exist, -1 is returned. Otherwise the file is positioned at the end of the file and the file descriptor from the open( ) is returned.

int
G_fopen_append (char *element, char *name)

open a database file for update

The database file **name** under the **element** in the current mapset is opened for appending (but not for reading).
The UNIX fopen( ) routine, with "a" append mode, is used to open the file. If the file does not exist, the NULL pointer is returned. Otherwise the file is positioned at the end of the file and the file descriptor from the fopen( ) is returned.

12.5.7 Creating and Opening a New Database File

The following routines create the new file **name** in the current mapset[6] under the specified database **element** and open it for writing. The database **element** is created, if it does not already exist.

The file **name** should be obtained interactively using *G_ask_new.* If obtained noninteractively (e.g., from the command line), *G_legal_filename* should be called first to make sure that **name** is a valid GRASS file name. **Warning.** It is not an error for **name** to already exist. However, the file will be removed and recreated empty. The interactive routine *G_ask_new* guarantees that **name** will not exist, but if **name** is obtained from the command line, **name** may exist. In this case *G_find_file* could be used to see if **name** exists.

int
G_open_new (char *element, char *name)

open a new database file

The database file **name** under the **element** in the current mapset is created and opened for writing (but not reading).
The UNIX open( ) routine is used to open the file. If the file does not exist, -1 is returned. Otherwise the file is positioned at the end of the file and the file descriptor from the open( ) is returned.

FILE *
G_fopen_new (char *element, char *name)

open a new database file

The database file **name** under the **element** in the current mapset is created and opened for writing (but not reading).
The UNIX fopen( ) routine, with "w" write mode, is used to open the file. If the file does not exist, the NULL pointer is returned. Otherwise the file is positioned at the end of the file and the file descriptor from the fopen( ) is returned.

[6] GRASS does not allow files to be created outside the current mapset; see *4.7 Database Access Rules (p. 22).*

12.5.8 Database File Management

The following routines allow the renaming and removal of database files in the current mapset.[7]

rename a database file

int
G_rename (char *element, char *old, char *new)

The file or directory **old** under the database **element** directory in the current mapset is renamed to **new.**
Returns 1 if successful, 0 if **old** does not exist, and -1 if there was an error.
Bug. This routine does not check to see if the **new** name is a valid database file name.

remove a database file

int
G_remove (char *element, char *name)

The file or directory **name** under the database **element** directory in the current mapset is removed.
Returns 1 if successful, 0 if **name** does not exist, and -1 if there was an error.
Note. If **name** is a directory, everything within the directory is removed as well.

Note. These functions only apply to the specific **element** and not to other "related" elements. For example, if **element** is "cell", then the specified raster file will be removed (or renamed), but the other support files, such as "cellhd" or "cats", will not. To remove these other files as well, specific calls must be made for each related **element.**

12.6 Memory Allocation

The following routines provide memory allocation capability. They are simply calls to the UNIX suite of memory allocation routines malloc( ), realloc( ) and calloc( ), except that if there is not enough memory, they print a diagnostic message to that effect and then call exit( ).

Note. Use the G_free( ) routine to release memory allocated by these routines.

free the memory allocated

int
G_free(void *buf)

Free the memory allocated by the GRASS malloc routines.

[7] These functions only apply to the current mapset since GRASS does permit users to modify things in mapsets other than the current mapset; see *4.7 Database Access Rules (p. 22)*.

void * — *memory allocation*
G_malloc (int size)

Allocates a block of memory at least **size** bytes which is aligned properly for all data types. A pointer to the aligned block is returned.

void * — *memory allocation*
G_realloc (void *ptr, int size)

Changes the **size** of a previously allocated block of memory at **ptr** and returns a pointer to the new block of memory. The **size** may be larger or smaller than the original size. If the original block cannot be extended "in place", then a new block is allocated and the original block copied to the new block.
Note. If **ptr** is NULL, then this routine simply allocates a block of **size** bytes. This routine works around broken realloc( ) routines, which do not handle a NULL **ptr.**

void * — *memory allocation*
G_calloc (int n, int size)

Allocates a properly aligned block of memory **n***__size__ bytes in length, initializes the allocated memory to zero, and returns a pointer to the allocated block of memory.
Note. Allocating memory for reading and writing raster files is discussed in *12.9.5 Allocating Raster I/O Buffers (p. 109)*.

double * — *memory allocation*
G_alloc_vector(int n)

Allocate a vector (array) of **n** doubles initialized to zero.

float * — *memory allocation*
G_alloc_fvector(int n)

Allocate a vector (array) of **n** floats initialized to zero.

double ** — *memory allocation*
G_alloc_matrix(int rows, int cols)

Allocate a matrix of **rows** by **cols** doubles initialized to zero.

float **
G_alloc_fmatrix(int rows, int cols) *memory allocation*

Allocate a matrix of **rows** by **cols** floats initialized to zero.

memory deallocation int
G_free_vector(double *v)

Deallocate a vector (array) of doubles or floats.

memory deallocation int
G_free_matrix(double **m)

Deallocate a matrix of doubles.

memory deallocation int
G_free_fmatrix(float **m)

Deallocate a matrix of floats.

12.7 The Region

The region concept is explained in *9.1 Region (p. 61)*. It can be thought of as a two-dimensional matrix with known boundaries and rectangular cells.

There are logically two different regions. The first is the database region that the user has set in the current mapset. The other is the region that is active in the module. This active module region is what controls reading and writing of raster file data and sites files. The vector map export does not take care for the active region settings.

The routines described below use a GRASS data structure *Cell_head* to hold region information. This structure is defined in the "gis.h" header file. It is discussed in detail under *12.21 GIS Library Data Structures (p. 212)*.

12.7.1 The Database Region

Reading and writing the user's database region[8] are done by the following routines:

int
G_get_window (struct Cell_head *region)

read the database region

Reads the database region as stored in the WIND file in the user's current mapset **into region.**
An error message is printed and exit( ) is called if there is a problem reading the region.
Note. GRASS applications that read or write raster files should not use this routine since its use implies that the active module region will not be used. Programs that read or write raster file data (or vector data) can query the active module region *using G_window_rows and G_window_cols.*.

int
G_put_window (struct Cell_head *region)

write the database region

Writes the database region file (WIND) in the user's current mapset from **region.** Returns 1 if the region is written ok. Returns -1 if not (no diagnostic message is printed).
Warning. Since this routine actually changes the database region, it should only be called by modules which the user knows will change the region. It is probably fair to say that under GRASS 3.0 only the *g.region*, and *d.zoom* modules should call this routine.

There is another database region. This region is the default region for the location. The default region provides the user with a "starting" region, i.e., a region to begin with and return to as a reference point. The GRASS modules *g.region* allow the user to set their database region from the default region. (See *4.6 Permanent Mapset (p. 21)* for a discussion of the default region.) The following routine reads this region:

int
G_get_default_window (struct Cell_head *region)

read the default region

Reads the default region for the location into **region.**
An error message is printed and exit( ) is called if there is a problem reading the default region.

[8] Previous versions of GRASS called this the "window". Due to overuse of this term (database window, graphics window, etc.), the term was changed to "region". However, to maintain compatibility with existing programs, library routine names were not changed - hence the term "window" is used in the routine name (where "region" should probably be used instead.)

12.7.2 The Active Module Region

The active module region is the one that is used when reading and writing raster file data. This region determines the resampling when reading raster data. It also determines the extent and resolution of new raster files.

Initially the active module region and the user's database region are the same, but the programmer can make them different. The following routines manage the active module region.

number of rows in active region

int
G_window_rows (void)

number of columns in active region

int
G_window_cols (void)

These routines return the number of rows and columns (respectively) in the active module region. Before raster files can be read or written, it is necessary to known how many rows and columns are in the active region. For example:

```
int nrows, cols;
int row, col;
nrows = G_window_rows( );
ncols = G_window_cols( );
for (row = 0; row < nrows; row++)
{
read row ...
for (col = 0; col < ncols; col++)
{
process col ...
}
}
```

set the active region

int
G_set_window (struct Cell_head *region)

This routine sets the active region from **region.** Setting the active region does not change the WIND file in the database. It simply changes the region for the duration of the module.[9] A warning message is printed and -1 returned if **region** is not valid. Otherwise 1 is returned.
Note. This routine overrides the region as set by the user. Its use should be very limited since it changes what the user normally expects to happen. If this routine is not called, then the active region will be the same as what is in the user's WIND file.

[9] However, the new region setting is not retained across the UNIX exec( ) call. This implies that G_set_window( ) cannot be used to set the region for a module to be executed using the system( ) or popen( ) routines.

Warning. Calling this routine with already opened raster files has some side effects. If there are raster files which are open for reading, they will be read into the newly set region, not the region that was active when they were opened. However, CELL buffers allocated for reading the raster files are not automatically reallocated. The module must reallocate them explicitly. Also, this routine does not change the region for raster files which are open for writing. The region that was active when the open occurred still applies to these files.

int *get the active*
G_get_set_window (struct Cell_head *region) *region*

Gets the values of the currently active region into **region.** If *G_set_window* has been called, then the values set by that call are retrieved. Otherwise the user's database region is retrieved.
Note. For modules that read or write raster data, and really need the full region information, this routine is preferred over *G_get_window.* However, since *G_window_rows and G_window_cols return the number of rows and* columns in the active region, the programmer should consider whether or not the full region information is really needed before using this routine.

char * *align two*
G_align_window (struct Cell_head *region, struct Cell_head *ref) *regions*

Modifies the input **region** to align to the **ref** region. The resolutions in **region** are set to match those in **ref** and the **region** edges (north, south, east, west) are modified to align with the grid of the **ref** region.
The **region** may be enlarged if necessary to achieve the alignment. The north is rounded northward, the south southward, the east eastward and the west westward. This routine returns NULL if ok, otherwise it returns an error message.

double *column to*
G_col_to_easting (double col, struct Cell_head *region) *easting*

Converts a **col**umn relative to a **region** to an easting;
Note. col is a double: col+0.5 will return the easting for the center of the column; col+0.0 will return the easting for the western edge of the column; and col+1.0 will return the easting for the eastern edge of the column.

double *row to northing*
G_row_to_northing (double row, struct Cell_head *region)

Converts a **row** relative to a **region** to a northing;
Note. row is a double: row+0.5 will return the northing for the center of the row; row+0.0 will return the northing for the northern edge of the row; and row+1.0 will return the northing for the southern edge of the row. double **G_easting_to_col** (east, region) *easting to column* double east; struct Cell_head *region;
Converts an **east**ing relative to a **region** to a column.
Note. The result is a double. Casting it to an integer will give the column number.

northing to row double
G_northing_to_row (double north, struct Cell_head *region)

Converts a **north**ing relative to a **region** to a row.
Note. the result is a double. Casting it to an integer will give the row number.

12.7.3 Projection Information

The following routines return information about the cartographic projection and zone. See *9.1 Region (p. 61)* for more information about these values.

query cartographic projection int
G_projection (void)

This routine returns a code indicating the projection for the active region. The current values are:
0 unreferenced x,y (imagery data)
1 UTM
2 State Plane
3 Latitude-Longitude[10]
Others may be added in the future. HINT GRASS 5: 121 projections!!

query cartographic projection char *
G_database_projection_name (int proj)

Returns a pointer to a string which is a printable name for projection code **proj** (as returned by *G_projection*). Returns NULL if **proj** is not a valid projection.

database units char *
G_database_unit_name (int plural)

[10] Latitude-Longitude is not yet fully supported in GRASS.

Returns a string describing the database grid units. It returns a plural form (eg. feet) if **plural** is true. Otherwise it returns a singular form (eg. foot).

double
G_database_units_to_meters_factor (void)

conversion to meters

Returns a factor which converts the grid unit to meters (by multiplication). If the database is not metric (eg. imagery) then 0.0 is returned.

int
G_zone (void)

query cartographic zone

This routine returns the zone for the active region. The meaning for the zone depends on the projection. For example zone 18 for projection type 1 would be UTM zone 18.

12.8 Latitude-Longitude Databases

GRASS supports databases in a longitude-latitude grid using a projection where the x coordinate is the longitude and the y coordinate is the latitude. This projection is called the Equidistant Cylindrical Projection.[11] ECP has the property that *where am I* and *row-column* calculations are identical to those in planimetric grids (like UTM[12]). This implies that normal GRASS registration and overlay functions will work without any special considerations or modifications to existing code. However, the projection is not planimetric. This means that distance and area calculations are no longed Euclidean.

Also, since the world is round, maps may not have edges in the east-west direction, especially for global databases. Maps may have the same longitude at both the east and west edges of the display. This feature, called global wraparound, must be accounted for by GRASS modules (particularly vector based functions, like plotting.) What follows is a description of the GISLIB library routines that are available to support latitude-longitude databases.

12.8.1 Coordinates

Latitudes and longitudes are specified in degrees. Northern latitudes range from 0 to 90 degrees, and southern latitudes from 0 to -90. Longitudes have no limits since longitudes ś360 degrees are equivalent.

[11] Also known as Plate Carree.
[12] Universal Transverse Mercator Projection.

Coordinates are represented in ASCII using the format **dd:mm:ssN** or **dd:mm:ssS** for latitudes, **ddd:mm:ssE** or **ddd.mm.ssW** for longitudes, and **dd.mm.ss** for grid resolution. For example, 80:30:24N represents a northern latitude of 80 degrees, 30 minutes, and 24 seconds. 120:15W represents a longitute

120 degrees and 15 minutes west of the prime meridian. 30:15 represents a resolution of 30 degrees and 15 minutes. These next routines convert between ASCII representations and the machine representation for a coordinate. They work both with latitude-longitude projections and planimetric projections.

Note. In each subroutine, the programmer must specify the projection number. If the projection number is PROJECTION_LL,[13] then latitude-longitude ASCII format is invoked. Otherwise, a standard floating-point to ASCII conversion is made.

easting to ASCII

int
G_format_easting (double east, char *buf, int projection)

Converts the double representation of the **east** coordinate to its ASCII representation (into **buf**).

northing to ASCII

int
G_format_northing (double north, char *buf, int projection)

Converts the double representation of the **north** coordinate to its ASCII representation (into **buf**).

resolution to ASCII

int
G_format_resolution (double resolution, char *buf, int projection)

Converts the double representation of the **resolution** to its ASCII representation (into **buf**).

ASCII easting to double

int
G_scan_easting (char *buf, double *easting, int projection)

Converts the ASCII "easting" coordinate string in **buf** to its double representation (into **easting**).

ASCII northing to double

int

[13] Defined in "gis.h".

G_scan_northing (char *buf, double *northing, int projection)

Converts the ASCII "northing" coordinate string in **buf** to its double representation (into **northing**).

int
G_scan_resolution (char *buf, double *resolution, int projection)

ASCII resolution to double

Converts the ASCII "resolution" string in **buf** to its double representation (into resolution).

The following are examples of how these routines are used.

```
  double north ;
  char buf[50] ;
  G_scan_northing(buf, north, G_projection( )); /* ASCII to double
*/
  G_format_northing(north, buf, G_projection( )); /* double to
ASCII */
  G_format_northing(north, buf, -1); /* double to ASCII */
  /* This last example forces floating-point ASCII format */
```

12.8.2 Raster Area Calculations

The following routines perform area calculations for raster maps., They are based on the fact that while the latitude-longitude grid is not planimetric, the size of the grid cell at a given latitude is constant. The first routines work in any projection.

int
G_begin_cell_area_calculations (void)

begin cell area calculations

This routine must be called once before any call to *G_area_of_cell_at_row.* It can be used in either planimetric projections or the latitude-longitude projection. It returns 2 if the projection is latitude-longitude, 1 if the projection is planimetric, and 0 of the projection doesn't hav e a metric (e.g. imagery.) If the return value is 1 or 0, all the grid cells in the map have the same area. Otherwise the area of a grid cell varies with the row.

double
G_area_of_cell_at_row (int row)

cell area in specified row

This routine returns the area in square meters of a cell in the specified **row.** This value is constant for planimetric grids and varies with the row if the projection is latitude-longitude.

begin area calculations for ellipsoid

int
G_begin_zone_area_on_ellipsoid (double a, double e2, double s)

Initializes raster area calculations for an ellipsoid, where **a** is the semi-major axis of the ellipse (in meters), **e2** is the ellipsoid eccentricity squared, and **s** is a scale factor to allow for calculations of part of the zone (**s**=1.0 is full zone, **s**=0.5 is half the zone, and **s**=360/ew_res is for a single grid cell).
Note. e2 must be positive. A negative value makes no sense, and zero implies a sphere.

area between latitudes

double
G_area_for_zone_on_ellipsoid (double north, double south)

Returns the area between latitudes **north** and **south** scaled by the factor **s** passed to *G_begin_zone_area_on_ellipsoid.*

initialize calculations for sphere

int
G_begin_zone_area_on_sphere (double r, double s)

Initializes raster area calculations for a sphere. The radius of the sphere is **r** and **s** is a scale factor to allow for calculations of a part of the zone (see *G_begin_zone_area_on_ellipsoid).*

area between latitudes

double
G_area_for_zone_on_sphere (double north, double south)

Returns the area between latitudes **north** and **south** scaled by the factor **s** passed to *G_begin_zone_area_on_sphere.*

12.8.3 Polygonal Area Calculations

These next routines provide area calculations for polygons. Some of the routines are specifically for latitude-longitude, while others will function for all projections.

However, there is an issue for latitude-longitude that does not occur with planimetric grids. Vector/polygon data is described as a series of x,y coordinates. The lines connecting the points are not stored but are inferred. This is a simple, straight-forward process for planimetric grids, but it is not simple for latitude-longitude. What is the shape of the line that connects two points on the surface of a globe?

One choice (among many) is the shortest path from **x1,y1** to **x2,y2**, known as the geodesic. Another is a straight line on the grid. The area routines described below assume the latter. Routines to work with the former have not yet been developed.

int
G_begin_polygon_area_calculations (void)

begin polygon area calculations

This initializes the polygon area calculation routines. It is used both for planimetric and latitude-longitude projections.
It returns 2 if the projection is latitude-longitude, 1 if the projection is planimetric, and 0 if the projection doesn't hav e a metric (e.g. imagery.)

double
G_area_of_polygon (double *x, double *y, int n)

area in square meters of polygon

Returns the area in square meters of the polygon described by the **n** pairs of **x,y** coordinate vertices. It is used both for planimetric and latitude-longitude projections.
Note. If the database is planimetric with the non-meter grid, this routine performs the required unit conversion to produce square meters. double **G_planimetric_polygon_area** (x, y, n) *area in coordinate units* double *x, *y ; int n ;
Returns the area in coordinate units of the polygon described by the **n** pairs of **x,y** coordinate vertices for planimetric grids. If the units for **x,y** are meters, then the area is in square meters. If the units are feet, then the area is in square feet, and so on.

int
G_begin_ellipsoid_polygon_area (double a, double e2)

begin area calculations

This initializes the polygon area calculations for the ellipsoid with semi-major axis **a** (in meters) and ellipsoid eccentricity squared **e2.**

double
G_ellipsoid_polygon_area (double *lon, double *lat, int n)

area of lat-long polygon

Returns the area in square meters of the polygon described by the **n** pairs of **lat,long** vertices for latitude-longitude grids.
Note. This routine assumes grid lines on the connecting the vertices (as opposed to geodesics.)

12.8.4 Distance Calculations

Two routines perform distance calculations for any projection.

begin distance calculations
int
G_begin_distance_calculations (void)

Initializes the distance calculations. It is used both for the planimetric and latitude-longitude projections.
It returns 2 if the projection is latitude-longitude, 1 if the projection is planimetric, and 0 if the projection doesn't hav e a metric (e.g. imagery.)

distance in meters
double
G_distance (double x1, y1, x2, y2)

This routine computes the distance, in meters, from **x1,y1** to **x2,y2.** If the projection is latitude-longitude, this distance is measured along the geodesic. Two routines perform geodesic distance calculations.

begin geodesic distance
int
G_begin_geodesic_distance (double a, double e2)

Initializes the distance calculations for the ellipsoid with semi-major axis **a** (in meters) and ellipsoid eccentricity squared **e2.** It is used only for the latitude-longitude projection.

geodesic distance
double
G_geodesic_distance (double lon1, double lat1, double lon2, double lat2)

Calculates the geodesic distance from **lon1,lat1** to **lon2,lat2** in meters.
The calculation of the geodesic distance is fairly costly. These next three routines provide a mechanism for calculating distance with two fixed latitudes and varying longitude separation.

set geodesic distance lat1

int
G_set_geodesic_distance_lat1 (double lat1)

Set the first latitude.

set geodesic distance lat2

int
G_set_geodesic_distance_lat2 (double lat2)

Set the second latitude.

geodesic distance

double
G_geodesic_distance_lon_to_lon (double lon1, double lon2)

Calculates the geodesic distance from **lon1,lat1** to **lon2,lat2** in meters, where **lat1** was the latitude passed to *G_set_geodesic_distance_latl* and **lat2** was the *latitude passed to G_set_geodesic_distance_lat2*.

12.8.5 Global Wraparound

These next routines provide a mechanism for determining the relative position of a pair of longitudes. Since longitudes of ś360 are equivalent, but GRASS requires the east to be bigger than the west, some adjustment of coordinates is necessary.

returns east larger than west

double
G_adjust_easting (double east, struct Cell_head *region)

If the region projection is PROJECTION_LL, then this routine returns an equivalent **east** that is larger, but no more than 360 degrees larger, than the coordinate for the western edge of the region. Otherwise no adjustment is made and the original **east** is returned.

adjust east longitude

double
G_adjust_east_longitude (double east, double west)

This routine returns an equivalent **east** that is larger, but no more than 360 larger than the **west** coordinate.
This routine should be used only with latitude-longitude coordinates.

shortest way between eastings

int
G_shortest_way (double *east1, double *east2)

If the database projection is PROJECTION_LL, then **east1,east2** are changed so that they are no more than 180 degrees apart. Their true locations are not changed. If the database projection is not PROJECTION_LL, then **east1,east2** are not changed.

12.8.6 Miscellaneous

return ellopsoid name

char *
G_ellipsoid_name (int n)

This routine returns a pointer to a string containg the name for the **n***th* ellipsoid in the GRASS ellipsoid table; NULL when **n** is too large. It can be used as follows:

```
int n ;
char *name ;
for ( n=0 ; name=G_ellipsoid_name(n) ; n++ )
fprintf(stdout, "%s\n", name);
```

get ellipsoid by name

int
G_get_ellipsoid_by_name (char *name, double *a, double *e2)

This routine returns the semi-major axis **a** (in meters) and eccentricity squared **e2** for the named ellipsoid. Returns 1 if **name** is a known ellipsoid, 0 otherwise.

get ellipsoid parameters

int
G_get_ellipsoid_parameters (double *a, double *e2)

This routine returns the semi-major axis **a** (in meters) and the eccentricity squared **e2** for the ellipsoid associated with the database. If there is no ellipsoid explicitly associated with the database, it returns the values for the WGS 84 ellipsoid.

meridional radius of curvature

double
G_meridional_radius_of_curvature (double lon, double a, double e2)

Returns the meridional radius of curvature at a given longitude:

$$\rho = \frac{a(1-e^2)}{(1-e^2\sin^2 lon)^{3/2}}$$

double
G_transverse_radius_of_curvature (double lon, double a, double e2)

transverse radius of curvature

Returns the transverse radius of curvature at a given longitude:

$$\nu = \frac{a}{(1 - e^2 \sin^2 lon)^{1/2}}$$

double
G_radius_of_conformal_tangent_sphere (double lon, double a, double e2)

radius of conformal tangent sphere

Returns the radius of the conformal sphere tangent to ellipsoid at a given longitude:

$$r = \frac{a(1 - e^2)^{1/2}}{(1 - e^2 \sin^2 lon)}$$

int
G_pole_in_polygon (double *x, double *y, int n)

pole in polygon

For latitude-longitude coordinates, this routine determines if the polygon defined by the **n** coordinate vertices **x,y** contains one of the poles.
Returns -1 if it contains the south pole; 1 if it contains the north pole; 0 if it contains neither pole.
Note. Use this routine only if the projection is PROJECTION_LL.

12.9 Raster File Processing

Raster files are the heart and soul of GRASS. Because of this, a suite of routines which process raster file data has been provided. The processing of raster files consists of determining which raster file or files are to be processed (either by prompting the user or as specified on the module command line), locating the raster file in the database, opening the raster file, dynamically allocating i/o buffers, reading or writing the raster file, closing the raster file, and creating support files for newly created raster files.

All raster file data is of type CELL[14] , which is defined in "gis.h".

12.9.1 Prompting for Raster Files

The following routines interactively prompt the user for a raster file name. In each, the **prompt** string will be printed as the first line of the full prompt which asks the user to enter a raster file

[14] See *A.2 Appendix B: The CELL Data Type (p. 436)* for a discussion of the CELL type and how to use it (and avoid misusing it).

name. If **prompt** is the empty string "" then an appropriate prompt will be substituted. The name that the user enters is copied into the **name** buffer[15] These routines have a built-in 'list' capability which allows the user to get a list of existing raster files.

The user is required to enter a valid raster file name, or else hit the RETURN key to cancel the request. If the user enters an invalid response, a message is printed, and the user is prompted again. If the user cancels the request, the NULL pointer is returned. Otherwise the mapset where the raster file lives or is to be created is returned. Both the name and the mapset are used in other routines to refer to the raster file.

prompt for existing raster file

char *
G_ask_cell_old (char *prompt, char *name)

Asks the user to enter the name of an existing raster file in any mapset in the database.

prompt for existing raster file

char *
G_ask_cell_in_mapset (char *prompt, char *name)

Asks the user to enter the name of an existing raster file in the current mapset.

prompt for new raster file

char *
G_ask_cell_new (char *prompt, char *name)

Asks the user to enter a name for a raster file which does not exist in the current mapset.

Here is an example of how to use these routines. Note that the programmer must handle the NULL return properly:

```
  char *mapset;
  char name[50];
  mapset = G_ask_cell_old("Enter raster file to be processed",
name);
  if (mapset = = NULL)
  return(0);
```

[15] The size of **name** should be large enough to hold any GRASS file name. Most systems allow file names to be quite long. It is recommended that name be declared *char name.*

12.9.2 Finding Raster Files in the Database

Noninteractive modules cannot make use of the interactive prompting routines described above. For example, a command line driven module may require a raster file name as one of the command arguments. GRASS allows the user to specify raster file names (or any other database file) either as a simple unqualified name, such as "soils", or as a fully qualified name, such as "soils@*mapset*", where *mapset* is the mapset where the raster file is to be found. Often only the unqualified raster file name is provided on the command line.

The following routines search the database for raster files:

char * — *find a raster file*
G_find_cell (char *name, char *mapset)

Looks for the raster file **name** in the database. The **mapset** parameter can either be the empty string "", which means search all the mapsets in the user's current mapset search path,[16] or it can be a specific mapset name, which means look for the raster file only in this one mapset (for example, in the current mapset). If found, the mapset where the raster file lives is returned. If not found, the NULL pointer is returned.
If the user specifies a fully qualified raster file which exists, then *G_find_cell( )* modifies **name** by removing the "@*mapset*".
For example, to find a raster file anywhere in the database:

```
char name[50];
char *mapset;
if ((mapset = G_find_cell(name,"")) = = NULL)
/* not found */
```

To check that the raster file exists in the current mapset:

```
char name[50];
if (G_find_cell(name,G_mapset( )) = = NULL)
/* not found */
```

12.9.3 Opening an Existing Raster File

The following routine opens the raster file **name** in **mapset** for reading.

The raster file **name** and **mapset** can be obtained interactively using *G_ask_cell_old* or *G_ask_cell_in_mapset,*, and noninteractively using *G_find_cell*

int — *open an existing raster file*
G_open_cell_old (char *name, char *mapset)

[16] See *4.7.1 Mapset Search Path (p. 22)* for more details about the search path.

This routine opens the raster file **name** in **mapset** for reading. A nonnegative file descriptor is returned if the open is successful. Otherwise a diagnostic message is printed and a negative value is returned. This routine does quite a bit of work. Since GRASS users expect that all raster files will be resampled into the current region, the resampling index for the raster file is prepared by this routine after the file is opened. The resampling is based on the active module region.[17] Preparation required for reading the various raster file formats[18] is also done.

12.9.4 Creating and Opening New Raster Files

The following routines create the new raster file **name** in the current mapset[19] and open it for writing. The raster file **name** should be obtained interactively using *G_ask_cell_new*. If obtained noninteractively (e.g., from the command line), *G_legal_filename* should be called first to make sure that **name** is a valid GRASS file name.

Note. It is not an error for **name** to already exist. New raster files are actually created as temporary files and moved into the cell directory when closed. This allows an existing raster file to be read at the same time that it is being rewritten. The interactive routine *G_ask_cell_new* guarantees that **name** will not exist, but if **name** is obtained from the command line, **name** may exist. In this case *G_find_cell* could be used to see if **name** exists.

Warning. However, there is a subtle trap. The temporary file, which is created using *G_tempfile*, is named using the current process id. If the new raster file is opened by a parent process which exits after creating a child process using fork( ),[20] the raster file may never get created since the temporary file would be associated with the parent process, not the child. GRASS management automatically removes temporary files associated with processes that are no longer running. If fork( ) must be used, the safest course of action is to create the child first, then open the raster file. (See the discussion under *G_tempfile* for more details.)

open a new raster file (sequential)

FILE *
G_open_cell_new (char *name)

Creates and opens the raster file **name** for writing by *G_put_map_row* which writes the file row by row in sequential order. The raster file data will be compressed as it is written.
A nonnegative file descriptor is returned if the open is successful. Otherwise a diagnostic message is printed and a negative value is returned.

open a new raster file (random)

FILE *
G_open_cell_new_random (char *name)

[17] See also *12.7 The Region (p. 92)*.
[18] See *5.2 Raster File Format (p. 28)* for an explanation of the various raster file formats.
[19] GRASS does not allow files to be created outside the current mapset. See *4.7 Database Access Rules (p. 22)*.
[20] See also *G_fork*.

Creates and opens the raster file **name** for writing by *G_put_map_row_random* which allows writing the raster file in a random fashion. The file will be created uncompressed.[21]
A nonnegative file descriptor is returned if the open is successful. Otherwise a diagnostic message is printed and a negative value is returned.

FILE *
G_open_cell_new_uncompressed (char *name

open a new raster file (uncompressed)

Creates and opens the raster file **name** for writing by *G_put_map_row* which writes the file row by row in sequential order. The raster file will be in uncompressed format when closed.
A nonnegative file descriptor is returned if the open is successful. Otherwise a warning message is printed on stderr and a negative value is returned.
General use of this routine is not recommended.[22] This routine is provided so the *r.compress module* can create uncompressed raster files.

12.9.5 Allocating Raster I/O Buffers

Since there is no predefined limit for the number of columns in the region,[23] buffers which are used for reading and writing raster data must be dynamically allocated.

CELL *
G_allocate_cell_buf (void)

allocate a raster buffer

This routine allocates a buffer of type CELL just large enough to hold one row of raster data (based on the number of columns in the active region).

```
CELL *cell;
cell = G_allocate_cell_buf(void);
```

If larger buffers are required, the routine *G_malloc* can be used.
If sufficient memory is not available, an error message is printed and exit() is called.

int
G_zero_cell_buf (CELL *buf)

zero a raster buffer

[21] Nor will the file get automatically compressed when it is closed. If a compressed file is desired, it can be compressed explicitly after closing by a system call: system("r.compress *name*").

[22] At present, automatic raster file compression will create files which, in most cases, are smaller than if they were uncompressed. In certain cases, the compressed raster file may be larger. This can happen with imagery data, which do not compress well at all. However, the size difference is usually small. Since future enhancements to the compression method may improve compression for imagery data as well, it is best to create compressed raster files in all cases.

[23] See A.3 to find the number of columns in the region.

This routines assigns each member of the raster buffer array **buf** to zero. It assumes that **buf** has been allocated using *G_allocate_cell_buf*.

12.9.6 Reading Raster Files

Needs updating for GRASS 5!! See later in this file.

Raster data can be thought of as a two-dimensional matrix. The routines described below read one full row of the matrix. It should be understood, however, that the number of rows and columns in the matrix is determined by the region, not the raster file itself. Raster data is always read resampled into the region.[24] This allows the user to specify the coverage of the database during analyses. It also allows databases to consist of raster files which do not cover exactly the same area, or do not have the same grid cell resolution. When raster files are resampled into the region, they all "look" the same.

Note. The rows and columns are specified "C style", i.e., starting with 0.

THIS FUNCTION IS DEPRECATED IN GRASS 5! SEE NEXT CHAPTER!

read a raster file

int
G_get_map_row (int fd, CELL *cell, int row)

This routine reads the specified **row** from the raster file open on file descriptor **fd** (as returned by *G_open_cell_old*) into the **cell** buffer. The **cell** buffer must be dynamically allocated large enough to hold one full row of raster data. It can be *allocated using G_allocate_cell_buf*.
This routine prints a diagnostic message and returns -1 if there is an error reading the raster file. Otherwise a nonnegative value is returned.

read a raster file (without masking)

int
G_get_map_row_nomask (int fd, CELL *cell, int row)

This routine reads the specified **row** from the raster file open on file descriptor **fd** into the **cell** buffer like G_get_map_row( ) does. The difference is that masking is suppressed. If the user has a mask set, G_get_map_row( ) will apply the mask but G_get_map_row_nomask( ) will ignore it.
This routine prints a diagnostic message and returns -1 if there is an error reading the raster file. Otherwise a nonnegative value is returned.

[24] The GRASS region is discussed from a user perspective in *9.1 Region (p. 61)* and from a programmer perspective in *12.7 The Region (p. 92)* . The routines which are commonly used to determine the number of rows and columns in the region are *G_window_row* and *G_window_cols*.

Note. Ignoring the mask is not generally acceptable. Users expect the mask to be applied. However, in some cases ignoring the mask is justified. For example, the GRASS modules *r.describe*, which reads the raster file directly to report all data values in a raster file, and *r.slope.aspect*, which produces slope and aspect from elevation, ignore both the mask and the region. However, the number of GRASS modules which do this should be minimal. See *9.2 Mask (p. 63)* for more information about the mask.

12.9.7 Writing Raster Files

Needs updating for GRASS 5!! See later in this file.

The routines described here write raster file data.

int
G_put_map_row (int fd, CELL *buf) — *write a raster file (sequential)*

This routine writes one row of raster data from **buf** to the raster file open on file descriptor **fd.** The raster file must have been opened with *G_open_cell_new.*
The cell **buf** must have been allocated large enough for the region, perhaps using *G_allocate_cell_buf.*
If there is an error writing the raster file, a warning message is printed and -1 is returned. Otherwise 1 is returned.
Note. The rows are written in sequential order. The first call writes row 0, the second writes row 1, etc. The following example assumes that the raster file **name** is to be created:

```
int fd, row, nrows, ncols;
CELL *buf;
fd =G_open_cell_new(name);
if (fd < 0) ERROR}
buf = G_allocate_cell_buf();
ncols =G_window_cols();
nrows = G_window_rows();
for (row = 0; row < nrows; row++)
{
/* prepare data for this row into buf */
/* write the data for the row */
G_put_map_row(fd, buf);
}
```

int
G_put_map_row_random (int fd, CELL *buf, int row, int col, int ncells) — *write a raster file (random)*

This routine allows random writes to the raster file open on file descriptor **fd.** The raster file must have been opened using *G_open_cell_new_random.* The raster buffer **buf** contains **ncells** columns of data and is to be written into the raster file at the specified **row**, starting at column **col.**

12.9.8 Closing Raster Files

All raster files are closed by one of the following routines, whether opened for reading or for writing.

close a raster file

int
G_close_cell (int fd)

The raster file opened on file descriptor **fd** is closed. Memory allocated for raster processing is freed. If open for writing, skeletal support files for the new raster file are created as well.
Note. If a module wants to explicitly write support files (e.g., a specific color table) for a raster file it creates, it must do so after the raster file is closed. Otherwise the close will overwrite the support files. See *12.10 Raster Map Layer Support Routines (p. 112)* for routines which write raster support files.

unopen a raster file

int
G_unopen_cell (int fd)

The raster file opened on file descriptor **fd** is closed. Memory allocated for raster processing is freed. If open for writing, the raster file is not created and the temporary file created when the raster file was opened is removed (see *12.9.4 Creating and Opening New Raster Files (p. 108)*).
This routine is useful when errors are detected and it is desired to not create the new raster file. While it is true that the raster file will not be created if the module exits without closing the file, the temporary file will not be removed at module exit. GRASS database management will eventually remove the temporary file, but the file can be quite large and will take up disk space until GRASS does remove it. Use this routine as a courtesy to the user.

12.10 Raster Map Layer Support Routines

GRASS map layers have a number of support files associated with them. These files are discussed in detail in *5 Raster Maps (p. 27)*. The support files are the *raster header*, the *category* file, the *color* table, the *history* file, and the *range* file. Each support file has its own data structure and associated routines.

12.10.1 Raster Header File

The raster header file contains information describing the geographic extent of the map layer, the grid cell resolution, and the format used to store the data in the raster file. The format of this file is described in *5.3 Raster Header Format (p. 29)*. The routines described below use the *Cell_head* structure which is shown in detail in *12.21 GIS Library Data Structures (p. 212)*.

int
G_get_cellhd (char *name, char *mapset, struct Cell_Head *cellhd)

read the raster header

The raster header for the raster file **name** in the specified **mapset** is read into the **cellhd** structure.
If there is an error reading the raster header file, a diagnostic message is printed and -1 is returned. Otherwise, 0 is returned.

Note. If the raster file is a reclass file, the raster header for the referenced raster file is read instead. See *5.3.2 Reclass Format (p. 31)* for information about reclass files, and *G_is_reclass* for distinguishing reclass files from regular raster files.

Note. It is not necessary to get the raster header for a map layer in order to read the raster file data. The routines which read raster file data automatically retrieve the raster header information and use it for resampling the raster file data into the active region.[25] If it is necessary to read the raster file directly without resampling into the active region,[26] then the raster header can be used to set the active region using *G_set_window.*

char *
G_adjust_Cell_head (struct Cell_Head *cellhd, int rflag, int cflag)

adjust cell header

This function fills in missing parts of the input cell header (or region). It also makes projection-specific adjustments. The **cellhd** structure must have its *north, south, east, west*, and *proj* fields set. If **rflag** is true, then the north-south resolution is computed from the number of *rows* in the **cellhd** structure. Otherwise the number of *rows* is computed from the north-south resolution in the structure, similarly for **cflag** and the number of columns and the east-west resolution. This routine returns NULL if execution occurs without error, otherwise it returns an error message.

char *
G_put_cellhd (char *name, struct Cell_Head *cellhd)

write the raster header

This routine writes the information from the **cellhd** structure to the raster header file for the map layer **name** in the current mapset.

[25] See *12.7 The Region (p. 92)*.
[26] But see *9 Region and Mask (p. 61)* for a discussion of when this should and should not be done.

If there was an error creating the raster header, -1 is returned. No diagnostic is printed. Otherwise, 1 is returned to indicate success.

Note. Programmers should have no reason to use this routine. It is used by *G_close_cell* to giv e new raster files correct header files, and by the *r.support module* to give users a means of creating or modifying raster headers.

reclass file? int
G_is_reclass (char *name, char *mapset, char r_name, char **r_mapset)

This function determines if the raster file ***name** in **mapset** is a reclass file. If it is, then the name and mapset of the referenced raster file are copied into the **r_name** and **r_mapset** buffers.
Returns 1 if **name** is a reclass file, 0 if it is not, and -1 if there was a problem reading the raster header for **name.**

get child reclass maps list int
G_is_reclassed_to (char *name, char *mapset, int *nrmaps, char ***rmaps)

This function generates a child reclass maps list from the cell_misc/reclassed_to file which stores this list. The cell_misc/reclassed_to file is written by G_put_reclass().
G_is_reclassed_to() is used by g.rename, g.remove and r.reclass to prevent accidentally deleting the parent map of a reclassed raster map.

12.10.2 Raster Category File

GRASS map layers have category labels associated with them. The category file is structured so that each category in the raster file can have a one-line description. The format of this file is described in *5.4 Raster Category File Format (p. 32)*.

The routines described below manage the category file. Some of them use the *Categories* structure which is described in *12.21 GIS Library Data Structures (p. 212)*.

12.10.2.1 Reading and Writing the Raster Category File

The following routines read or write the category file itself:

read raster category file int
G_read_cats (char *name, char *mapset, struct Categories *cats)

The category file for raster file **name** in **mapset** is read into the **cats** structure. If there is an error reading the category file, a diagnostic message is printed and -1 is returned. Otherwise, 0 is returned.

int
G_write_cats (char *name, struct Categories *cats)

write raster category file

Writes the category file for the raster file **name** in the current mapset from the **cats** structure.
Returns 1 if successful. Otherwise, -1 is returned (no diagnostic is printed).

char *
G_get_cell_title (char *name, char *mapset)

get raster map title

If only the map layer title is needed, it is not necessary to read the entire category file into memory. This routine gets the title for raster file **name** in **mapset** directly from the category file, and returns a pointer to the title. A legal pointer is always returned. If the map layer does not have a title, then a pointer to the empty string "" is returned.

char *
G_put_cell_title (char *name, char *title)

change raster map title

If it is only desired to change the title for a map layer, it is not necessary to read the entire category file into memory, change the title, and rewrite the category file. This routine changes the **title** for the raster file **name** in the current mapset directly in the category file. It returns a pointer to the title.

12.10.2.2 Querying and Changing the Categories Structure

The following routines query or modify the information contained in the category structure:

char *
G_get_cat (CELL n, struct Categories *cats)

get a category label

This routine looks up category **n** in the **cats** structure and returns a pointer to a string which is the label for the category. A legal pointer is always returned. If the category does not exist in **cats,** then a pointer to the empty string "" is returned.
Warning. The pointer that is returned points to a hidden static buffer. Successive calls to G_get_cat( ) overwrite this buffer.

get title from category structure struct

char *
G_get_cats_title (Categories *cats)

Map layers store a one-line title in the category structure as well. This routine returns a pointer to the title contained in the **cats** structure. A legal pointer is always returned. If the map layer does not have a title, then a pointer to the empty string "" is returned.

initialize category structure

int
G_init_cats (CELL n, char *title, struct Categories *cats)

To construct a new category file, the structure must first be initialized. This routine initializes the **cats** structure, and copies the **title** into the structure. The number of categories is set initially to **n.**
For example:

```
struct Categories cats;
G_init_cats ( (CELL)0, "", &cats);
```

set a category label

int
G_set_cat (CELL n, char *label, struct Categories *cats)

The **label** is copied into the **cats** structure for category **n.**

set title in category structure

int
G_set_cats_title (char *title, struct Categories *cats)

The **title** is copied into the **cats** structure.

free category structure memory

int
G_free_cats (struct Categories *cats)

Frees memory allocated by*G_read_cats, G_init_cats* and *G_set_cat.*

12.10.3 Raster Color Table

GRASS map layers have colors associated with them. The color tables are structured so that each category in the raster file has its own color. The format of this file is described in *5.5 Raster Color Table Format (p. 33)*.

The routines that manipulate the raster color file use the *Colors* structure which is described in detail in *12.21 GIS Library Data Structures (p. 212)*.

12.10.3.1 Reading and Writing the Raster Color File

The following routines read, create, modify, and write color tables.

int
G_read_colors (char *name, char *mapset, struct Colors *colors)

read map layer color table

The color table for the raster file **name** in the specified **mapset** is read into the **colors** structure.
If the data layer has no color table, a default color table is generated and 0 is returned. If there is an error reading the color table, a diagnostic message is printed and -1 is returned. If the color table is read ok, 1 is returned.

int
G_write_colors (char *name, char *mapset, struct Colors *colors)

write map layer color table

The color table is written for the raster file **name** in the specified **mapset** from the **colors** structure.
If there is an error, -1 is returned. No diagnostic is printed. Otherwise, 1 is returned.
The **colors** structure must be created properly, i.e., *G_init_colors* to initialize the structure and *G_add_color_rule* to set the category colors.[27]
Note. The calling sequence for this function deserves special attention. The **mapset** parameter seems to imply that it is possible to overwrite the color table for a raster file which is in another mapset. However, this is not what actually happens. It is very useful for users to create their own color tables for raster files in other mapsets, but without overwriting other users' color tables for the same raster file. If **mapset** is the current mapset, then the color file for **name** will be overwritten by the new color table. But if **mapset** is not the current mapset, then the color table is actually written in the current mapset under the **colr2** element as: colr2/mapset/name.

12.10.3.2 Lookup Up Raster Colors

These routines translates raster values to their respective colors.

int
G_lookup_colors (CELL *raster, unsigned char *red, unsigned char *green, unsigned char *blue, set, int n, struct Colors *colors)

lookup an array of colors

[27] These routines are called by higher level routines which read or create entire color tables, such as*G_read_colors* or *G_make_ramp_colors.*

Extracts colors for an array of **raster** values. The colors for the **n** values in the **raster** array are stored in the **red, green**, and **blue** arrays. The values in the **set** array will indicate if the corresponding **raster** value has a color or not (1 means it does, 0 means it does not). The programmer must allocate the **red, green, blue**, and **set** arrays to be at least dimension **n.**
Note. The **red, green**, and **blue** intensities will be in the range 0 - 255.

get a category color
int
G_get_color (CELL cat, int *red, int *green, int *blue, struct Colors *colors)

The **red, green**, and **blue** intensities for the color associated with category **cat** are extracted from the **colors** structure. The intensities will be in the range 0 - 255.

12.10.3.3 Creating and/or Modifying the Color Table

These routines allow the creation of customized color tables as well as the modification of existing tables.

initialize color structure
int
G_init_colors (struct Colors *colors)

The **colors** structure is initialized for subsequent calls to *G_add_color_rule* and*G_set_color.*

set colors
int
G_add_color_rule (CELL cat1, int r1, int g1, int b1, CELL cat2, int r2, int g2, int b2, struct Colors *colors)

This is the heart and soul of the new color logic. It adds a color rule to the **colors** structure. The colors defined by the red, green, and blue values **r1,g1,b1** and **r2,g2,b2** are assigned to **cat1** and **cat2** respectively. Colors for data values between **cat1** and **cat2** are not stored in the structure but are interpolated when queried by *G_lookup_colors* and *G_get_color.* The color components **r1,g1,b1** and **r2,g2,b2** must be in the range 0 – 255.
For example, to create a linear grey scale for the range 200 – 1000:

```
  struct Colors colr;
  G_init_colors (&colr);
  G_add_color_rule ((CELL)200, 0,0,0, (CELL)1000,
255,255,255);
```

The programmer is encouraged to review *5.5 Raster Color Table Format (p. 33)* how this routine fits into the 5.x raster color logic.
Note. The **colors** structure must have been initialized by *G_init_colors*. See *12.10.3.4 Predefined Color Tables (p. 119)* for routines to build some predefined color tables.

int
G_set_color (CELL cat, int red, int green, int blue, struct Colors *colors)

set a category color

The **red, green**, and **blue** intensities for the color associated with category **cat** are set in the **colors** structure. The intensities must be in the range 0 -■ 255. Values below zero are set as zero, values above 255 are set as 255.
Use of this routine is discouraged because it defeats the new color logic. It is provided only for backward compatibility. Overuse can create large color tables. *G_add_color_rule* should be used whenever possible.
Note. The **colors** structure must have been initialized by *G_init_color*.

int
G_get_color_range

(

C

ELL *min, CELL *max, struct Colors *colors)

get color rangeGets the minimum and maximum raster values that have colors associated with them.

int
G_free_colors (struct Colors *colors)

free color structure memory

The dynamically allocated memory associated with the **colors** structure is freed.
Note. This routine may be used after *G_read_colors* as well as after *G_init_colors*.

12.10.3.4 Predefined Color Tables

The following routines generate entire color tables. The tables are loaded into a **colors** structure based on a range of category values from **min** to **max.** The range of values for a raster map can be obtained, for example, using *G_read_range*. **Note.** The color tables are generated without information about any particular raster file.

These color tables may be created for a raster file, but they may also be generated for loading graphics colors.

These routines return -1 if **min** is greater than **max**, 1 otherwise.

make aspect colors

int
G_make_aspect_colors (struct Colors *colors, CELL min, CELL max)

Generates a color table for aspect data.

make color ramp

int
G_make_ramp_colors (struct Colors *colors, CELL min, CELL max)

Generates a color table with 3 sections: red only, green only, and blue only, each increasing from none to full intensity. This table is good for continuous data, such as elevation.

make color wave

int
G_make_wave_colors (struct Colors *colors, CELL min, CELL max)

Generates a color table with 3 sections: red only, green only, and blue only, each increasing from none to full intensity and back down to none. This table is good for continuous data like elevation.

make linear grey scale

int
G_make_grey_scale_colors (struct Colors *colors, CELL min, CELL max)

Generates a grey scale color table. Each color is a level of grey, increasing from black to white.

make rainbow colors

int
G_make_rainbow_colors (struct Colors *colors, CELL min, CELL max)

Generates a "shifted" rainbow color table - yellow to green to cyan to blue to magenta to red. The color table is based on rainbow colors. (Normal rainbow colors are red, orange, yellow, green, blue, indigo, and violet.) This table is good for continuous data, such as elevation.

make random colors

int
G_make_random_colors (struct Colors *colors, CELL min, CELL max)

Generates random colors. Good as a first pass at a color table for nominal data.

make red,yellow,green colors

int
G_make_ryg_colors (struct Colors *colors, CELL min, CELL max)

Generates a color table that goes from red to yellow to green.

make green,yellow,red colors

int
G_make_gyr_colors (struct Colors *colors, CELL min, CELL max)

Generates a color table that goes from green to yellow to red.

make histogram-stretched grey colors

int
G_make_histogram_eq_colors (struct Colors *colors, struct Cell_stats *s)

Generates a histogram contrast-stretched grey scale color table that goes from the ,histogram information in the Cell_stats structure **s.** (See *12.10.5 Raster Histograms (p. 123)*).

12.10.3.4.1 Raster History File The history file contains documentary information about the raster file: who created it, when it was created, what was the original data source, what information is contained in the raster file, etc. This file is discussed in *5.6 Raster History File Format (p. 35)*

The following routines manage this file. They use the *History* structure which is described in *12.21 GIS Library Data Structures (p. 212)*.

Note. This structure has existed relatively unmodified since the inception of GRASS. It is in need of overhaul. Programmers should be aware that future versions of GRASS may no longer support either the routines or the data structure which support the history file.

read raster history file

int
G_read_history (char *name, char *mapset, struct History *history)

This routine reads the history file for the raster file **name** in **mapset** into the **history** structure.
A diagnostic message is printed and -1 is returned if there is an error reading the history file. Otherwise, 0 is returned.

write raster history file

int
G_write_history (char *name, struct History *history)

This routine writes the history file for the raster file **name** in the current mapset from the **history** structure.
A diagnostic message is printed and -1 is returned if there is an error writing the history file. Otherwise, 0 is returned.
Note. The **history** structure should first be initialized using *G_short_history*.

initialize history structure
int
G_short_history (char *name, char *type, struct History *history)

This routine initializes the **history** structure, recording the date, user, module name and the raster file **name** structure. The **type** is an anachronism from earlier versions of GRASS and should be specified as "raster".
Note. This routine only initializes the data structure. It does not write the history file.

12.10.4 Raster Range File

The following routines manage the raster range file. This file contains the minimum and maximum values found in the raster file. The format of this file is described in *5.7 Raster Range File Format (p. 36)*.

The routines below use the *Range* data structure which is described in *12.21 GIS Library Data Structures (p. 212)*.

read raster range
int
G_read_range (char *name, char *mapset, struct Range *range)

This routine reads the range information for the raster file **name** in **mapset** into the **range** structure.
A diagnostic message is printed and -1 is returned if there is an error reading the range file. Otherwise, 0 is returned.

write raster range file
int
G_write_range (char *name, struct Range *range)

This routine writes the range information for the raster file **name** in the current mapset from the **range** structure.
A diagnostic message is printed and -1 is returned if there is an error writing the range file. Otherwise, 0 is returned.

The range structure must be initialized and updated using the following routines:

int
G_init_range (struct Range *range)

initialize range structure

Initializes the **range** structure for updates by *G_update_range* and *G_row_update_range.*

int
G_update_range (CELL cat, struct Range *range)

update range structure

Compares the **cat** value with the minimum and maximum values in the **range** structure, modifying the range if **cat** extends the range.

int
G_row_update_range (CELL *cell, int n, struct Range *range)

update range structure

This routine updates the **range** data just like *G_update_range*, but for **n** values from the **cell** array.

The range structure is queried using the following routine:

int
G_get_range_min_max (struct Range *range, CELL *min, CELL *max)

get range min and max

The **min**inum and **max**imum CELL values are extracted from the **range** structure.

12.10.5 Raster Histograms

The following routines provide a relatively efficient mechanism for computing and querying a histogram of raster data. They use the *Cell_stats* structure to hold the histogram information. The histogram is a count associated with each unique raster value representing the number of times each value was inserted into the structure.

These next two routines are used to manage the Cell_stats structure:

int
G_init_cell_stats (struct Cell_stats *s)

initialize cell stats

This routine, which must be called first, initializes the Cell_stats structure **s.**

free cell stats

int
G_free_cell_stats (struct Cell_stats *s)

The memory associated with structure **s** is freed. This routine may be called any time after calling*G_init_cell_stats.*

This next routine stores values in the histogram:

add data to cell stats

int
G_update_cell_stats (CELL *data, int n, struct Cell_stats *s)

The **n** CELL values in the **data** array are inserted (and counted) in the Cell_stats structure **s.**

Once all values are stored, the structure may be queried either randomly (ie. search for a specific raster value) or sequentially (retrieve all raster values, in ascending order, and their related count):

random query of cell stats

int
G_find_cell_stat (CELL cat, long *count, struct Cell_stats *s)

This routine allows a random query of the Cell_stats structure **s.** The **count** associated with the raster value **cat** is set. The routine returns 1 if **cat** was found in the structure, 0 otherwise.

Sequential retrieval is accomplished using these next 2 routines:

reset/rewind cell stats

int
G_rewind_cell_stats (struct Cell_stats *s)

The structure **s** is rewound (i.e., positioned at the first raster category) so that sorted sequential retrieval can begin.

retrieve sorted cell stats

int
G_next_cell_stat (CELL *cat, long *count, struct Cell_stats *s)

Retrieves the next **cat,count** combination from the structure **s.** Returns 0 if there are no more items, non-zero if there are more.
For example:

```
struct Cell_stats s;
CELL cat;
long count;
.
.  /* updating s occurs here */
.
G_rewind_cell_stats(&s);
while (G_next_cell_stat(&cat,&count,&s)
fprintf(stdout, "%ld %ld\n", (long) cat, count);
```

12.11 GRASS 5 raster API [needs to be merged into above sections]

12.11.1 Changes to "gis.h"

The "gis.h" contains 5 new items:

```
typedef float FCELL
typedef double DCELL
typedef int RASTER_MAP_TYPE;
#define CELL_TYPE 0
#define FCELL_TYPE 1
#define DCELL_TYPE 2
```

Also "gis.h" contains the definitions for new structures:

```
struct FPReclass;
struct FPRange;
struct Quant;
```

Some of the old structures such as

```
struct Categories
struct Cell_stats;
struct Range;
struct _Color_Rule_;
struct _Color_Info_;
struct Colors;
```

were modified, so it is very important to use functional interface to access and set elements of these structures instead of accessing elements of the structures directly. Because some former elements such as for example (struct Range range.pmin) do not exist anymore. It

was made sure non of the former elements have different meaning, so that the programs which do access the old elements directly either do not compile or work exactly the same way as prior to change.

12.11.2 New NULL-value functions

Set NULL value
int
G_set_null_value (void *rast, int count, RASTER_MAP_TYPE data_type)

If the *data_type* is CELL_TYPE, calls G_set_c_null_value((CELL *) rast, count);
If the *data_type* is FCELL_TYPE, calls G_set_f_null_value((FCELL *) rast, count);
If the *data_type* is DCELL_TYPE, calls G_set_d_null_value((DCELL *) rast, count);

Set CELL NULL value
int
G_set_c_null_value (CELL *cell, int count)

Set the *count* elements in the *cell* array to the NULL value (the largest positive integer).

Set FCELL NULL value
int
G_set_f_null_value (FCELL *fcell, int count)

Set the *count* elements in the *fcell* array to the NULL value (a bit pattern for a `float` NaN - 32 bits of 1's).

Set CELL NULL value
int
G_set_d_null_value (DCELL *dcell, int count)

Set the *count* elements in the *dcell* array to the NULL value - which (a bit pattern for a `double` NaN - 64 bits of 1's).

Insert NULL value
int
G_insert_null_values (void *rast, char *flags, int count, RASTER_MAP_TYPE data_type)

If the *data_type* is CELL_TYPE, calls G_insert_c_null_values ((CELL *) rast, flags, count);
If the *data_type* is FCELL_TYPE, calls G_insert_f_null_values ((FCELL *) rast, flags, count);
If the *data_type* is DCELL_TYPE, calls G_insert_d_null_values ((DCELL *) rast, flags, count);

int
G_insert_c_null_values (CELL *cell, char *flags, int count) *Insert CELL NULL value*

For each of the *count flags* which is true(!=0), set the corresponding *cell* to the NULL value.

int
G_insert_f_null_values (FCELL *fcell, char *flags, int count) *Insert FCELL NULL value*

For each of the *count flags* which is true(!=0), set the corresponding *fcell* to the NULL value.

int
G_insert_d_null_values (DCELL *dcell, char *flags, int count) *Insert DCELL NULL value*

For each for the *count flag* which is true(!=0), set the corresponding *dcell* to the NULL value.

int
G_is_null_value (void *rast, RASTER_MAP_TYPE data_type)

If the *data_type* is CELL_TYPE, calls G_is_c_null_value ((CELL *) rast);
If the *data_type* is FCELL_TYPE, calls G_is_f_null_value ((FCELL *) rast);
If the *data_type* is DCELL_TYPE, calls G_is_d_null_value ((DCELL *) rast);

int
G_is_c_null_value (CELL *cell)

Returns 1 if *cell* is NULL, 0 otherwise. This will test if the value *cell* is the largest `int`.

int
G_is_f_null_value (FCELL *fcell)

Returns 1 if *fcell* is NULL, 0 otherwise. This will test if the value *fcell* is a NaN. It isn't good enough to test for a particular NaN bit pattern since the machine code may change this bit pattern to a different NaN. The test will be

```
  if(fcell==0.0) return 0; if(fcell>0.0) return 0;
if(fcell<0.0) return 0; return 1;
```

or (as suggested by Mark Line)

```
  return (fcell != fcell);
```

int
G_is_d_null_value (DCELL *dcell)

Returns 1 if *dcell* is NULL, 0 otherwise. This will test if the value *dcell* is a NaN. Same test as in `G_is_f_null_value()`.

char *
G_allocate_null_buf()

Allocate an array of char based on the number of columns in the current region.

int
G_get_null_value_row (int fd, char *flags, int row)

Reads a row from NULL value bitmap file for the raster map open for read on *fd*. If there is no bitmap file, then this routine simulates the read as follows: non-zero values in the raster map correspond to non-NULL; zero values correspond to NULL. When MASK exists, masked cells are set to null. *flags* is a resulting array of 0's and 1's where 1 corresponds to `"no data"` cell.

12.11.3 New Floating-point and type-independent functions

test for current mask

int
G_maskfd(void)

returns file descriptor number if MASK is in use and -1 if no MASK is in use.

int
G_raster_map_is_fp(char *name, char *mapset)

Returns true(1) if raster map *name* in *mapset* is a floating-point dataset; false(0) otherwise.

int
G_raster_map_type(char *name, char *mapset)

Returns the storage type for raster map *name* in *mapset*: `CELL_TYPE` (int); `FCELL_TYPE` (float); or `DCELL_TYPE` (double).

int
G_open_raster_new[_uncompressed](char *name, RASTER_MAP_TYPE map_type)

If map_type == CELL_TYPE, calls G_open_map_new[_uncompressed](name);
If map_type == FCELL_TYPE, calls G_set_fp_type (FCELL_TYPE); G_open_fp_map_new[_uncompressed](name);
If map_type == DCELL_TYPE, calls G_set_fp_type (DCELL_TYPE); G_open_fp_map_new[_uncompressed](name);
The use of this routine by applications is discouraged since its use would override user preferences (what precision to use).

int
G_set_fp_type (RASTER_MAP_TYPE type)

This controls the storage type for floating-point maps. It affects subsequent calls to `G_open_fp_map_new()`. The *type* must be one of `FCELL_TYPE` (float) or `DCELL_TYPE` (double). The use of this routine by applications is discouraged since its use would override user preferences.

int
G_open_fp_map_new (char *name)

Opens a new floating-point raster map (in `.tmp`) and returns a file descriptor. The storage type (`float` or `double`) is determined by the last call to `G_set_fp_type()` or the default (`float` - unless the Unix env variable GRASS_FP_DOUBLE is set).

void *
G_allocate_raster_buf(RASTER_MAP_TYPE data_type)

Allocate an array of CELL, FCELL, or DCELL (depending on *data_type*) based on the number of columns in the current region.

CELL *
G_allocate_c_raster_buf()

Allocate an array of CELL based on the number of columns in the current region.

FCELL *
G_allocate_f_raster_buf()

Allocate an array of FCELL based on the number of columns in the current region.

DCELL *
G_allocate_d_raster_buf()

Allocate an array of DCELL based on the number of columns in the current region.

void *
G_incr_void_ptr (void *ptr, int size)

Advances void pointer by n bytes. returns new pointer value. Usefull in raster row processing loops, substitutes

```
CELL *cell; cell += n;
```

Now

```
rast = G_incr_void_ptr(rast,
G_raster_size(data_type))
```

(where rast is void* and data_type is RASTER_MAP_TYPE can be used instead of rast++.) very usefull to generalize the row processing - loop (i.e. void * buf_ptr += G_raster_size(data_type)

int
G_raster_size (RASTER_MAP_TYPE data_type)

If *data_type* is CELL_TYPE, returns sizeof(CELL)
If *data_type* is FCELL_TYPE, returns sizeof(FCELL)
If *data_type* is DCELL_TYPE,q returns sizeof(DCELL)

int
G_raster_cmp (void *p, *q, RASTER_MAP_TYPE data_type)

Compares raster vlues p and q. Returns 1 if p > q or only q is null value -1 if p < q or only p is null value 0 if p == q or p==q==null value

int
G_raster_cpy (void *p, void *q, int n, RASTER_MAP_TYPE data_type)

Copies raster values q into p. If q is null value, sets q to null value.

int
G_set_raster_value_c (void *p, CELL val, RASTER_MAP_TYPE data_type)

If G_is_c_null_value(val) is true, sets p to null value. Converts CELL val to data_type (type of p) and stores result in p. Used for assigning CELL values to raster cells of any type.

int
G_set_raster_value_f (void *p, FCELL val, RASTER_MAP_TYPE data_type)

If G_is_f_null_value(val) is true, sets p to null value. Converts FCELL val to data_type (type of p) and stores result in p. Used for assigning FCELL values to raster cells of any type.

int
G_set_raster_value_d (void *p, DCELL val, RASTER_MAP_TYPE data_type)

If G_is_d_null_value(val) is true, sets p to null value. Converts DCELL val to data_type (type of p) and stores result in p. Used for assigning DCELL values to raster cells of any type.

CELL
G_get_raster_value_c (void *p, RASTER_MAP_TYPE data_type)

Retrieves the value of type data_type from pointer p, converts it to CELL type and returns the result. If null value is stored in p, returns CELL null value. Used for retreiving CELL values from raster cells of any type. NOTE: when data_type != CELL_TYPE, no quantization is used, only type conversion.

FCELL
G_get_raster_value_f (void *p, RASTER_MAP_TYPE data_type)

Retrieves the value of type data_type from pointer p, converts it to FCELL type and returns the result. If null value is stored in p, returns FCELL null value. Used for retreiving FCELL values from raster cells of any type.

DCELL
G_get_raster_value_d (void *p, RASTER_MAP_TYPE data_type)

Retrieves the value of type data_type from pointer p, converts it to DCELL type and returns the result. If null value is stored in p, returns DCELL null value. Used for retreiving DCELL values from raster cells of any type.

int
G_get_raster_row (int fd, void *rast, int row, RASTER_MAP_TYPE data_type)

If *data_type* is CELL_TYPE, calls G_get_c_raster_row(fd, (CELL *) rast, row);
If *data_type* is FCELL_TYPE, calls G_get_f_raster_row(fd, (FCELL *) rast, row);
If *data_type* is DCELL_TYPE, calls G_get_d_raster_row(fd, (DCELL *) rast, row);

int
G_get_raster_row_nomask (int fd, FCELL *fcell, int row, RASTER_MAP_TYPE map_type)

Same as `G_get_f_raster_row()` except no masking occurs.

int
G_get_f_raster_row (int fd, FCELL fcell, int row)

Read a row from the raster map open on *fd* into the `float` array *fcell* performing type conversions as necessary based on the actual storage type of the map. Masking, resampling into the current region. NULL-values are always embedded in `fcell` (*never converted to a value*).

int
G_get_f_raster_row_nomask (int fd, FCELL *fcell, int row)

Same as `G_get_f_raster_row()` except no masking occurs.

int
G_get_d_raster_row (int fd, DCELL *dcell, int row)

Same as `G_get_f_raster_row()` except that the array *dcell* is `double`.

int
G_get_d_raster_row_nomask (int fd, DECLL *dcell, int row)

Same as `G_get_d_raster_row()` except no masking occurs.

int
G_get_c_raster_row (int fd, CELL buf, int row)

Reads a row of raster data and leaves the NULL values intact. (As opposed to the deprecated function `G_get_map_row()` which converts NULL values to zero.) **NOTE.** When the raster map is old and null file doesn't exist, it is assumed that all 0-cells are no-data. When map is floating point, uses quant rules set explicitly by G_set_quant_rules or stored in map's quant file to convert floats to integers.

int
G_get_c_raster_row_nomask (int fd, CELL buf, int row)

Same as `G_get_c_raster_row()` except no masking occurs.

int
G_put_raster_row (int fd, void *rast, RASTER_MAP_TYPE data_type)

If *data_type* is CELL_TYPE, calls G_put_c_raster_row(fd, (CELL *) rast);
If *data_type* is FCELL_TYPE, calls G_put_f_raster_row(fd, (FCELL *) rast);
If *data_type* is DCELL_TYPE, calls G_put_d_raster_row(fd, (DCELL *) rast);

int
G_put_f_raster_row (int fd, FCELL *fcell)

Write the next row of the raster map open on *fd* from the `float` array *fcell*, performing type conversion to the actual storage type of the resultant map. Keep track of the range of floating-point values. Also writes the NULL-value bitmap from the NULL-values embedded in the *fcell* array.

int
G_put_d_raster_row (int fd, DCELL *dcell)

Same as `G_put_f_raster_row()` except that the array *dcell* is `double`.

int
G_put_c_raster_row (int fd, CELL buf)

Writes a row of raster data and a row of the null-value bitmap, only treating NULL as NULL. (As opposed to the deprecated function `G_put_map_row()` which treats zero values also as NULL.)

int
G_zero_raster_row (void *rast, RASTER_MAP_TYPE data_type)

Depending on *data_type* zeroes out G_window_cols() CELLs, FCELLs, or DCELLs stored in cell buffer.

12.11.4 Upgrades to Raster Functions (comparing to GRASS 4.x)

These routines will be modified (internally) to work with floating-point and NULL-values.

Changes to `GISLIB`:

int
G_close_cell()

If the map is a new floating point, move the `.tmp` file into the `fcell` element, create an empty file in the `cell` directory; write the floating-point range file; write a default quantization file quantization file is set here to round fp numbers (this is a default for now). create an empty category file, with max cat = max value (for backwards compatibility). Move the `.tmp` NULL-value bitmap file to the `cell_misc` directory.

int
G_open_cell_old()

Arrange for the NULL-value bitmap to be read as well as the raster map. If no NULL-value bitmap exists, arrange for the production of NULL-values based on zeros in the raster map.

If the map is floating-point, arrange for quantization to integer for `G_get_c_raster_row()`, et. al., by reading the quantization rules for the map using `G_read_quant()`.

If the programmer wants to read the floating point map using uing quant rules other than the ones stored in map's quant file, he/she should call G_set_quant_rules() after the call to G_open_cell_old().

int
G_get_map_row()

If the map is floating-point, quantize the floating-point values to integer using the quantization rules established for the map when the map was opened for reading (this quantization is read from cell_misc/name/f_quant file, but can be reset after opening raster map by G_set_quant_rules()).

NULL values are converted to zeros.

This routine is deprecated!!

int
G_put_map_row()

Zero values are converted to NULLs. Write a row of the NULL value bit map.

This routine is deprecated!!

Changes to `D_LIB`:

int
Dcell()

If the map is a floating-point map, read the map using `G_get_d_map_row()` and plot using `D_draw_d_cell()`. If the map is an integer map, read the map using `G_get_c_raster_row()` and plot using `D_draw_cell()`.

12.11.5 Color Functions (new and upgraded)

12.11.5.1 Upgraded Colors structures

```
  struct _Color_Rule_
  struct
  DCELL value;
  unsigned char red,grn,blu;
  low, high;
  struct _Color_Rule_ *next;
  struct _Color_Rule_ *prev; ;
struct _Color_Info_
  struct _Color_Rule_ *rules;
  int n_rules;
  struct
  unsigned char *red;
  unsigned char *grn;
  unsigned char *blu;
  unsigned char *set;
  int nalloc;
  int active;
  lookup;
  struct
  DCELL *vals;
  /* pointers to color rules corresponding to the intervals between
vals */
  struct _Color_Rule_ **rules;
  int nalloc;
  int active;
  fp_lookup;
  DCELL min, max;
  ;
struct Colors
  int version; /* set by read_colors:  -1=old,1=new */
  DCELL shift;
  int invert;
  int is_float; /* defined on floating point raster data?  */
  int null_set; /* the colors for null are set?  */
  unsigned char null_red, null_grn, null_blu;
  int undef_set; /* the colors for cells not in range are set?
*/
  unsigned char undef_red, undef_grn, undef_blu;
  struct _Color_Info_ fixed, modular;
  DCELL cmin, cmax;
  ;
```

12.11.5.2 New functions to support colors for floating-point

Changes to `GISLIB`:

int
G_lookup_raster_colors (void *rast, char *r, char *g, char *b, char *set, int n, struct Colors *colors, RASTER_MAP_TYPE cell_type)

If the *cell_type* is CELL_TYPE, calls G_lookup_colors((CELL *)cell, r, g, b, set, n, colors);
If the *cell_type* is FCELL_TYPE, calls G_lookup_f_raster_colors(FCELL *)cell, r, g, b, set, n, colors);
If the *cell_type* is DCELL_TYPE, calls G_lookup_d_raster_colors(DCELL *)cell, r, g, b, set, n, colors);

int
G_lookup_c_raster_colors (CELL *cell, char *r, char *g, char *b, char *set, int n, struct Colors *colors)

The same as G_lookup_colors(cell, r, g, b, set, n, colors).

int
G_lookup_f_raster_colors (FCELL *fcell, char *r, char *g, char *b, char *set, int n, struct Colors *colors)

Converts the *n* floating-point values in the *fcell* array to their *r,g,b* color components. Embedded NULL-values are handled properly as well.

int
G_lookup_d_raster_colors (DCELL *dcell, char *r, char *g, char *b, char *set, int n, struct Colors *colors)

Converts the *n* floating-point values in the *dcell* array to their *r,g,b* color components. Embedded NULL-values are handled properly as well.

int
G_add_raster_color_rule (void *v1, int r1, int g1, int b1, void *v2, int r2, int g2, int b2, struct Colors *colors, RASTER_MAP_TYPE map_type)

If *map_type* is CELL_TYPE, calls G_add_c_raster_color_rule ((CELL *) v1, r1, g1, b1, (CELL *) v2, r2, g2, b2, colors);
If *map_type* is FCELL_TYPE, calls G_add_f_raster_color_rule ((FCELL *) v1, r1, g1, b1, (FCELL *) v2, r2, g2, b2, colors);
If *map_type* is DCELL_TYPE, calls G_add_d_raster_color_rule ((DCELL *) v1, r1, g1, b1, (DCELL *) v2, r2, g2, b2, colors);

int
G_add_c_raster_color_rule (CELL *v1, int r1, int g1, int b1, CELL *v2, int r2, int g2, int b2, struct Colors *colors)

Calls G_add_color_rule(*v1, r1, g1, b1, *v2, r2, g2, b2, colors).

int
G_add_f_raster_color_rule (FCELL *v1, int r1, int g1, int b1, FCELL *v2, int r2, int g2, int b2, struct Colors *colors)

Adds the floating-point rule that the range [*v1,v2*] gets a linear ramp of colors from [*r1,g1,b1*] to [*r2,g2,b2*].
If either *v1* or *v2* is the NULL-value, this call is converted into `G_set_null_value_color (r1, g1, b1, colors)`

int
G_add_d_raster_color_rule (DCELL *v1, int r1, int g1, int b1, DCELL *v2, int r2, int g2, int b2, struct Colors *colors)

Adds the floating-point rule that the range [*v1,v2*] gets a linear ramp of colors from [*r1,g1,b1*] to [*r2,g2,b2*].
If either *v1* or *v2* is the NULL-value, this call is converted into `G_set_null_value_color (r1, g1, b1, colors)`

int
G_get_raster_color (void *v, int *r, int *g, int *b, struct Colors *colors, RASTER_MAP_TYPE data_type)

Looks up the rgb colors for *v* in the color table *colors*

int
G_get_c_raster_color (CELL *v, int *r, int *g, int *b, struct Colors *colors)

Calls G_get_color(*v, r, g, b, colors).

int
G_get_f_raster_color (FCELL *v, int *r, int *g, int *b, struct Colors *colors)

Looks up the rgb colors for *v* in the color table *colors*

int
G_get_d_raster_color (DCELL *v, int *r, int *g, int *b, struct Colors *colors)

Looks up the rgb colors for *v* in the color table *colors*

int
G_set_raster_color (void *v, int r, int g, int b, struct Colors *colors, RASTER_MAP_TYPE data_type)

calls `G_add_raster_color_rule (v, r, g, b, v, r, g, r, colors, data_type);`

int
G_set_c_raster_color (CELL *v, int r, int g, int b, struct Colors *colors)

Calls G_set_color(*v, r, g, b, colors).

int
G_set_f_raster_color (FCELL *v, int r, int g, int b, struct Colors *colors)

Inserts a rule that assigns the color *r,g,b* to *v*. It is implemented as:
`G_add_f_raster_color_rule (v, r, g, b, v, r, g, r, colors);`

int
G_set_d_raster_color (DCELL *v, int r, int g, int b, struct Colors *colors)

Inserts a rule that assigns the color *r,g,b* to *v*. It is implemented as:
`G_add_d_raster_color_rule (v, r, g, b, v, r, g, r, colors);`

int
G_mark_colors_as_fp (struct Colors *colors)

Sets a flag in the *colors* structure that indicates that these colors should only be looked up using floating-point raster data (not integer data).
In particular if this flag is set, the routine `G_get_colors_min_max()` should return min=-255^3 and max=255^3.

These routines are in the `DISPLAYLIB`:

int
D_raster_of_type (void *rast, int ncols, int nrows, struct Colors *colors, RASTER_MAP_TYPE data_type)

If *map_type* is CELL_TYPE, calls D_raster((CELL *) rast, ncols, nrows, colors);
If *map_type* is FCELL_TYPE, calls D_f_raster((FCELL *) rast, ncols, nrows, colors);
If *map_type* is DCELL_TYPE, calls D_d_raster((DCELL *) rast, ncols, nrows, colors);

int
D_f_raster (FCELL *fcell, int ncols, int nrows, struct Colors *colors)

Same functionality as `D_raster()` except that the *fcell* array is type `FCELL`. This implies that the floating-point interfaces to the *colors* are used by this routine.

int
D_d_raster (DCELL *dcell, int ncols, int nrows, struct Colors *colors)

Same functionality as `D_raster()` except that the *dcell* array is type `DCELL`. This implies that the floating-point interfaces to the *colors* are used by this routine.

int
D_color_of_type (void *value, struct Colors *colors, RASTER_MAP_TYPE data_type)

If the *data_type* is CELL_TYPE, calls D_color((CELL *value, colors);
If the *data_type* is FCELL_TYPE, calls D_f_color((FCELL *value, colors);
If the *data_type* is DCELL_TYPE, calls D_d_color((DCELL *value, colors);

int
D_f_color (FCELL *value, struct Colors *colors)

Same functionality as `D_color()` except that the *value* is type `FCELL`. This implies that the floating-point interfaces to the *colors* are used by this routine.

int
D_d_color (DCELL *value, struct Colors *colors)

Same functionality as `D_color()` except that the *value* is type `DCELL`. This implies that the floating-point interfaces to the *colors* are used by this routine.

int
D_lookup_raster_colors (void *rast, int *colornum, int n, struct Colors *colors, RASTER_MAP_TYPE data_type)

If the *data_type* is CELL_TYPE, calls D_lookup_c_raster_colors((CELL *) rast, colornum, n, colors);
If the *data_type* is FCELL_TYPE, calls D_lookup_f_raster_colors((FCELL *) rast, colornum, n, colors);
If the *data_type* is DCELL_TYPE, calls D_lookup_d_raster_colors((DCELL *) rast, colornum, n, colors);

int
D_lookup_c_raster_colors (CELL *cell, int *colornum, int n, struct Colors *colors)

Same functionality as `D_lookup_colors()` except that the resultant color numbers are placed into a separate *colornum* array (which the caller must allocate).

int
D_lookup_f_raster_colors (FCELL *fcell, int *colornum, int n, struct Colors *colors)

Same functionality as `D_lookup_colors()` except that the *fcell* array is type `FCELL` and that the resultant color numbers are placed into a separate *colornum* array (which the caller must allocate).

int
D_lookup_d_raster_colors (DCELL *dcell, int *colornum, int n, struct Colors *colors)

Same functionality as `D_lookup_colors()` except that the *dcell* array is type `DCELL` and that the resultant color numbers are placed into a separate *colornum* array (which the caller must allocate).

int
D_draw_cell_of_type(int A_row, DCELL *xarray, struct Colors *colors, RASTER_MAP_TYPE map_type)

If *map_type* is CELL_TYPE, calls D_draw_cell (A_row, (CELL *) xarray, colors);
If *map_type* is FCELL_TYPE, calls D_draw_f_cell (A_row, (FCELL *) xarray, colors);
If *map_type* is DCELL_TYPE, calls D_draw_d_cell (A_row, (DCELL *) xarray, colors);

int
D_draw_f_cell (int A_row, FCELL *xarray, struct Colors *colors)

Same functionality as `D_draw_cell()` except that the *xarray* array is type `FCELL` which implies a call to `D_f_raster()` instead of a call to `D_raster()`.

int
D_draw_d_cell (int A_row, DCELL *xarray, struct Colors *colors)

Same functionality as `D_draw_cell()` except that the *xarray* array is type `DCELL` which implies a call to `D_d_raster()` instead of a call to `D_raster()`.

12.11.5.3 New functions to support a color for the NULL-value

int
G_set_null_value_color (int r, int g, int b, struct Colors *colors)

Sets the color (in *colors*) for the NULL-value to *r,g,b*.

int
G_get_null_value_color (int *r, int *g, int *b, struct Colors *colors)

Puts the red, green, and blue components of the color for the NULL-value into *r,g,b*.

12.11.5.4 New functions to support a default color

int
G_set_default_color (int r, int g, int b, struct Colors *colors)

Sets the default color (in *colors*) to *r,g,b*. This is the color for values which do not have an explicit rule.

int
G_get_default_color (int *r, int *g, int *b, struct Colors *colors)

Puts the red, green, and blue components of the `"default"` color into *r,g,b*.

12.11.5.5 New functions to support treating a raster layer as a color image

int
G_get_raster_row_colors(int fd, int row, struct Colors *colors, unsigned char *red, unsigned char *grn, unsigned char *blu, unsigned char *nul)

Reads a row of raster data and converts it to red, green and blue components according to the *colors* parameter.
This provides a convenient way to treat a raster layer as a color image without having to explictly cater for each of `CELL`, `FCELL` and `DCELL` types

12.11.5.6 Upgraded color functions

int
G_read_colors()

This routine reads the rules from the color file. If the input raster map is is a floating-point map it calls `G_mark_colors_as_fp()`.

int
G_write_colors()

The rules are written out using floating-point format, removing trailing zeros (possibly producing integers). The flag marking the colors as floating-point is **not** written.

int
G_get_colors_min_max()

If the color table is marked as `"float"`, then return the minimum as -(255^3 * 128) and the maximum as (255^3 * 128). This is to simulate a very **large** range so that GRASS doesn't attempt to use *colormode float* to allow interactive toggling of colors.

int
G_lookup_colors()

Modified to return a color for NULL-values.

int
G_get_color()

Modified to return a color for the NULL-value.

12.11.5.7 Changes to the `Colors` structure

Modifications to the `Colors` structure to support colors for floating-point data and the NULL-value consist of

- the _Color_Rule_ struct was changed to have DCELL value (instead of CELL cat) to have the range be floating-point values instead of integer cats.
- a color for NULL was added
- the special color for zero was eliminated
- a default color for values which have no assigned color was added
- a flag was added to the Colors structure to indicate if either the map itself is floating-point (If the map is integer and the floating point functions are used to lookup colors, the values are checked to see if they are integer, and if they are, the integer mechanism is used)
- fp_lookup - a lookup table for floating point numbers is added. It orders the end points of fp intervals into array with a pointer to a color rule for each inteval, and the binary search is then used when looking up colors instead of linearly searching through all color rules.

12.11.5.8 Changes to the `colr` file

- The rules are written out using floating-point format, removing trailing zeros (possibly producing integers). For example, to ramp from red to green for the range [1.3,5.0]:

```
1.3:255:0:0  5:0:255:0
```

- The NULL-value color is written as:

```
nv:red:grn:blu
```

- The default color (for values that don't have an explicit rule) is written as:

```
*:red:grn:blu
```

12.11.6 Range functions (new and upgraded)

12.11.6.1 Modified range functions

int
G_read_range()

Old range file (those with 4 numbers) should treat zeros in this file as NULL-values. New range files (those with just 2 numbers) should treat these numbers as real data (zeros are real data in this case).
An empty range file indicates that the min, max are undefined. This is a valid case, and the result should be an initialized range struct with no defined min/max.
If the range file is missing and the map is a floating-point map, this function will create a default range by calling `G_construct_default_range()`.

int
G_init_range()

Must set a flag in the range structure that indicates that no min/max have been defined - probably a `"first"` boolean flag.

int
G_update_range()

NULL-values must be detected and ignored.

int
G_get_range_min_max()

If the range structure has no defined min/max (first!=0) there will not be a valid range. In this case the min and max returned must be the NULL-value.

int
G_write_range()

This routine only writes 2 numbers (min,max) to the range file, instead of the 4 (pmin,pmax,nmin,nmax) previously written. If there is no defined min,max, an empty file is written.

12.11.6.2 New range functions

int
G_construct_default_range (struct Range *r)

Sets the integer range *r* to [1,255]

int
G_read_raster_range (void *r, char *name, char *mapset, RASTER_MAP_TYPEmap_type)

If *map_type* is CELL_TYPE, calls G_read_range((struct Range *) r, name, mapset); otherwise calls G_read_fp_range((struct FPRange *) r, name, mapset);

int
G_read_fp_range (struct FPRange *r, char *name, char *mapset)

Read the floating point range file `f_range`. This file is written in binary using XDR format. If there is no defined min/max in *r*, an empty `f_range`file is created. An empty range file indicates that the min, max are undefined. This is a valid case, and the result should be an initialized range struct with no defined min/max.
If the range file is missing and the map is a floating-point map, this function will create a default range by calling `G_construct_default_range()`.

int
G_init_raster_range (FPRange *r, RASTER_MAP_TYPE map_type)

If *map_type* is CELL_TYPE, calls G_init_range(struct Range *) r); otherwise calls G_init_fp_range((struct FPRange *) r);

int
G_init_fp_range (FPRange *r)

Must set a flag in the range structure that indicates that no min/max have been defined - probably a "first" boolean flag.

int
G_update_f_range (FPRange *r, FCELL *fcell, int n)

Updates the floating-point range *r* from the *n* FCELL values in *fcell* NULL-values must be detected and ignored.

int
G_update_d_range (FPRange *r, DCELL *dcell, int n)

Updates the floating-point range *r* from the *n* DCELL values in *dcell* NULL-values must be detected and ignored.

int
G_get_fp_range_min_max (FPRange *r, DCELL *min, DCELL *max)

Extract the min/max from the range structure *r*.
If the range structure has no defined min/max (first!=0) there will not be a valid range. In this case the min and max returned must be the NULL-value.

int
G_write_fp_range (FPRange *r)

Write the floating point range file f_range. This file is written in binary using XDR format. If there is no defined min/max in *r*, an empty f_rangefile is created.

12.11.7 New and Upgraded Cell_stats functions

Modified `Cell_stats` functions to handle NULL-values:

int
G_init_cell_stats()

Set the count for NULL-values to zero.

int
G_update_cell_stats()

Look for NULLs and update the NULL-value count.

int
G_next_cell_stat()

Do not return a record for the NULL-value

int
G_find_cell_stat()

Allow finding the count for the NULL-value

int
G_get_stats_for_null_value(int *count, struct Cell_stats *s)

Get a number of null values from stats structure. Note: when reporting values which appear in a map using G_next_cell_stats(), to get stats for null, call G_get_stats_for_null_value() first, since G_next_cell_stats() does not report stats for null.

12.11.8 New Quantization Functions

New functions to support quantization of floating-point to integer:

int
G_write_quant (char *name, char *mapset, struct Quant *q)

Writes the `f_quant` file for the raster map *name* from *q*.
if mapset==G_mapset() i.e. the map is in current mapset, then the original quant file in cell_misc/map/f_quant is written. Otherwise *q* is written into quant2/mapset/name (much like colr2 element). This results in map@mapset being read using quant rules stored in *q* from G_mapset(). See G_read_quant() for detailes.

int
G_set_quant_rules (int fd, struct Quant *q)

Sets quant translation rules for raster map opened for reading. fd is a file descriptor returned by G_open_cell_old(). After calling this function, G_get_c_raster_row() and G_get_map_row() will use rules defined by q (instead of using rules defined in map's quant file) to convert floats to ints.

int
G_read_quant (char *name, char *mapset, struct Quant *q)

reads quantization rules for `"name"` in `"mapset"` and stores them in the quantization structure `"quant"`. If the map is in another mapset, first checks for quant2 table for this map in current mapset.
Return codes:
-2 if raster map is of type integer
-1 if (! G__name_is_fully_qualified ())
0 if quantization file does not exist, or the file is empty or has wrong format.
1 if non-empty quantization file exists.

int
G_quant_init (struct Quant *q)

Initializes the *q* struct.

int
G_quant_free (struct Quant *q)

Frees any memory allocated in *q* and re-initializes *q* by calling `G_quant_init()`.

int
G_quant_truncate (struct Quant *q)

sets the quant for *q* rules to perform simple truncation on floats.

int
G_quant_truncate (struct Quant *q)

sets the quant for *q* rules to perform simple rounding on floats.

int
G_quant_organize_fp_lookup (struct Quant *quant)

Organizes fp_lookup table for faster (logarithmic) lookup time G_quant_organize_fp_lookup() creates a list of min and max for each quant rule, sorts this list, and stores the pointer to quant rule that should be used inbetween any 2 numbers in this list Also it stores extreme points for 2 infinite rules, if exist After the call to G_quant_organize_fp_lookup() instead of linearly searching through list of rules to find a rule to apply, quant lookup will perform a binary search to find an interval containing floating point value, and then use the rule associated with this interval. when the value doesn't fall within any interval, check for the infinite rules.

int
G_quant_add_rule (struct Quant *q, DCELL dmin, DCELL dmax, CELL cmin, CELL cmax)

Add the rule that the floating-point range *[dmin,dmin]* produces an integer in the range *[cmin,cmax]* by linear interpolation.
Rules that are added later have higher precedence when searching.
If any of of *dmin*, *dmax cmin*, or *cmax* is the NULL-value, this rule is not added and 0 is returned. Otherwise return 1. if the fp_lookup is organized, destroy it.

int
G_quant_set_positive_infinite_rule (struct Quant *q, DCELL dmax, CELL c)

Set the rule that values greater than or equal to *dmax* produce the integer *c*. If *dmax* or *c* is the NULL-value, return 0 and don't set the rule. Otherwise return 1.
This rule has lower precedence than rules added with `G_quant_add_rule()`.

int
G_quant_get_positive_infinite_rule (struct Quant *q, DCELL *dmax, CELL *c)

Sets *dmax* and *c* to the positive `"infinite"` rule in *q* if there is one and returns 1. If there is no such rule, it just returns 0. if the fp_lookup is organized, updates infinite limits.

int
G_quant_set_negative_infinite_rule (struct Quant *q, DCELL dmin, CELL c)

Set the rule that values less than or equal to *dmin* produce the integer *c*. If *dmin* or *c* is the NULL-value, return 0 and don't set the rule. Otherwise return 1. if the fp_lookup is organized, updates infinite limits.
This rule has lower precedence than rules added with `G_quant_add_rule()`.

int
G_quant_get_negative_infinite_rule (struct Quant *q, DCELL *dmin, CELL *c)

Sets *dmin* and *c* to the negative `"infinite"` rule in *q* if there is one and returns 1. If there is no such rule, it just returns 0.

int
G_quant_get_limits (struct Quant *q, DCELL *dmin, DCELL *dmax, CELL *cmin, CELL *cmax)

Extracts the minimum and maximum floating-point and integer values from all the rules (except the `"infinite"` rules) in *q* into *dmin*, *dmax*, *cmin*, and *cmax*. Returns 1 if there are any explicit rules. If there are no explicit rules, (this includes cases when q is set to truncate or round map), it returns 0 and sets *dmin*, *dmax*, *cmin*, and *cmax* to NULL.

int
G_quant_nrules (struct Quant *q)

Returns the number of rules in *q*, excluding the negative and positive `"infinite"` rules.

int
G_quant_get_rule (struct Quant *q, int n, DCELL *dmin, DCELL *dmax, CELL *cmin, CELL *cmax)

Get the *n*th rule from *q*. If 0 <= *n* < nrules(q), extract the rule and return 1. Otherwise return 0. This function can't be used to get the "infinite" rules.
The order of the rules returned by increasing *n* is the order in which the rules are applied when quantizing a value - the first rule applicable is used.

CELL
G_quant_get_cell_value (struct Quant *q, DCELL value)

Returns a CELL category for the floating-point *value* based on the quantization rules in *q*. The first rule found that applies is used. The rules are searched in the reverse order they are added to *q*. If no rule is found, the *value* is first tested against the negative infinite rule, and finally against the positive infinite rule. if none of these rules apply, the NULL-value is returned.
NOTE. See G_quant_organize_fp_lookup() for details on how the values are looked up from fp_lookup table when it is active. (Right now fp_lookup is automatically organized during the first call to G_quant_get_cell_value()

int
G_quant_perform_d (struct Quant *q, DCELL *dcell, CELL *cell, int n)

Performs a quantization of the *n* DCELL values in the *dcell* array and puts the results into the *cell* array.

int
G_quant_perform_f (struct Quant *q, FCELL *fcell, CELL *cell, int n)

Performs a quantization of the *n* FCELL values in the *fcell* array and puts the results into the *cell* array.

These next two functions are convenience functions to allow applications to easily create quantization rules other than the defaults:

int
G_quantize_fp_map (char *name, CELL cmin, CELL cmax)

Writes the f_quant file for the raster map *name* with one rule. The rule is generated using the floating-point range in f_range producing the integer range [*cmin,cmax*].

int
G_quantize_fp_map_range (char *name, DCELL dmin, DCELL dmax, CELL cmin, CELL cmax)

Writes the `f_quant` file for the raster map *name* with one rule. The rule is generated using the floating-point range [*dmin,dmax*] and the integer range [*min,max*]. This routine differs from the one above in that the application controls the floating-point range. For example, r.slope.aspect will use this routine to quantize the slope map from [0.0, 90.0] to [0, 90] even if the range of slopes is not 0-90. The aspect map would be quantized from [0.0, 360.0] to [0, 360].

12.11.9 Categories Labeling Functions (new and upgraded)

12.11.9.1 Upgraded Categories structure

All the new programs which are using Categories structure directly have to be modified to use API functions to update and retrieve info from Categories structure. Both new and old API function can be used, since old functions still have exact same functionality (even though internally they are implemented very differently). New function names end with raster_cats(); old function names end with _cats().

We made sure that all old fields in Categories structure are either missing in new Categories structure or have exactly the same meaning. We did it so that the modules using Categories structure directly either do not compile with new gis library or work exactly the same as bnefore. A programmer might want to read the data in a floating point map in a way that each cell value stores index of it's category label and data range. The way to do it is to call G_set_quant_rules(fd, &pcats->q) after openning the map.

This is helpful when trying to collect statistics (how many cells of each category are in the map. (although there is another new mechanism to collect such stats - see G_mark_raster_cats()). Another reason to get a category index instead of fp values is that this index will be the FID into GRASS-DBMS link. Also he can use G_get_ith_raster_cat() to get the category information for each cell using this index.

Here is the new Categories structure defined in `"gis.h"`:

```
struct Categories
CELL ncats ; /* total number of categories */
CELL num ; /* the highest cell values.  Only exists
for backwards compatibility = (CELL)
max_fp_values in quant rules */
char *title ; /* name of data layer */
```

```
char *fmt ; /* printf-like format to generate labels */
float m1 ; /* Multiplication coefficient 1 */
float a1 ; /* Addition coefficient 1 */
float m2 ; /* Multiplication coefficient 2 */
float a2 ; /* Addition coefficient 2 */
struct Quant q ; /* rules mapping cell values to index in
list of labels */
char **labels ; /* array of labels of size num */
int * marks ; /* was the value with this label was used?  */
int nalloc;
int last_marked_rule ;
;
```

12.11.9.2 Changes to the `cats` file

The format of explicit label entries is the same for integer maps.

```
cat:description
```

In addition label entries of new format is supported for floating point maps.

```
val:descr (where val is a floating point number)
```

or

```
val1:val2:descr (where val1, val2 is a floating point range)
```

Internally the labels are stored for fp ranges of data. However when the cats file is written, all the decimal zeros are stripped so that integer values appear as integers in the file. Also if values are the same, only 1 value is written (i.e. first format).

This way even though the old cats files will be processed differently internally, the user or application programmer will not notice this difference as long as the proper api is used and the elements of Categories structure are not accessed directly without API calls.

12.11.10 Range functions (new and upgraded)

12.11.10.1 New Functions to read/write access and modify Categories structure.

int
G_read_raster_cats (char *name, *mapset, struct Categories *pcats)

Is the same as existing G_read_cats()

int
G_copy_raster_cats (struct Categories *pcats_to, struct Categories*pcats_from)

Allocates NEW space for quant rules and labels n *pcats_to* and copies all info from *pcats_from* cats to *pcats_to* cats.
returns:
0 if successful
-1 on fail

char *
G_get_raster_cat (void *val, struct Categories *pcats, RASTER_MAP_TYPE data_type)

given a raster value *val* of type *data_type* Returns pointer to a string describing category.

char *
G_get_c_raster_cat (CELL *val, struct Categories *pcats)

given a CELL value *val* Returns pointer to a string describing category.

char *
G_get_d_raster_cat (DCELL *val, struct Categories *pcats)

given a DCELL value *val* Returns pointer to a string describing category.

char *
G_get_f_raster_cat (FCELL *val, struct Categories *pcats)

given a FCELL value *val* Returns pointer to a string describing category.

int
G_set_raster_cat (void *rast1, void *rast2, struct Categories *pcats, RASTER_MAP_TYPE data_type)

Adds the label for range *rast1* through *rast2* in category structure *pcats*.

int
G_set_c_raster_cat (CELL *rast1, CELL *rast2, struct Categories *pcats)

Adds the label for range *rast1* through *rast2* in category structure *pcats*.

int
G_set_f_raster_cat (FCELL *rast1, FCELL *rast2, struct Categories *pcats)

Adds the label for range *rast1* through *rast2* in category structure *pcats*.

int
G_set_d_raster_cat (DCELL *rast1, DCELL *rast2, struct Categories *pcats)

Adds the label for range *rast1* through *rast2* in category structure *pcats*.

int *
G_number_of_raster_cats (pcats)

Returns the number of labels. DO NOT use G_number_of_cats() (it returns max cat number)

char *
G_get_ith_raster_cat (struct Categories *pcats, int i, void *rast1, void *rast2, RASTER_MAP_TYPE data_type)

Returns i-th description and i-th data range from the list of category descriptions with corresponding data ranges. Stores end points of data interval in *rast1* and *rast2* (after converting them to *data_type*.

char *
G_get_ith_c_raster_cat (struct Categories *pcats, int i, CELL *rast1, CELL *rast2)

Returns i-th description and i-th data range from the list of category descriptions with corresponding data ranges. end points of data interval in *rast1* and *rast2*.

char *
G_get_ith_f_raster_cat (struct Categories *pcats, int i, FCELL *rast1, FCELL *rast2)

Returns i-th description and i-th data range from the list of category descriptions with corresponding data ranges. end points of data interval in *rast1* and *rast2*.

char *
G_get_ith_d_raster_cat (struct Categories *pcats, int i, DCELL *rast1, DCELL *rast2)

Returns i-th description and i-th data range from the list of category descriptions with corresponding data ranges. end points of data interval in *rast1* and *rast2*.

char *
G_get_raster_cats_title (struct Categories *pcats)

Returns pointer to a string with title.

int
G_unmark_raster_cats (struct Categories *pcats)

Sets marks for all categories to 0. This initializes Categories structure for subsequest calls to G_mark_raster_cats (rast_row,...) for each row of data, where non-zero mark for i-th label means that some of the cells in rast_row are labeled with i-th label and fall into i-th data range.
These marks help determine from the Categories structure which labels were used and which weren't.

int
G_get_next_marked_raster_cat(struct Categories *pcats, void *rast1, void *rast2, long *stats, RASTER_MAP_TYPE data_type)

Finds the next label and corresponding data range in the list of marked categories. The category (label + data range) is marked by G_mark_raster_cats (). End points of the data range are converted to *data_type* and returned in rast1, rast2. the number of times value from i-th cat. data range appeared so far is returned in stats. See G_unmark_raster_cats(), G_rewind_raster_cats() and G_mark_raster_cats ().

int
G_get_next_marked_c_raster_cat(struct Categories *pcats, CELL *rast1, CELL *rast2, long *stats)

Finds the next label and corresponding data range in the list of marked categories. The category (label + data range) is marked by G_mark_raster_cats (). End points of the data range are converted to *data_type* and returned in rast1, rast2. the number of times value from i-th cat. data range appeared so far is returned in stats. See G_unmark_raster_cats(), G_rewind_raster_cats() and G_mark_raster_cats ().

int
G_get_next_marked_f_raster_cat(struct Categories *pcats, FCELL *rast1, FCELL *rast2, long *stats)

Finds the next label and corresponding data range in the list of marked categories. The category (label + data range) is marked by G_mark_raster_cats (). End points of the data range are converted to *data_type* and returned in rast1, rast2. the number of times value from i-th cat. data range appeared so far is returned in stats. See G_unmark_raster_cats(), G_rewind_raster_cats() and G_mark_raster_cats ().

int
G_get_next_marked_d_raster_cat(struct Categories *pcats, DCELL *rast1, DCELL *rast2, long *stats)

Finds the next label and corresponding data range in the list of marked categories. The category (label + data range) is marked by G_mark_raster_cats (). End points of the data range are converted to *data_type* and returned in rast1, rast2. the number of times value from i-th cat. data range appeared so far is returned in stats. See G_unmark_raster_cats(), G_rewind_raster_cats() and G_mark_raster_cats ().

int
G_mark_raster_cats (void *rast_row, int ncols, struct Categories *pcats, RASTER_MAP_TYPE data_type)

Looks up the category label for each raster value in the *rast_row* (row of raster cell value) and updates the marks for labels found.
NOTE: non-zero mark for i-th label stores the number of of raster cells read so far which are labeled with i-th label and fall into i-th data range.

int
G_mark_c_raster_cats (CELL *rast_row, int ncols, struct Categories *pcats)

Looks up the category label for each raster value in the *rast_row* and updates the marks for labels found.
NOTE: non-zero mark for i-th label stores the number of of raster cells read so far which are labeled with i-th label and fall into i-th data range.

int
G_mark_f_raster_cats (FCELL *rast_row, int ncols, struct Categories *pcats)

Looks up the category label for each raster value in the *rast_row* and updates the marks for labels found.
NOTE: non-zero mark for i-th label stores the number of of raster cells read so far which are labeled with i-th label and fall into i-th data range.

int
G_mark_d_raster_cats (DCELL *rast_row, int ncols, struct Categories *pcats)

Looks up the category label for each raster value in the *rast_row* and updates the marks for labels found.
NOTE: non-zero mark for i-th label stores the number of of raster cells read so far which are labeled with i-th label and fall into i-th data range.

int
G_rewind_raster_cats (struct Categories *pcats)

after call to this function G_get_next_marked_raster_cat() returns the first marked cat label.

int
G_init_raster_cats (char *title, struct Categories *pcats)

Same as existing G_init_raster_cats() only ncats argument is missign. ncats has no meaning in new Categories structure and only stores (int) largets data value for backwards compatibility.

int
G_set_raster_cats_fmt (char *fmt, float m1, a1, m2, a2, struct Categories*pcats)

Same as existing G_set_cats_fmt()

int
G_set_raster_cats_title (char *title, struct Categories *pcats)

Same as existing G_set_cats_title()

int
G_write_raster_cats (char *name, struct Categories *pcats)

Same as existing G_write_cats()

int
G_free_raster_cats (struct Categories *pcats)

Same as existing G_free_cats()

12.11.11 Library Functions that are Deprecated

These functions are deprecated, since they imply that the application that uses them has not been upgraded to handle NULL-values and should be eliminated from GRASS code.

- `G_get_map_row()`:
  To be replaced by `G_get_c_raster_row()`.
- `G_get_map_row_nomask()`:
  To be replaced by `G_get_c_raster_row_nomask()`.
- `G_put_map_row()`:
  To be replaced by `G_put_c_raster_row()`.

These functions are deprecated, since they can not be upgraded to support NULL-values, and should be eliminated from GRASS code.

- `G_open_map_new_random()`
- `G_put_map_row_random()`

Also, no support for random writing of floating-point rasters will be provided.

12.11.12 Guidelines for upgrading GRASS 4.x Modules

- Modules that process raster maps as *continuous* data should read raster maps as floating-point. Modules that process raster maps as *nominal* data should read raster maps as integer.

  Exception: Modules that process raster colors or the modules which report on raster categories labels should either always read the maps as floating-point, or read the maps as integer if the map is integer and floating-point if the map is floating-point.

- The quantization of floating-point to integer should NOT change the color table. The color lookup should have its own separate quantization.
- The quantization of floating-point to integer should NOT change the Categories table. The Categories structure should have its own separate quantization.
- Modules that read or write floating-point raster maps should use `double` (`DCELL`) arrays instead of `float` (`FCELL`) arrays.
- Modues should process NULL values in a well defined (consistent) manner. Modules that processed zero as the pseudo NULL-value should be changed to use the true NULL-value for this and process zero as normal value.
- Modules should process non-NULL values as normal numbers and not treat any particular numbers (e.g. zero) as special.

12.11.13 Important hints for upgrades to raster modules

In general modules that use `G_get_map_row()`. should use `G_get_c_raster_row()` instead.

Modules that use `G_put_map_row()`. should use `G_put_c_raster_row()` instead.

12.12 Vector File Processing

Authors:

Written by CERL, with contributions from David D. Gray.

The *GIS Library* contains some functions related to vector file processing. These include prompting the user for vector files, locating vector files in the database, opening vector files, and a few others.

Note. Most vector file processing, however, is handled by routines in the *Vector Library*, which is described in *13 Vector Library (p. 219)*.

12.12.1 Prompting for Vector Files

The following routines interactively prompt the user for a vector file name. In each, the **prompt** string will be printed as the first line of the full prompt which asks the user to enter a vector file name. If **prompt** is the empty string "" then an appropriate prompt will be substituted. The

name that the user enters is copied into the **name** buffer.[28] These routines have a built-in 'list' capability which allows the user to get a list of existing vector files.

The user is required to enter a valid vector file name, or else hit the RETURN key to cancel the request. If the user enters an invalid response, a message is printed, and the user is prompted again. If the user cancels the request, the NULL pointer is returned. Otherwise the mapset where the vector file lives or is to be created is returned. Both the name and the mapset are used in other routines to refer to the vector file.

prompt for an existing vector file

char *
G_ask_vector_old (char *prompt, char *name)

Asks the user to enter the name of an existing vector file in any mapset in the database.

prompt for an existing vector file

char *
G_ask_vector_in_mapset (char *prompt, char *name)

Asks the user to enter the name of an existing vector file in the current mapset.

prompt for a new vector file

char *
G_ask_vector_new (char *prompt, char *name)

Asks the user to enter a name for a vector file which does not exist in the current mapset.

Here is an example of how to use these routines. Note that the programmer must handle the NULL return properly:

```
  char *mapset;
  char name[50];
  mapset = G_ask_vector_old("Enter vector file to be processed",
name);
  if (mapset = = NULL)
  exit(0);
```

[28] The size of **name** should be large enough to hold any GRASS file name. Most systems allow file names to be quite long. It is recommended that name be declared *char name.*

12.12.2 Finding Vector Files in the Database

Noninteractive modules cannot make use of the interactive prompting routines described above. For example, a command line driven module may require a vector file name as one of the command arguments. GRASS allows the user to specify vector file names (or any other database file) either as a simple unqualified name, such as ”roads”, or as a fully qualified name, such as ”roads in *mapset*”, where *mapset* is the mapset where the vector file is to be found. Often only the unqualified vector file name is provided on the command line.

The following routines search the database for vector files:

int *find a vector file*
G_find_vector (char *name, char *mapset)

int *find a vector file*
G_find_vector2 (char *name, char *mapset)

> Look for the vector file **name** in the database. The **mapset** parameter can either be the empty string ””, which means search all the mapsets in the user's current mapset search path,[29] or it can be a specific mapset name, which means look for the vector file only in this one mapset (for example, in the current mapset). If found, the mapset where the vector file lives is returned. If not found, the NULL pointer is returned.

The difference between these two routines is that if the user specifies a fully qualified vector file which exists, then *G_find_vector2( )* modifies **name** by removing the ”in *mapset*” while *G_find_vector( )* does not.[30] Normally, the GRASS programmer need not worry about qualified vs. unqualified names since all library routines handle both forms. However, if the programmer wants the name to be returned unqualified (for displaying the name to the user, or storing it in a data file, etc.), then *G_find_vector2( )* should be used.

For example, to find a vector file anywhere in the database:

```
char name[50];
char *mapset;
if ((mapset = G_find_vector(name,” ”)) = = NULL)
/* not found */
```

[29] See *4.7.1 Mapset Search Path (p. 22)* for more details about the search path.

[30] Be warned that G_find_vector2( ) should not be used directly on a command line argument, since modifying argv[ ] may not be valid. The argument should be copied to another character buffer which is then passed to G_find_vector2( ).

To check that the vector file exists in the current mapset:

```
char name[50];
if (G_find_vector(name,G_mapset( )) = = NULL)
/* not found */
```

12.12.3 Opening an Existing Vector File

The following routine opens the vector file **name** in **mapset** for reading.

The vector file **name** and **mapset** can be obtained interactively using *G_ask_vector_old* or *G_ask_vector_in_mapset*, and noninteractively using *G_find_vector( )* or *G_find_vector2( )*.

open an existing vector file

FILE *
G_fopen_vector_old (char *name, char *mapset)

This routine opens the vector file **name** in **mapset** for reading. A file descriptor is returned if the open is successful. Otherwise the NULL pointer is returned (no diagnostic message is printed).
The file descriptor can then be used with routines in the *Dig Library* to read the vector file. (See *13 Vector Library (p. 219)*).
Note. This routine does not call any routines in the *Dig Library* ; No initialization of the vector file is done by this routine, directly or indirectly.

12.12.4 Creating and Opening New Vector Files

The following routine creates the new vector file **name** in the current mapset[31] and opens it for writing. The vector file **name** should be obtained interactively using *G_ask_vector_new*. If obtained noninteractively (e.g., from the command line), *G_legal_filename* should be called first to make sure that **name** is a valid GRASS file name.

Warning. If **name** already exists, it will be erased and re-created empty. The interactive routine *G_ask_vector_new* guarantees that **name** will not exist, but if **name** is obtained from the command line, **name** may exist. In this case *G_find_vector* could be used to see if **name** exists.

open a new vector file

FILE *
G_fopen_vector_new (char *name)

Creates and opens the vector file **name** for writing.
A file descriptor is returned if the open is successful. Otherwise the NULL pointer is returned (no diagnostic message is printed).

[31] GRASS does not allow files to be created outside the current mapset. See *4.7 Database Access Rules (p. 22)*.

The file descriptor can then be used with routines in the *Dig Library* to write the *vector file*. (See *13 Vector Library (p. 219)*.)
Note. This routine does not call any routines in the *Dig Library*; No initialization of the vector file is done by this routine, directly or indirectly. Also, only the vector file itself (i.e., the *dig* file), is created. None of the other vector support files are created, removed, or modified in any way.

12.12.5 Reading and Writing Vector Files

Reading and writing vector files is handled by routines in the *Dig Library.* See *13 Vector Library (p. 219)* for details.

12.12.6 Vector Category File

GRASS vector files have category labels associated with them. The category file is structured so that each category in the vector file can have a one-line description.

The routines described below read and write the vector category file. They use the *Categories structure* which is described in *12.21 GIS Library Data Structures (p. 212)*.

Note. The vector category file has exactly the same structure as the raster category file. In fact, it exists so that the module *v.to.rast* can convert a vector file to a raster file that has an up-to-date category file.

The routines described in *12.10.2.2 Querying and Changing the Categories Structure (p. 115)* which modify the *Categories* structure can therefore be used to set and change vector categories as well.

int
G_read_vector_cats (char *name, name *mapset, struct Categories *cats) *read vector category file*

The category file for vector file **name** in **mapset** is read into the **cats** structure. If there is an error reading the category file, a diagnostic message is printed and -1 is returned. Otherwise, 0 is returned.

int
G_write_vector_cats (char *name, struct Categories *cats) *write vector category file*

Writes the category file for the vector file **name** in the current mapset from the **cats** structure.
Returns 1 if successful. Otherwise, -1 is returned (no diagnostic is printed).

12.13 Site List Processing (GRASS 5 Sites API)

Authors:
Darrell McCauley and Bill Brown (brown@gis.uiuc.edu)

Site files contain records describing punctual information. Records are limited to files containing only characters from the US-ASCII character set. Records are separated by a newline character (ASCII 0x0a). There are three types of records: comment records, header records, and data records. The formats of each these types of records are described in the following sections.

A site record in the GRASS Sites Format is divided into two parts, each with a different field separator. Part 1 contains location in 2 or more dimensions and part 2 optionally contains attribute information for this location. Both types of fields (and thus site records) are variable length.

12.13.0.1 Part 1 of a Site Record: Location

Part 1 of a site record gives information about location. The field separator in part 1 of the site record is a "pipe" (ASCII 0x7c) character. The last (non-escaped) pipe signifies the end of part 1 (an escaped character is defined as one prefixed by a "backslash" (ASCII 0x5c)). Any additional fields are considered attribute information.

Each field in part 1 indicates a coordinate in some space. There must be at least two fields in part 1: the first describing a geographic easting and the second describing a geographic northing. These may be in either decimal or degrees-minutes-second format.

Additional fields in part 1 are optional but must be stored in decimal format. They should only be used to represent coordinate information about some space (e.g., elevation, time; depending upon how a space is defined).

12.13.1 Part 2 of a Site Record: Attributes

Part 2 contains attribute information for the location given in part 1. The field separator in part 2 of the site record is a "space" character (ASCII 0x20), except when the space character is contained in double quotes (ASCII 0x22). The three types of attributes are: category, decimal, and string. These attributes may be in any order. Each of these attributes have an associated identifier tag defining the type of attribute in a field: # (ASCII 0x23), % (ASCII 0x25), and @ (ASCII 0x40), for category, decimal, and string, respectively. No space character may immediately follow an identifier tag.

12.13.1.1 Category Attributes

Categories are a special kind of attribute. They are used to represent vector or raster categories when sites are transformed into these different data formats. There may be only one category field per record and it must be prefixed with a "pound" or "number" symbol (#). Categories must be integers.

12.13.1.2 Decimal Attributes

Decimal attributes include both integers and floating-point numbers. They are prefixed with a "percent" symbol (%). There may be be zero, one, or more decimal attributes in a site record.

12.13.1.3 String Attributes

String attributes are fields that contain possibly non-numeric information and are prefixed with the "at" or "each" symbol (@). There may be be zero, one, or more string attributes in a site record. String attributes may contain space (ASCII 0x20) characters if the entire attribute, not including the attribute tag (@), is contained within pairs of "double quotes" ("). String attributes may also contain double quotes if they are escaped by prefixing a "backslash" (\).

12.13.1.4 Default

If no identifier tag is prefixed (i.e., none of #, %, or @), the type of attribute defaults to string.

12.13.2 Header and Comment Record Format

In addition to the data record format, the site file may contain comment lines (records containing a pound symbol, 0x23, in the first column) and header lines, both of which are optional. Header records must precede all data records while comment records may occur anywhere within a sites data file.

There are five types of header records: (1) name, (2) description, (3) timestamp, (4) label, and (5) format.

name A name record contains the string "name|" beginning in column 1 and optionally specifies the name of the database file.

description A description record contains the string "desc|" beginning in column 1 and optionally describes the database file (metadata).

timestamp A timestamp record is special type of metadata that contains the string "time|" beginning in column 1 and optionally gives a time and date associated with the entire sites file. GRASS timestamps may be a single date/time or a range (begin/end).

Valid timestamp strings should be formatted using the routine G_format_timestamp, after creating a valid TimeStamp structure using G_set_timestamp or G_set_timestamp_range. Similar routines exist for reading (see: *23 DateTime Library (p. 349)*)

The GRASS DateTime utility library (see *23 DateTime Library (p. 349)*) may be used to easily and accurately perform DateTime arithmetic. *A possible future upgrade would be to specify a particular format identifier tag to indicate a DateTime. Currently, to store a DateTime for each site record, you must specify it as a string and your application must know to expect a DateTime.*

label A label record describes what each dimension and attribute field in site data records represent. It contains the string "labels|" beginning in column 1 and optionally contains field descriptions. No special formatting is required since this record is for user convenience only.

format A format record describes the format of site data records. It contains the string "form|" beginning in column 1 and a special sample data record beginning in column 6. The special sample data record is a site data record (as describe above) containing only field separators and identifier tags (i.e., all data removed).

All header records are optional. If present in a sites data file, header records must occur in the before any data records in a site file.

12.13.3 TimeStamp GISlib functions for sites

```
#include "gis.h"
#include "site.h"
```

This structure is defined in gis.h, but there should be no reason to access its elements directly:

```
struct TimeStamp {
   DateTime dt[2];   /* two datetimes */
   int count;
};
```

Using the G_*_timestamp routines reads/writes a timestamp file in the cell_misc/rastername or dig_misc/vectorname mapset element.

A TimeStamp can be one DateTime, or two DateTimes representing a range. When preparing to write a TimeStamp, the programmer should use one of:

int
G_set_timestamp

to set a single DateTime

int
G_set_timestamp_range

to set two DateTimes.

int
G_read_raster_timestamp (char *name, char *mapset, struct TimeStamp *ts)

Returns 1 on success. 0 or negative on error.

int
G_read_vector_timestamp (char *name, char *mapset, struct TimeStamp *ts)

Returns 1 on success. 0 or negative on error.

int
G_get_timestamps (struct TimeStamp *ts, DateTime *dt1, DateTime *dt2, int *count)

Use to copy the TimeStamp information into Datetimes, so the members of struct TimeStamp shouldn't be accessed directly.
count=0 means no datetimes were copied
count=1 means 1 datetime was copied into dt1
count=2 means 2 datetimes were copied

int
G_init_timestamp (struct TimeStamp *ts)

Sets ts->count = 0, to indicate no valid DateTimes are in TimeStamp.

int
G_set_timestamp (struct TimeStamp *ts, DateTime *dt)

Copies a single DateTime to a TimeStamp in preparation for writing. (overwrites any existing information in TimeStamp)

int
G_set_timestamp_range (struct TimeStamp *ts, DateTime *dt1, DateTime *dt2)

Copies two DateTimes (a range) to a TimeStamp in preparation for writing. (overwrites any existing information in TimeStamp)

int
G_write_raster_timestamp (char *name, struct TimeStamp *ts)

Returns: 1 on success
-1 error - can't create timestamp file
-2 error - invalid datetime in ts

int
G_write_vector_timestamp (char *name, struct TimeStamp *ts)

Returns: 1 on success
-1 error - can't create timestamp file
-2 error - invalid datetime in ts

int
G_format_timestamp (struct TimeStamp *ts, char *buf)

Returns: 1 on success
-1 error

int
G_scan_timestamp (struct TimeStamp *ts, char *buf)

Returns: 1 on success
-1 error

int
G_remove_raster_timestamp (char *name)

Only files in current mapset can be removed Returns: 0 if no file
1 if successful
-1 on fail

int
G_remove_vector_timestamp (char *name)

Only files in current mapset can be removed Returns: 0 if no file
1 if successful
-1 on fail

12.13.4 Record Structure and Definitions

```
typedef struct
 {
  double east, north;
  double *dim;
  int dim_alloc;
  RASTER_MAP_TYPE cattype;
  CELL ccat;
  FCELL fcat;
  DCELL dcat;
  int str_alloc;
  char **str_att;
  int dbl_alloc;
  double *dbl_att;
} Site;
```

`#define MAX_SITE_STRING 1024` The maximum length of a string attribute.

`#define MAX_SITE_LEN 4096` The maximum length of a site record (*i.e.*, the maximum number of characters per line). This is the same value used in GRASS 4.x.

```
typedef struct
{
  char *name, *desc, *form, *labels, *stime;
  struct TimeStamp *time;
} Site_head;
```

12.13.5 Function Prototypes

12.13.5.1 Prompting for Site List Files

The following routines interactively prompt the user for a site list file name. In each, the `prompt` string will be printed as the first line of the full prompt which asks the user to enter a site list file name. If `prompt` is the empty string `""` then an appropriate prompt will be substituted. The name that the user enters is copied into the `name` buffer. (The size of name should be large enough to hold any GRASS file name. Most systems allow file names to be quite long. It is recommended that name be declared `char name[50]`.) These routines have a built-in `"list"` capability which allows the user to get a list of existing site list files.

The user is required to enter a valid site list file name, or else hit the RETURN key to cancel the request. If the user enters an invalid response, a message is printed, and the user is prompted again. If the user cancels the request, the NULL pointer is returned. Otherwise the mapset

where the site list file lives or is to be created is returned. Both the name and the mapset are used in other routines to refer to the site list file.

char *
G_ask_sites_old (char *prompt, char *name)

Asks user to input name of an existing site list file in any mapset in the database.

char *
G_ask_sites_in_mapset (char *prompt, char *name)

Asks user to input name of an existing site list file in the current mapset.

char *
G_ask_sites_new (char *prompt, char *name)

Asks user to input name for a site list file which does not exist in the current mapset.

Here is an example of how to use these routines. Note that the programmer must handle the NULL return properly.

```
  char *mapset;
  char name[50];
  mapset = G_ask_sites_old("Enter site list file to be processed",
name);
  if (mapset == NULL)
  exit(0);
```

12.13.5.2 Opening Site List Files

The following routines open site list files:

FILE *
G_sites_open_new (char *name)

Creates an empty site list file `name` in the current mapset and opens it for writing. Returns an open file descriptor is successful. Otherwise, returns NULL.

FILE *
G_sites_open_old (char *name, char *mapset)

Opens the site list file `name` in `mapset` for reading.
Returns an open file descriptor is successful. Otherwise, returns NULL.

12.13.5.3 Site Memory Management

Sites routines require the use of a `Site` structure. Routines to allocate and deallocate memory are provided, as well as a routine which describes the format of a site list, helpful in determining the amount of memory to be allocated.

Site *
G_site_new_struct (RASTER_MAP_TYPE c, int n, int s, int d)

Allocates and returns pointer to memory for a `Site` structure for storing `n` dimensions (*including* easting and northing; must be > 1), an optional category `c`, `s` string attributes, and `d` decimal attributes. The category `c` can be `CELL_TYPE`, `FCELL_TYPE`, `DCELL_TYPE` (as defined in `gis.h`), or -1 (indicating no category attribute). Returns a pointer to a `Site` structure or `NULL` on error.

int
G_site_describe (FILE *fd, RASTER_MAP_TYPE n, int *c, int *s, int *d)

Guesses the format of a sites list (the dimensionality, the presence and type of a category, and the number of string and decimal attributes) by reading the first record in the file. The type of category will be `CELL_TYPE`, `FCELL_TYPE` (as defined in `gis.h`), or -1 (indicating no category attribute). Reads `fd`, rewinds it, and returns:
0 on success,
-1 on EOF, and
-2 for any other error.

void
G_site_free_struct (Site *site)

Free memory for a `site` struct previously allocated using G_site_new_struct.

Here is an example of how to use these routines.

```
int dims,cat,strs,dbls;
FILE *fp;
Site *mysite;

/* G_site_describe should be called immediately after the
 * file is opened or at least before any seeks are done
 * on the file.
 */

if (G_site_describe (fp, &dims, &cat, &strs, &dbls)!=0)
  G_fatal_error("failed to guess format");

/*
 * Allocate enough memory, according to the output
 * of G_site_describe(~)
 */

mysite = G_site_new_struct (cat, dims, strs, dbls);

G_site_free_struct (mysite);
```

12.13.5.4 Reading and Writing Site List Files

int
G_site_get (FILE *fd, Site *s)

Reads one site record from `fd` and returns:
0 on success
-1 on EOF
-2 on fatal error or insufficient data
1 on format mismatch (extra data)

int
G_site_put (FILE *fd, Site *s)

Writes a site to file pointed to by `fd`.

char *
G_site_format (Site *s, char *fs, int id)

Returns a string containing a formatted site record, with all fields separated by `fs`. If `fs` is NULL, a space character is used. If `id` is non-zero, attribute identifiers (#, %, and @) are included.

int
G_site_get_head (FILE *fd, Site_head *head)

Reads the header from `fd` and stores it in `head`. If a type of header record is not present in `fd`, the corresponding element of `head` is returned as NULL.

int
G_site_put_head (FILE *fd, Site_head *head)

Writes header information stored in `head` to `fd`. Only non-NULL fields of `head` struct are written.

int
G_site_in_region (Site *site, struct Cell_head *region)

Returns 1 if `site` is contained within `region`, 0 otherwise.

int
G_site_c_cmp (void *a, void *b)

compare category attributes

int
G_site_d_cmp (void *a, void *b)

compare first decimal attributes

int
G_site_s_cmp (void *a, void *b)

compare first string attributes

Comparison functions for sorting an array of `Site` records using `qsort`. See examples.

12.13.5.5 GRASS 5: Reading sites with G_readsites_xyz( )

[Written by Eric G . Miller <egm2@jps.net>]

int
G_readsites_xyz (FILE *fdsite, int type, int index, int size, struct Cell_head *region, SITE_XYZ *xyz)

Read a chunk of a site file into a `SITE_XYZ` array setting the **Z** dimension from the specified attribute. The `fdsite` parameter is the `FILE *` for the sites file; `type` is the attribute type to use for the `z` variable value; the index is the 1–based index value for the attribute; the `size` is the size of the `SITE_XYZ` array passed to the function; the `region` is a pointer to a `struct Cell_head` for the current region or `NULL`; and, finally, `xyz` is a pointer to an array of `SITE_XYZ` which will be populated. The return value is the number of records read or `EOF`.

SITE_XYZ *
G_alloc_site_xyz (size_t num)

Allocate an array of `SITE_XYZ` with size `num`.

void
G_free_site_xyz (SITE_XYZ *xyz)

Free a previously allocated array of `SITE_XYZ`.

Constants and the structure used by **G_readsites_xyz( )**.

```
#define SITE_COL_NUL 0
#define SITE_COL_DIM 1
#define SITE_COL_DBL 2
#define SITE_COL_STR 3

typedef struct {
    double x, y, z;
    RASTER_MAP_TYPE cattype;
    union {
            double d;
            float  f;
            int    c;
    } cat ;
} SITE_XYZ;
```

The **G_readsites_xyz( )** function, and its related memory management functions **G_alloc_site_xyz( )** and **G_free_site_xyz( )**, allows the user to process a *site_list* when a third dimension is wanted, but the other attributes aren't needed. The third dimension can come from one of the *n-dims*, a numeric attribute, or a string attribute (provided it can be converted to a double). The category value is also read into the **SITE_XYZ** struct array when it is available. If the region [window] parameter is not `NULL`, then the *site_list* will be filtered based on the region. **G_readsites_xyz( )** can be used to get just the easting, northing and category value (if available) by passing `SITE_COL_NUL` for the field parameter.

A different idea about the indexing of *n-dims* is used by **G_readsites_( )** as compared to functions operating on a `struct Site`. The easting and northing are not counted, so the index is 2 less. Index values are 1–based; that is, the value passed to **G_readsites_xyz( )** for the **index** should be 1 or greater (it can be anything if `type == SITE_COL_NUL`).

G_readsites_xyz( ) makes it possible to process large *site_lists* in memory as less space is needed for a **SITE_XYZ** struct versus a **Site** struct. Still, the user should choose a reasonably sized array and use a looping call structure to prevent out of memory errors. The function will die with a fatal error under the following conditions:

- Failure to "guess" the *site_list* structure using **G_site_describe( )**.
- Asking for an unknown attribute type (use the #define's above).
- The *site_list* does not have a consistent number and type of attributes for every record.
- The *Z-dimension* is requested from a non-existent attribute type or the index is out of range for the attribute type.
- Failure to convert a string attribute to a double.

The return value of **G_readsites_xyz( )** will either be `EOF` (which is typically -1) or the number of records read. If the number of records returned is less than the size of the **SITE_XYZ** array, then it is safe to assume there are no more records. Subsequent calls will return `EOF`. **WARNING: Never make read calls on the *site_list* file stream in between calls to G_readsites_xyz( ) without first saving the file position and then restoring it. G_readsites_xyz( ) assumes the file stream is at the position left by previous calls.**

The following is a simple program showing the use of the functions. It assumes a *site_list* file with the name "test" in the `PERMANENT` mapset.

```
/* Test the new G_readsites_xyz() interface */

#include <stdio.h>
#include <stdlib.h>
#include "gis.h"
#include "site.h"


int main (void)
{
```

```
    int i, num, ret_code, index, type;
    SITE_XYZ *mysites;
    struct Cell_head *region;
    FILE *site_file;

    G_gisinit("test_readsites");

    site_file = G_sites_open_old("test", "PERMANENT");

    if(!site_file) {
        fprintf(stderr, "Failed to open test file\n");
        exit(EXIT_FAILURE);
    }

    region = (struct Cell_head *) G_malloc(sizeof(struct Cell_head));
    G_get_set_window(region);
    num = 100;
    if(NULL == (mysites = G_alloc_site_xyz(num))) {
        fprintf(stderr, "Failed to allocate site array!\n");
        exit(EXIT_FAILURE);
    }

    type = SITE_COL_DIM;
    index = 1;
    ret_code = G_readsites_xyz(site_file, type, index, num,
            region, mysites);
    printf("First Run: num = %d, type = %d, index = %d\n",
            num, type, index);
    printf("Returned: ret_code = %d\n", ret_code);
    printf("Values --->\n");
    for (i = 0; i < ret_code; i++) {
        printf ("X: %f, Y: %f, Z: %f ",
                mysites[i].x, mysites[i].y, mysites[i].z);
        switch (mysites[i].cattype) {
            case CELL_TYPE:
                printf("Cat: %d\n", mysites[i].cat.c); break;
            case FCELL_TYPE:
                printf("Cat: %f\n", mysites[i].cat.f); break;
            case DCELL_TYPE:
                printf("Cat: %f\n", mysites[i].cat.d); break;
            default:
                printf("Cat: (nil)\n");
        }
    }
    printf("\n");

    G_free(region);
    G_free_site_xyz(mysites);

    return 0;
}
```

12.13.6 Sites Programming Examples

12.13.6.1 Time as String Attributes

(TODO: change to use TimeStamps or DateTime library as a single string)

In this example, we will work with the following site list:

```
name|time
desc|Example of using time as an attribute
time|Mon Apr 17 14:24:06 EST 1995
10.8|0|9.8|Fri Sep 13 10:00:00 1986 %31.4
11|5.5|9.9|Fri Sep 14 00:20:00 1985 %36.4
5.1|3.9|10|Fri Sep 15 00:00:30 1984 %28.4
```

This data has three dimensions (assume easting, northing, and elevation), five string attributes, and one decimal attributes.

Now follow along in this skeleton C program. Remember that in real code, you should always check return values.

```
#include "gis.h"            /* includes stdio.h for file I/O */
#include "site.h"           /* include definitions and prototypes */

int main (int argc, char **argv)
\{
  int dims=0,strs=0,dbls=0;
  RASTER_MAP_TYPE map_type;
  Site *mysite;               /* pointer to Site */
  Site_head info;
  FILE *fp;
  char *mapset;

  /* find mapset that site list is in and open the site list */
  mapset = G_find_file ("site_lists", parm.input->answer, "");

  fp = G_fopen_sites_old (parm.input->answer, mapset);

  /* G_site_describe should be called immediately after the
   * file is opened or at least before any seeks are done
   * on the file.
   */
  if (G_site_describe (fp, &dims, &map_type, &strs, &dbls)!=0)
    G_fatal_error("failed to guess format");

  fprintf(stdout,"Guessed %d %d %d %d (should be 3 5 1 0)\n",
          dims, strs, dbls, cat);

  /*
  * Read header fields first, then write to stderr.
  * This step is optional since the first call to G_site_get(~)
  * would skip over any comment or header records.
  */
  G_site_get_head(fp,&info);
```

```
  G_site_put_head(stderr,&info);

  /*
  * Allocate enough memory, according to the output
  * of G_site_describe(~)
  */
  mysite = G_site_new_struct (fp, &dims, &map_type, &strs, &dbls);

  /*
   * G_site_get(~) returns -1 on EOF, -2 on error. This code ignores
   * all records following the first invalid one.
   */
  while ((err=G_site_get (fp, mysite)) == 0)
  \{
    /* do something useful with time information */

    /* write the site to stderr instead of output file */
    G_site_put(stderr,mysite);
  \}
\}
```

Running our sample program, we get:

```
Mapset <PERMANENT> in Location <temporal>
GRASS 5.0 > s.egtime time-h
name|time
desc|Example of using time as an attribute
time|Mon Apr 17 14:24:06 EST 1995
10.8|0|9.8|%31.4 @Fri @Sep @13 @10:00:00 @1986
11|5.5|9.9|%36.4 @Fri @Sep @14 @00:20:00 @1985
5.1|3.9|10|%28.4 @Fri @Sep @15 @00:00:30 @1984
```

Compare the above output to the input site list given earlier.

In this example, we read "time" as five string attributes. Using the GRASS DateTime library, we could convert this to GRASS DateTimes and do sometime more useful with this information. We also could have used the TimeStamp GISlib functions to format a single standard GRASS TimeStamp string instead of requiring 5 separate strings.

12.13.6.2 Key Points

After studying the above, you should:

- Understand the difference between dimensions and attributes;
- Understand how untagged attributes are interpreted (as string attributes);
- Know when to call `G_site_get_fmt( )` and how to get *all* fields from a site record; and
- Know how to read and write a site file.

12.13.6.3 Example 2: Sorting Arrays and Selective Reads

In this example, we will work again with the site list from the Time Attribute Example:

```
name|time
desc|Example of using time as an attribute
time|Mon Apr 17 14:24:06 EST 1995
10.8|0|9.8|Fri Sep 13 10:00:00 1986 %31.4
11|5.5|9.9|Fri Sep 14 00:20:00 1985 %36.4
5.1|3.9|10|Fri Sep 15 00:00:30 1984 %28.4
```

Recall that the data has three dimensions, five string attributes, and one decimal attributes. However, in this example we are writing a program which only uses two dimensional attributes and one decimal attribute.

Follow along in this skeleton C program and remember that, in real code, you should always check return values!

```
#include "gis.h"            /* includes stdio.h for file I/O */
#include "site.h"           /* include definitions and prototypes */


int main (argc, argv)
  char **argv;
  int argc;
{
  int sites_alloced=5, n=0;
  Site **mysite;               /* pointer to pointer to Site */

  /*
   * We allocate memory for an array of Site structs.
   */
  mysites=(Site **) G_malloc(sites_alloced*sizeof(Site *));


  /*
   * Here we only allocate space for 2 dimensions and one decimal attribute.
   * Thus any calls to G_site_get(~) will ingore dimensional fields
   * past the first two, any category attribute, and all string attributes
   */
  mysites[n] = G_site_new_struct (2, 0, 1);

  while ((i=G_site_get (fp, mysites[n])) != EOF)
  {
    /*
     * (we should test for i==2 and deal with appropriately)
     */
    G_site_put(stdout,mysites[n++],0);
    /*
     * This snippet could have been left out for compactness since
     * it is not critical to this example. However, this shows how
     * to read an unknown number of sites in a robust fashion.
     */
    if (n==sites_alloced)
```

```
    {
      sites_alloced+=100;
      mysites=(Site **) G_realloc(mysites, sites_alloced*sizeof(Site *));
      if (mysites==NULL)
        G_fatal_error("memory reallocation error");
    }

    /*
     * We must call G_site_new_struct(~) for each element
     * in this array. Doing this inside the while loop instead of
     * before the while loop saves memory (since we are only allocating
     * on an as-needed basis).
     */
    mysites[n] = G_site_new_struct (2, 0, 1);
  }
  G_free(mysites[n]);      /* We did not need the last one */
  fprintf(stdout, "\n");

  /* sort the array of sites into ascending order */
  qsort (mysites, n, sizeof (Site *), G_site_d_cmp);

  /* write the sorted array to standard output */
  for(i=0;i<n;++i)
    G_site_put(stdout,mysites[i],0);

  fprintf(stdout, "\n");

  /* write only dimensional fields and no attributes */
  for(i=0;i<n;++i)
  {
    mysites[i]->dbl_alloc=0;
    G_site_put(stdout,mysites[i],0);
  }

  return 0;
}
```

Running our sample program, we get:

```
Mapset <PERMANENT> in Location <temporal>
GRASS 5.0 > s.egsort time-h
10.8|0|%31.4
11|5.5|%36.4
5.1|3.9|%28.4

5.1|3.9|%28.4
10.8|0|%31.4
11|5.5|%36.4

5.1|3.9|
10.8|0|
11|5.5|
```

Compare the above output to the input site list given earlier. We *read* only the first two dimensional attributes and the first decimal attribute–all others were safely ignored.

The resulting site list is sorted into ascending order according to the first decimal attribute. Similar functions exist for sorting by the first string attribute or by category attribute. For sorting by second or third specific fields, you may write your own `qsort` comparison functions using these examples.

We can selectively *write* some or none of attribute fields by altering the `Site` structure. For situations requiring writing of variable attributes (more complex than this example), pointer manipulation may be necessary.

In this example, we read selectively read dimension and attribute fields,

12.13.6.4 Key Points

After studying the above, you should:

- Understand memory allocation for selective reads;
- Understand how to re-allocate memory for vectors so that static limits are not necessary;
- Know how to sort an array of `Site structs` by certain attributes.
- Know how to selectively write attributes to a site file.

12.14 General Plotting Routines

The following routines form the foundation of a general purpose line and polygon plotting capability.

int
G_bresenham_line (int x1, int y1, int x2, int y2, int (*point)( ))

Bresenham line algorithm

Draws a line from **x1,y1** to **x2,y2** using Bresenham's algorithm. A routine to plot points must be provided, as is defined as:
point(x, y) plot a point at x,y
This routine does not require a previous call to *G_setup_plot* to function correctly, and is independent of all following routines.

int
G_setup_plot (double t, double b, double l, double r, nt (*Move)( ), int (*Cont)( ))

initialize plotting routines

Initializes the plotting capability. This routine must be called once before calling the **G_plot_*( )** routines described below.

The parameters **t, b, l, r** are the top, bottom, left, and right of the output x,y coordinate space. They are not integers, but doubles to allow for subpixel registration of the input and output coordinate spaces. The input coordinate space is assumed to be the current GRASS region, and the routines supports both planimetric and latitude- longitude coordinate systems.
Move and **Cont** are subroutines that will draw lines in x,y space. They will be called as follows:
Move(x, y) move to x,y (no draw)
Cont(x, y) draw from previous position
to x,y. Cont( ) is responsible for clipping

plot line between latlon coordinates

int
G_plot_line (double east1, double north1, double east2, double north2)

A line from **east1,north1** to **east2,north2** is plotted in output x,y coordinates (e.g. pixels for graphics.) This routine handles global wrap-around for latitude-longitude databases.

See **G_setup_plot** (*A.3 Appendix C: Index to GIS Library (p. 445)*) for the required coordinate initialization procedure.

plot filled polygon with n vertices

int
G_plot_polygon (double *east, double *north, int n)

The polygon, described by the **n** vertices **east,north**, is plotted in the output x,y space as a filled polygon.

See **G_setup_plot** (*A.3 Appendix C: Index to GIS Library (p. 445)*) for the required coordinate initialization procedure.

plot multiple polygons

int
G_plot_area (double **xs, double **ys, int *npts, int rings)

Like G_plot_polygon, except it takes a set of polygons, each with **npts[*i*]** vertices, where the number of polygons is specified with the **rings** argument. It is especially useful for plotting vector areas with interior islands.

x,y to east,north

int
G_plot_where_en (int x, int y, double *east, double *north)

The pixel coordinates **x,y** are converted to map coordinates **east,north.**

See **G_setup_plot** (*A.3 Appendix C: Index to GIS Library (p. 445)*) for the required coordinate initialization procedure.

int
G_plot_where_xy (double *east, double *north, int *x, int *y)

east,north to x,y

The map coordinates **east,north** are converted to pixel coordinates **x,y.**

See **G_setup_plot** (*A.3 Appendix C: Index to GIS Library (p. 445)*) for the required coordinate initialization procedure.

int
G_plot_fx (double (*f)( ), double east1, double east2)

plot f(east1) to f(east2)

The function **f(east)** is plotted from **east1** to **east2.** The function **f(east)** must return the map northing coordinate associated with east.

See **G_setup_plot** (*A.3 Appendix C: Index to GIS Library (p. 445)*) for the required coordinate initialization procedure.

12.15 Temporary Files

Often it is necessary for modules to use temporary files to store information that is only useful during the module run. After the module finishes, the information in the temporary file is no longer needed and the file is removed. Commonly it is required that temporary file names be unique from invocation to invocation of the module. It would not be good for a fixed name like "/tmp/mytempfile" to be used. If the module were run by two users at the same time, they would use the same temporary file. The following routine generates temporary file names which are unique within the module and across all GRASS programs.

char *
G_tempfile ( )

returns a temporary file name

This routine returns a pointer to a string containing a unique file name that can be used as a temporary file within the module. Successive calls to G_tempfile( ) will generate new names.
Only the file name is generated. The file itself is not created. To create the file, the module must use standard UNIX functions which create and open files, e.g., creat( ) or fopen( ).
The programmer should take reasonable care to remove (unlink) the file before the module exits. However, GRASS database management will eventually remove all

temporary files created by G_tempfile( ) that have been left behind by the modules which created them.

Note. The temporary files are created in the GRASS database rather than under /tmp. This is done for two reasons. The first is to increase the likelihood that enough disk is available for large temporary files since /tmp may be a very small file system. The second is so that abandoned temporary files can be automatically removed (but see the warning below).

Warning. The temporary files are named, in part, using the process id of the module. GRASS database management will remove these files only if the module which created them is no longer running. However, this feature has a subtle trap. Programs which create child processes (using the UNIX fork( )[32] routine) should let the child call G_tempfile( ). If the parent does it and then exits, the child may find that GRASS has removed the temporary file since the process which created it is no longer running.

12.16 Command Line Parsing

The following routines provide a standard mechanism for command line parsing. Use of the provided set of routines will standardize GRASS commands that expect command line arguments, creating a family of GRASS modules that is easy for users to learn. As soon as a GRASS user familiarizes himself with the general form of command line input as defined by the parser, it will greatly simplify the necessity of remembering or at least guessing the required command line arguments for any GRASS command. It is strongly recommended that GRASS programmers use this set of routines for all command line parsing. With their use, the programmer is freed from the burden of generating user interface code for every command. The parser will limit the programmer to a pre-defined look and feel, but limiting the interface is well worth the shortened user learning curve.

12.16.1 Description

The GRASS parser is a collection of five subroutines which use two structures that are defined in the GRASS ”gis.h” header file. These structures allow the programmer to define the options and flags that make up the valid command line input of a GRASS command.

The parser routines behave in one of three ways:

(1) If no command line arguments are entered by the user, the parser searches for a completely interactive version of the command. If the interactive version is found, control is passed over to this version. If not, the parser will prompt the user for all programmer-defined options and flags. This prompting conforms to the same standard for every GRASS command that uses the parser routines.

(2) If command line arguments are entered but they are a subset of the options and flags that the programmer has defined as required arguments, three things happen. The parser will pass an

[32] See also *G_fork*.

error message to the user indicating which required options and/or flags were missing from the command line, the parser will then display a complete usage message for that command, and finally the parser cancels execution of the command.

(3) If all necessary options and flags are entered on the command line by the user, the parser executes the command with the given options and flags.

12.16.2 Structures

The parser routines described below use two structures as defined in the GRASS "gis.h" header file.

This is a basic list of members of the Option and Flag structures. A comprehensive description of all elements of these two structures and their possible values can be found in *12.16.5 Full Structure Members Description (p. 194)*.

12.16.2.1 Option structure

These are the basic members of the Option structure.

```
struct Option *opt; /* to declare a command line option */
```

Structure Member Description of Member

```
opt->key Option name that user will use
opt->description Option description that is shown to the user
opt->type Variable type of the user's answer to the option
opt->required Is this option required on the command line?  (Boolean)
```

12.16.2.2 Flag structure

These are the basic members of the Flag structure.

```
struct Flag *flag; /* to declare a command line flag */
```

Structure Member Description of Member

```
flag->key Single letter used for flag name
flag->description Flag description that is shown to the user
```

12.16.3 Parser Routines

Associated with the parser are five routines that are automatically included in the GRASS Gmakefile process. The Gmakefile process is documented in *11 Compiling and Installing GRASS Modules (p. 69).*

returns Option structure
struct Option *
G_define_option ( )

Allocates memory for the Option structure and returns a pointer to this memory (of *type struct Option *).*

return Flag structure
struct Flag *
G_define_flag ( )

Allocates memory for the Flag structure and returns a pointer to this memory (of *type struct Flag *).*

parse command line
int
G_parser (int argc, char *argv[ ])

The command line parameters **argv** and the number of parameters **argc** from the main( ) routine are passed directly to *G_parser* ( ). *G_parser* ( ) accepts the command line input entered by the user, and parses this input according to the input options and/or flags that were defined by the programmer.
G_parser ( ) returns 0 if successful. If not successful, a usage statement is displayed that describes the expected and/or required options and flags and a non-zero value is returned.

command line help/usage message
int
G_usage ( )

Calls to *G_usage ( )* allow the programmer to print the usage message at any time. This will explain the allowed and required command line input to the user. This description is given according to the programmer's definitions for options and flags. This function becomes useful when the user enters options and/or flags on the command line that are syntactically valid to the parser, but functionally invalid for the command (e.g. an invalid file name.)
For example, the parser logic doesn't directly support grouping options. If two options be specified together or not at all, the parser must be told that these options

are not required and the programmer must check that if one is specified the other must be as well. If this additional check fails, then *G_parser* will succeed, but the programmer can then call *G_usage* ( ) to print the standard usage message and print additional information about how the two options work together.

int
G_disable_interactive ( )

turns off interactive capability

When a user calls a command with no arguments on the command line, the parser will enter its own standardized interactive session in which all flags and options are presented to the user for input. A call to *G_disable_interactive( )* disables the parser's interactive prompting.

Note: Displaying multiple answers default values (new in GRASS 5, see d.pan for example).

```
char *def[] = "One", "Two", "Last", NULL;
opt->multiple = YES;
opt->answers = def;
G_parser( );
```

The programmer may not forget last NULL value.

12.16.4 Parser Programming Examples

The use of the parser in the programming process is demonstrated here. Both a basic step by step example and full code example are presented.

12.16.4.1 Step by Step Use of the Parser

These are the four basic steps to follow to implement the use of the GRASS parser in a GRASS command:

(1) Allocate memory for Flags and Options:

Flags and Options are pointers to structures allocated through the parser routines *G_define_option and G_define_flag* as defined in *12.16.3 Parser Routines (p. 188)*.

```
  #include "gis.h" ; /* The standard GRASS include file */
  struct Option *opt ; /* Establish an Option pointer for each
option */
  struct Flag *flag ; /* Establish a Flag pointer for each option
*/
```

```
  opt = G_define_option( ) ; /* Request a pointer to memory for
each option */
  flag = G_define_flag( ) ; /* Request a pointer to memory for
each flag */
```

(2) Define members of Flag and Option structures:

The programmer should define the characteristics of each option and flag desired as outlined by the following example:

```
  opt->key = "option"; /* The name of this option is "option".
*/
  opt->description = "Option test"; /* The option description
is "Option test" */
  opt->type = TYPE_STRING; /* The data type of the answer to the
option */
  opt->required = YES; /* This option *is* required from the user
*/
  flag->key = 't'; /* Single letter name for flag */
  flag->description = "Flag test"; /* The flag description is
"Flag test" */
```

Note. There are more options defined later in *12.16.5.1 Complete Structure Members Table (p. 194)*.

(3) Call the parser:

```
  int main(argc,argv) char *argv[ ]; /* command line args passed
into main( ) */
  G_parser(argc,argv); /* Returns 0 if successful, non-zero otherwise
*/
```

(4) Extracting information from the parser structures:

```
  fprintf(stdout, "For the option \"%s\" you chose:  <%s>\n",
opt->description, opt->answer );
  fprintf(stdout, "The flag \"-%s\" is %s set.\n", flag->key,
flag->answer ?  ""  :  "not" );
```

(5) Running the example program

Once such a module has been compiled (for example to the default executable file *a.out* , execution will result in the following user interface scenarios. Lines that begin with # imply user entered commands on the command line.

a.out help

This is a standard user call for basic help information on the module. The command line options (in this case, ”help”) are sent to the parser via *G_parser*. The parser recognizes the ”help” command line option and returns a list of options and/or flags that are applicable for the specific command. Note how the programmer provided option and flag information is captured in the output.

a.out [-t] option=name

Flags:

-tFlag test

Parameters:

option Option test

Now the following command is executed:

a.out -t

This command line does not contain the required option. Note that the output provides this information along with the standard usage message (as already shown above.)

Required parameter <option> not set (Option test).

Usage:

a.out[-t] option=name

Flags:

-t Flag test

Parameters:

option Option test

The following commands are correct and equivalent. The parser provides no error messages and the module executes normally:

a.out option=Hello -t

a.out -t option=Hello

For the option ”Option test” you chose: Hello

The flag ”-t” is set.

If this specific command has no fully interactive version (a user interface that does not use the parser), the parser will prompt for all programmer-defined options and/or flags.

User input is in *italics*, default answers are displayed in square brackets [ ].

a.out

OPTION: Option test

key: option

required: YES

enter option > *Hello*

You have chosen:

option=Hello

Is this correct? (y/n) [y] *y*

FLAG: Set the following flag?

Flag test? (y/n) [n] *n*

You chose: <Hello>

The flag is not set

12.16.4.2 Full Module Example

The following code demonstrates some of the basic capabilities of the parser. To compile this code, create this Gmakefile and run the *gmake* command (see *11 Compiling and Installing GRASS Modules (p. 69)*.

sample: sample.o

$(CC) $(LDFLAGS) -o $@ sample.o $(GISLIB)

The sample.c code follows. You might experiment with this code to familiarize yourself with the parser.

Note. This example includes some of the advanced structure members described in *12.16.5.1 Complete Structure Members Table (p. 194)*.

#include ”gis.h”

main(argc , argv)

int argc ;

```
char *argv ;
{
struct Option *opt ;
struct Option *coor ;
struct Flag *flag ;
double X , Y ;
int n ;
opt = G_define_option( ) ;
opt->key = ”debug” ;
opt->description = ”Debug level” ;
opt->type = TYPE_STRING ;
opt->required = NO ;
opt->answer = ”0” ;
coor = G_define_option( ) ;
coor->key = ”coordinate” ;
coor->key_desc = ”x,y” ;
coor->description = ”One or more coordinates” ;
coor->type = TYPE_STRING ;
coor->required = YES ;
coor->multiple = YES ;

/* Note that coor->answer is not given a default value. */

flag = G_define_flag( ) ;
flag->key = ’v’ ;
flag->description = ”Verbose execution” ;

/* Note that flag->answer is not given a default value. */
```

```
if (G_parser( argc , argv ))

exit( -1 );

fprintf(stdout, "For the option \"%s\" you chose: <%s>\n", opt->description, opt->answer );

fprintf(stdout, "The flag \"-%s\" is: %s set\n", flag->key, flag->answer ? "" : "not");

fprintf(stdout, "You specified the following coordinates:\n");

for ( n=0 ; coor->answers[n] != NULL ; n+=2 )

{

G_scan_easting ( coor->answers[n ] , &X , G_projection( ) );

G_scan_northing ( coor->answers[n+1] , &Y , G_projection( ) );

fprintf(stdout, "%.31f,%.21f\n", X , Y );

}

}
```

12.16.5 Full Structure Members Description

There are many members to the Option and Flag structures. The following tables and descriptions summarize all defined members of both the Option and Flag structures.

An in-depth summary of the more complex structure members is presented in *12.16.5.2 Description of Complex Structure Members (p. 196)*.

12.16.5.1 Complete Structure Members Table

struct Flag

structure member	C type	required	default	description and example
key	char	YES	none	Key char used on command line flag->key = 'f' ;
Description	char *	YES	none	String describing flag meaning flag->description = "run in fast mode" ;
answer	char	NO	NULL	Default and parser-returned flag states.

struct Option

structure member	C type	required	default	description and example
key	char *	YES	none	Key word used on command line. opt->key = ”map” ;
type	int	YES	none	Option type: TYPE_STRING TYPE_INTEGER TYPE_DOUBLE opt->type = TYPE_STRING ;
Description	char *	YES	none	String describing option opt->description = ”Map name” ;
answer	char *	NO	NULL	Default and parser-returned answer to an option. opt->answer = ”defaultmap” ;
key_desc	char *	NO	NULL	Single word describing the key. Commas in this string denote to the parser that several comma-separated arguments are expected from the user as one answer. For example, if a pair of coordinates is desired, this element might be defined as follows. opt->key_desc = ”x,y” ;
structure member	C type	required	default	description and example
multiple	int	NO	NO	Indicates whether the user can provide multiple answers or not. YES and NO are defined in ”gis.h” and should be used (NO is the default.) Multiple is used in conjunction with the answers structure member below. opt->multiple = NO ;
answers		NO	NULL	Multiple parser-returned answers to an option. N/A
required	int	NO	NO	Indicates whether user MUST provide the option on the command line. YES and NO are defined in ”gis.h” and should be used (NO is the default.) opt->required = YES ;
options	char *	NO	NULL	Approved values or range of values. opt->options = ”red,blue,white” ;For integers and doubles, the following format is available: opt->options = ”0-1000” ;

Gisprompt	char *	NO	NULL	Interactive prompt guidance. There are three comma separated parts to this argument which guide the use of the standard GRASS file name prompting routines. opt->gisprompt = ”old,cell,raster” ;
Checker	char *( )	NO	NULL	Routine to check the answer to an option m opt->checker = my_routine( ) ;

12.16.5.2 Description of Complex Structure Members

What follows are explanations of possibly confusing structure members. It is intended to clarify and supplement the structures table above.

12.16.5.2.1 Answer member of the Flag and Option structures. The answer structure member serves two functions for GRASS commands that use the parser.

(1) To set the default answer to an option:

If a default state is desired for a programmer-defined option, the programmer may define the Option structure member ”answer” before calling *G_parser* in his module. After the *G_parser* call, the answer member will hold this preset default value if the user did *not* enter an option that has the default answer member value.

(2) To obtain the command-line answer to an option or flag: After a call to* G_parser*, the answer member will contain one of two values:

(a) If the user provided an option, and answered this option on the command line, the default value of the answer member (as described above) is replaced by the user’s input.

(b) If the user provided an option, but did *not* answer this option on the command line, the default is not used. The user may use the default answer to an option by withholding mention of the option on the command line. But if the user enters an option without an answer, the default answer member value will be replaced and set to a NULL value by *G_parser.*

As an example, please review the use of answer members in the structures implemented in *12.16.4.2 Full Module Example (p. 192).*

12.16.5.2.2 Multiple and Answers Members The functionality of the answers structure member is reliant on the programmer’s definition of the multiple structure member. If the

multiple member is set to NO, the answer member is used to obtain the answer to an option as described above.

If the multiple structure member is set to YES, the programmer has told *G_parser* to capture multiple answers. Multiple answers are separated by commas on the command line after an option.

Note. *G_parser* does not recognize any character other than a comma to delimit multiple answers.

After the programmer has set up an option to receive multiple answers, these the answers are stored in the answers member of the Option structure. The answers member is an array that contains each individual user-entered answer. The elements of this array are the type specified by the programmer using the type member. The answers array contains however many comma-delimited answers the user entered, followed (terminated) by a NULL array element.

For example, here is a sample definition of an Option using multiple and answers structure members:

opt->key =”option” ;

opt->description = ”option example” ;

opt->type = TYPE_INTEGER ;

opt->required = NO ;

opt->multiple = YES ;

The above definition would ask the user for multiple integer answers to the option. If in response to a routine that contained the above code, the user entered ”option=1,3,8,15” on the command line, the answers array would contain the following values:

answers[0] = = 1

answers[1] = = 3

answers[2] = = 8

answers[3] = = 15

answers[4] = = NULL

12.16.5.2.3 key_desc Member The key_desc structure member is used to define the format of a single command line answer to an option. A programmer may wish to ask for one answer to an option, but this answer may not be a single argument of a type set by the type structure

member. If the programmer wants the user to enter a coordinate, for example, the programmer might define an Option as follows:

opt->key =”coordinate” ;

opt->description = ”Specified Coordinate” ;

opt->type = TYPE_INTEGER ;

opt->required = NO ;

opt->key_desc = ”x,y”

opt->multiple = NO ;

The answer to this option would *not* be stored in the answer member, but in the answers member. If the user entered ”coordinate=112,225” on the command line in response to a routine that contains the above option definition, the answers array would have the following values after the call to *G_parser*:

answers[0] = = 112

answers[1] = = 225

answers[2] = = NULL

Note that ”coordinate=112” would not be valid, as it does not contain both components of an answer as defined by the key_desc structure member.

If the multiple structure member were set to YES instead of NO in the example above, the answers are stored sequentially in the answers member. For example, if the user wanted to enter the coordinates (112,225), (142,155), and (43,201), his response on the command line would be ”coordinate=112,225,142,155,43,201”. Note that *G_parser* recognizes only a comma for both the key_desc member, and for multiple answers.

The answers array would have the following values after a call to *G_parser*:

answers[0] = = 112 answers[1] = = 225

answers[2] = = 142 answers[3] = = 155

answers[4] = = 43 answers[5] = = 201

answers[6] = = NULL

Note. In this case as well, neither ”coordinate=112” nor ”coordinate=112,225,142” would be valid command line arguments, as they do not contain even pairs of coordinates. Each answer’s format (as described by the key_desc member) must be fulfilled completely.

The overall function of the key_desc and multiple structure members is very similar. The key_desc member is used to specify the number of *required* components of a single option answer (e.g. a multi-valued coordinate.) The multiple member tells *G_parser* to ask the user for multiple instances of the compound answer as defined by the format in the key_desc structure member.

Another function of the key_desc structure member is to explain to the user the type of information expected as an answer. The coordinate example is explained above.

The usage message that is displayed by *G_parser* in case of an error, or by

G_usage on programmer demand, is shown below. The Option ”option” for the command *a.out* does not have its key_desc structure member defined.

Usage:

a.out option=name

The use of ”name” is a *G_parser* standard. If the programmer defines the key_desc structure member before a call to *G_parser*, the value of the key_desc member replaces ”name”. Thus, if the key_desc member is set to ”x,y” as was used in an example above, the following usage message would be displayed:

Usage:

a.out option=x,y

The key_desc structure member can be used by the programmer to clarify the usage message as well as specify single or multiple required components of a single option answer.

12.16.5.2.4 gisprompt Member The gisprompt Option structure item requires a bit more description. The three comma-separated

(no spaces allowed) sub-arguments are defined as follows:

First argument:

”old” results in a call to the GRASS library subroutine *G_ask_old*, ”new” to *G_ask_new, ”any” to G_ask_any, and ”mapset” to G_ask_in_mapset.*

Second argument:

This is identical to the ”element” argument in the above subroutine calls. It specifies a directory inside the mapset that may contain the user’s response.

Third argument:

Identical to the ”prompt” argument in the above subroutine calls. This is a string presented to the user that describes the type of data element being requested.

Here are two examples:

gisprompt arguments Resulting call

”new,cell,raster” G_ask_new(””, buffer, ”cell”, ”raster”)

”old,dig,vector” G_ask_old(””, buffer, ”dig”, ”vector”)

12.16.6 Common Questions

”How is automatic prompting turned off?”

GRASS 4.0 introduced a new method for driving GRASS interactive and non-interactive modules as described in 11 Compiling and Installing GRASS Programs. Here is a short overview.

For most modules a user runs a front-end module out of the GRASS bin directory which in turn looks for the existence of interactive and non-interactive versions of the module. If an interactive version exists and the user provided no command line arguments, then that version is executed.

In such a situation, the parser’s default interaction will never be seen by the user. A programmer using the parser is able to avoid the front-end’s default search for a fully interactive version of the command by placing a call to *G_disable_interactive* before calling *G_parser* (see *12.16.3 Parser Routines (p. 188)* for details.)

”Can the user mix options and flags?”

Yes. Options and flags can be given in any order.

”In what order does the parser present options and flags?”

Flags and options are presented by the usage message in the order that the programmer defines them using calls to *G_define_option* and *G_define_flag* .

”How does a programmer query for coordinates?”

For any user input that requires a set of arguments (like a pair of map coordinates,) the programmer specifies the number of arguments in the key_desc member of the Option structure. For example, if opt->key_desc was set to ”x,y”, the parser will require that the user enter a pair of arguments separated only by a comma. See the source code for the GRASS commands r.drain or r.cost for examples.

”Is a user required to use full option names?”

No! Users are required to type in only as many characters of an option name as is necessary to make the option choice unambiguous. If, for example, there are two options, ”input=” and ”output=”, the following would be valid command line arguments:

#command i=map1 o=map2

command in=map1 out=map2

"Are options standardized at all?"

Yes. There are a few conventions. Options which identify a single input map are usually "map=", not "raster=" or "vector=". In the case of an input and output map the convention is: "input=xx output=yy". By passing the 'help' option to existing GRASS commands, it is likely that you will find other conventions. The desire is to make it as easy as possible for the user to remember (or guess correctly) what the command line syntax is for a given command.

12.17 String Manipulation Functions

This section describes some routines which perform string manipulation. Strings have the usual C meaning: a NULL terminated array of characters.

These next 3 routines copy characters from one string to another.

char * *copy strings*
G_strcpy (char *dst, char *src)

> Copies the **src** string to **dst** up to and including the NULL which terminates the **src** string. Returns **dst.**

char * *copy strings*
G_strncpy (char *dst, char *src, int n)

> Copies at most **n** characters from the **src** string to **dst.** If **src** contains less than **n** characters, then only those characters are copied. A NULL byte is added at the end of **dst.** This implies that **dst** should be at least **n**+1 bytes long. Returns **dst. Note.** This routine varies from the UNIX strncpy( ) in that G_strncpy( ) ensures that **dst** is NULL terminated, while strncpy( ) does not.

char * *concatenate strings*
G_strcat (char *dst, char *src)

> Appends the **src** string to the end of the **dst** string, which is then NULL terminated. Returns **dst.**

These next 3 routines remove unwanted white space from a single string.

remove unnecessary white space

char *
G_squeeze (char *s)

Leading and trailing white space is removed from the string **s** and internal white space which is more than one character is reduced to a single space character. White space here means spaces, tabs, linefeeds, newlines, and formfeeds. Returns **s.**

remove leading/training white space

void
G_strip (char *s)

Leading and trailing white space is removed from the string **s.** White space here means only spaces and tabs. There is no return value.

char *
G_chop (char *s)

Chop leading and trailing white spaces: space, \f, \n, \r, \t, \v - returns pointer to string

The next routines replaces character(s) from string.

replace character(s)

char *
G_strchg (char *bug, char character, char new)

Replace all occurencies of character in string bug with new. Returns changed string

This next routine copies a string to allocated memory.

copy string to allocated memory

char *
G_store (char *s)

This routine allocates enough memory to hold the string **s**, copies **s** to the allocated memory, and returns a pointer to the allocated memory.

The next 2 routines convert between upper and lower case.

convert string to lower case

char *
G_tolcase (char *s)

Upper case letters in the string **s** are converted to their lower case equivalent. Returns **s.**

char *
G_toucase (char *s)

convert string to upper case

Lower case letters in the string **s** are converted to their upper case equivalent. Returns **s.**

And finally a routine which gives a printable version of control characters.

char *
G_unctrl (unsigned char c)

printable version of control character

This routine returns a pointer to a string which contains an English-like representation for the character **c.** This is useful for nonprinting characters, such as control characters. Control characters are represented by ctrl-C, e.g., control A is represented by ctrl-A. 0177 is represented by DEL/RUB. Normal characters remain unchanged.

This routine is useful in combination with *G_intr_char* for printing the user's interrupt character:

```
char G_intr_char( );
char *G_unctrl( );
fprintf(stdout, "Your interrupt character is %s\n", G_unctrl(G_intr_char( )));
```

Note. G_unctrl( ) uses a hidden static buffer which is overwritten from call to call.

FOLLOWING new FUNCTIONS need to be merged into the text:

int
G_trim_decimal (char *buf)

trim

this routine remove trailing zeros from decimal number for example: 23.45000 would come back as 23.45

char *
G_index (str, delim)

delimiter

position of delimiter

??? char *
G_rindex (str, delim)

???

int
G_strcasecmp(char *a, char *b

string compare ignoring case (upper or lower) returns: -1 if a<b
0 if a==b
1 if a>b

char *
G_strstr(char *mainString, char *subString)

Return a pointer to the first occurrence of subString in mainString, or NULL if no occurrences are found

char *
G_strdup(char *string)

Return a pointer to a string that is a duplicate of the string given to G_strdup. The duplicate is created using malloc. If unable to allocate the required space, NULL is returned.

12.18 Enhanced UNIX Routines

A number of useful UNIX library routines have side effects which are sometimes undesirable. The routines here provide the same functions as their corresponding UNIX routine, but with different side effects.

12.18.1 Running in the Background

The standard UNIX fork( ) routine creates a child process which is a copy of the parent process. The fork( ) routine is useful for placing a module into the background. For example, a module

that gathers input from the user interactively, but knows that the processing will take a long time, might want to run in the background after gathering all the input. It would fork( ) to create a child process, the parent would exit( ) allowing the child to continue in the background, and the user could then do other processing.

However, there is a subtle problem with this logic. The fork( ) routine does not protect child processes from keyboard interrupts even if the parent is no longer running. Keyboard interrupts will also kill background processes that do not protect themselves.[33] Thus a module which puts itself in the background may never finish if the user interrupts another module which is running at the keyboard.

The solution is to fork( ) but also put the child process in a process group which is different from the keyboard process group. G_fork( ) does this.

pid_t
G_fork ( )

create a protected child process

> This routine creates a child process by calling the UNIX fork( ) routine. It also changes the process group for the child so that interrupts from the keyboard do not reach the child. It does not cause the parent to exit( ).
> G_fork( ) returns what fork( ) returns: -1 if fork( ) failed; otherwise 0 to the child, and the process id of the new child to the parent.
> **Note.** Interrupts are still active for the child. Interrupts sent using the *kill* command, for example, will interrupt the child. It is simply that keyboard-generated interrupts are not sent to the child.

12.18.2 Partially Interruptible System Call

The UNIX system( ) call allows one program, the parent, to execute another UNIX command or module as a child process, wait for that process to complete, and then continue. The problem addressed here concerns interrupts. During the standard system( ) call, the child process inherits its responses to interrupts from the parent. This means that if the parent is ignoring interrupts, the child will ignore them as well. If the parent is terminated by an interrupt, the child will be also.

However, in some cases, this may not be the desired effect. In a menu environment where the parent activates menu choices by running commands using the system( ) call, it would be nice if the user could interrupt the command, but not terminate the menu module itself. The G_system( ) call allows this.

int

run a shell level command

[33] Programmers who use /bin/sh know that programs run in the background (using & on the command line) are not automatically protected from keyboard interrupts. To protect a command that is run in the background, /bin/sh users must do **nohup** *command* &. Programmers who use the /bin/csh (or other variants) do not know, or forget that the C-shell automatically protects background processes from keyboard interrupts.

G_system (command)

The shell level **command** is executed. Interrupt signals for the parent module are ignored during the call. Interrupt signals for the **command** are enabled. The interrupt signals for the parent are restored to their previous settings upon return.
G_system( ) returns the same value as system( ), which is essentially the exit status of the **command.** See UNIX manual system(1) for details.

12.18.3 ENDIAN test

To test if the user's machine is little or big ENDIAN, the following function is provided:

test little ENDIAN

int
G_is_little_endian()

Test if machine is little or big endian.
Returns:
1 little endian
0 big endian

12.19 Unix Socket Functions

The following provide a simplified interface for interprocess communication via Unix sockets. The caller need only be concerned with the path to the socket file and the various file descriptors for socket connections. The caller does not need to worry about handling socket structures – which, unlike internet sockets, have little utility once a file descriptor has been opened on a connection. All socket functions in the GIS library have a **G_sock** prefix. One should keep in mind that unix sockets connections can both be read from and written to. Also, it is possible for calls to **read()** and **write()** to read or write fewer bytes than specified. Hence, looping calls may be required to read or write all of the data. The **read()** will still normally block if there is nothing to read, so a zero byte return value typically means the connection has been closed. The **write()** function typically returns immediately. [34]

makes full socket path

char *
G_sock_get_fname (char *name)

Takes a simple **name** for a communication channel and builds the full path for a sockets file with that **name**. The path as of this writing (2000-02-18) is located in the temporary directory for the user's current mapset (although this will likely change). A **NULL** pointer is returned if the function fails for some reason. The

[34] *see* W. Richard Stevens. 1997. **UNIX network programming: Volume 1** 2^{nd} edition. Prentice Hall

caller is responsible for freeing the memory of the returned string when it is no longer needed.

int
G_sock_exists (char *name)

does the socket exist

Takes the full path to a unix socket; determines if the file exists; and if the file exists whether it is a socket file or not. Returns a non-zero value if the file exists and is a socket file. Otherwise it returns zero.

int
G_sock_bind (char *name)

binds the socket

Takes the full path to a unix socket and attempts to bind a file descriptor to the path **name**. If successful it will return the file descriptor. Otherwise, it returns -1. The socket file must not already exist. If it does, this function will fail and set the global **errno** to **EADDRINUSE**. Other error numbers may be set if the call to **bind()** fails. Server programs wishing to bind a socket should test if the socket file they wish to use already exists. And, if so, they may try to connect to the socket to see if it is in use. If it is not in use, such programs may then call **unlink()** or **remove()** to delete the file before calling **G_sock_bind()**. It is important that server processes do not just delete existing socket files without testing the connection. Doing so may make another server process unreachable (i.e. you will have hijacked the other server's communication channel). Server processes must call **G_sock_bind()** prior to calling **G_sock_listen()** and **G_sock_accept()**.

int
G_sock_listen (int fd, unsigned int queue)

listen on a socket

Takes the file descriptor returned by a successful call to **G_sock_bind()** and the length of the the listen queue. A successful call will return 0, while a failed call will return -1. The global **errno** will contain the error number corresponding to the reason for the failure. The queue length should never be zero. Some systems may interpret this to mean that no connections should be queued. Other systems may add a fudge factor to the queue length that the caller specifies. Servers that don't want additional connections queued should **close()** the listening file descriptor after a successful call to **G_sock_accept()**. This function is a simple wrapper around the system **listen()** function.

int
G_sock_accept (int fd)

accept a connection on the listening socket

Takes the file descriptor returned by a successful call to **G_sock_bind()**, for which a successful call to **G_sock_listen()** has also been made, and waits for an incoming connection. When a connection arrives, the file descriptor for the connection is returned. This function normally blocks indefinitely. However, an interrupt like **SIGINT** may cause this function to return without a valid connection. In this case, the return value will be -1 and the global error number will be set to **EINTR**. Servers should handle this possibility by calling **G_sock_accept()** again. A typical server might have a call to **fork()** after a successful return from **G_sock_accept()**. A server might also use **select()** to see if an a connection is ready prior to calling **G_sock_accept()**. This function is a simple wrapper around the system's **accept()** function, with the second and third arguments being **NULL**.

make a connection to a server process

int
G_sock_connect (char *name)

Takes the full path to a socket file and attempts to make a connection to a server listening for connections. If successful, the file descriptor for the socket connection is returned. Otherwise, -1 is returned and the global **errno** may be set. This function and **G_sock_get_fname()** are the only functions a client program really needs to worry about. If the caller wants to be sure that the global error number was set from an unsuccessful call to this function, she should zero **errno** prior to the call. Failures due to a non-existent socket file or a path name that exceeds system limits, will not change the global error number.

12.19.1 Trivial Socket Server Example

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include "gis.h"

int main (int argc, char *argv[])
{
    int listenfd, rwfd;
    char *path;
    pid_t pid;

    /* Path is built using server's name */
    if (NULL == (path = G_sock_get_fname (argv[0])))
        exit (EXIT_FAILURE);

    /* Make sure another instance isn't running */
    if (G_sock_exists (path))
    {
        if ((listenfd = G_sock_connect (path)) != -1)
        {
            close (listenfd);
```

```
            exit (EXIT_FAILURE);
        }
        remove (path);
    }

    /* Bind the socket */
    if ((listenfd = G_sock_bind (path)) < 0)
        exit (EXIT_FAILURE);

    /* Begin listening on the socket */
    if (G_sock_listen (listenfd, 1) != 0)
        exit (EXIT_FAILURE);

    /* Loop forever waiting for connections */
    for (;;)
    {
        if ((rwfd = G_sock_accept (listenfd)) < 0)
        {
            if (errno == EINTR)
                continue;
        }
        else
            exit (EXIT_FAILURE);

        /* Fork connection */
        if ((pid = fork()) == 0)
        {
            char c;
            /* child closes listenfd */
            close (listenfd);
            while (read (rwfd, &c, 1) > 0)
                write (rwfd, &c, 1);
            close (rwfd);
            return 0;
        }
        else if (pid > 0)
        {
            /* parent closes rwfd
             * a well behaved server would limit
             * the number of forks.
             */
            close (rwfd);
        }
        else
            exit (EXIT_FAILURE);

    }
    G_free (path);
    return 0;
}
```

12.20 Miscellaneous

A number of general purpose routines have been provided.

current date and time
char *
G_date ( )

Returns a pointer to a string which is the current date and time. The format is the same as that produced by the UNIX *date* command.

get a line of input (detect ctrl-z)
char *
G_gets (char *buf)

This routine does a *gets* ( ) from stdin into **buf.** It exits if end-of-file is detected. If stdin is a tty (i.e., not a pipe or redirected) then ctrl-z is detected. Returns 1 if the read was successful, or 0 if ctrl-z was entered.

Note. This is very useful for allowing a module to reprompt when a module is restarted after being stopped with a ctrl-z. If this routine returns 0, then the calling module should reprint a prompt and call *G_gets* ( ) again. For example:

```
char buf[1024];
do
fprintf(stdout, "Enter some input:  ")  ;
while ( !  G_gets(buf) ) ;
```

user's home directory
char *
G_home ( )

Returns a pointer to a string which is the full path name of the user's home directory.

return interrupt char
char
G_intr_char ( )

This routine returns the user's keyboard interrupt character. This is the character that generates the SIGINT signal from the keyboard.

See also *G_unctr* for converting this character to a printable format.

print percent complete messages

int
G_percent (int n, int total, int incr)

This routine prints a percentage complete message to stderr. The percentage complete is (**n/ total**)*100, and these are printed only for each **incr** percentage. This is perhaps best explained by example:

```
  # include <stdio.h>
int row;
int nrows;
nrows = 1352; /* 1352 is not a special value -
example only */
fprintf (stderr, "Percent complete:  ");
for (row = 0; row < nrows; row++)
  G_percent (row, nrows, 10);
```

This will print completion messages at 10% increments; i.e., 10%, 20%, 30%, etc., up to 100%. Each message does not appear on a new line, but rather erases the previous message. After 100%, a new line is printed.

return module name

char *
G_program_name ( )

Routine returns the name of the module as set by the call to *G_gisinit.*

user's name

char *
G_whoami ( )

Returns a pointer to a string which is the user's login name.

ask a yes/no question

int
G_yes (char *question, int default)

This routine prints a **question** to the user, and expects the user to respond either yes or no. (Invalid responses are rejected and the process is repeated until the user answers yes or no.)
The **default** indicates what the RETURN key alone should mean. A **default** of 1 indicates that RETURN means yes, 0 indicates that RETURN means no, and -1 indicates that RETURN alone is not a valid response.
The **question** will be appended with "(y/n) ", and, if **default** is not -1, with "[y] " or "[n] ", depending on the **default.**
G_yes ( ) returns 1 if the user said yes, and 0 if the user said no.

12.21 GIS Library Data Structures

Some of the data structures, defined in the ”gis.h” header file and used by routines in this library, are described in the sections below.

12.21.1 struct Cell_head

The raster header data structure is used for two purposes. It is used for raster header information for map layers. It also used to hold region values. The structure is:

```
struct Cell_head
{
int format; /* number of bytes per cell */
int compressed; /* compressed(1) or not compressed(0) */
int rows, cols; /* number of rows and columns */
int proj; /* projection */
int zone; /* zone */
double ew_res; /* east-west resolution */
double ns_res; /* north-south resolution */
double north; /* northern edge */
double south; /* southern edge */
double east; /* eastern edge */
double west; /* western edge */
};
```

The *format* and *compressed* fields apply only to raster headers. The *format* field describes the number of bytes per raster data value and the *compressed* field indicates if the raster file is compressed or not. The other fields apply both to raster headers and regions. The geographic boundaries are described by *north, south, east* and *west.* The grid resolution is described by *ew_res* and *ns_res.* The cartographic projection is described by *proj* and the related zone for the projection by *zone.* The *rows* and *cols* indicate the number of rows and columns in the raster file, or in the region. See *5.3 Raster Header Format (p. 29)* for more information about raster headers, and *9.1 Region (p. 61)* for more information about regions.

The routines described in *12.10.1 Raster Header File (p. 113)* use this structure.

12.21.2 struct Categories

The *Categories* structure contains a title for the map layer, the largest category in the map layer, an automatic label generation rule for missing labels, and a list of category labels.

The structure is declared: *struct Categories* .

This structure should be accessed using the routines described in *12.10.2 Raster Category File (p. 114)*.

12.21.3 struct Colors

The color data structure holds red, green, and blue color intensities for raster categories. The structure has become so complicated that it will not be described in this manual.

The structure is declared: *struct Colors* .

The routines described in *12.10.3 Raster Color Table (p. 116)* must be used to store and retrieve color information using this structure.

12.21.4 struct History

The *History* structure is used to document raster files. The information contained here is for the user. It is not used in any operational way by GRASS. The structure is:

```
# define MAXEDLINES 50
# define RECORD_LEN 80
struct History
{
char mapid[RECORD_LEN];
char title[RECORD_LEN];
char mapset[RECORD_LEN];
char creator[RECORD_LEN];
char maptype[RECORD_LEN];
char datsrc_1[RECORD_LEN];
char datsrc_2[RECORD_LEN];
char keywrd[RECORD_LEN];
int edlinecnt;
char edhist[MAXEDLINES][RECORD_LEN];
};
```

The *mapid* and *mapset* are the raster file name and mapset, *title* is the raster file title, *creator* is the user who created the file, *maptype* is the map type (which should always be ”raster”), *datasrc_1* and *datasrc_2* describe the original data source, *keywrd* is a one-line data description and *edhist* contains *edlinecnt* lines of user comments.

The routines described in *12.10.3.4.1 Raster History File (p. 121)* use this structure. However, there is very little support for manipulating the contents of this structure. The programmer must manipulate the contents directly.

Note. Some of the information in this structure is not meaningful. For example, if the raster file is renamed, or copied into another mapset, the *mapid* and *mapset* will no longer be correct. Also the *title* does not reflect the true raster file title. The true title is maintained in the category file.

Warning. This structure has remained unchanged since the inception of GRASS. There is a good possibility that it will be changed or eliminated in future releases.

12.21.5 struct Range

The *Range* structure contains the minimum and maximum values which occur in a raster file.

The structure is declared: *struct Range* .

The routines described in *12.10.4 Raster Range File (p. 122)* should be used to access this structure.

12.22 Loading the GIS Library

The library is loaded by specifying $(GISLIB) in the Gmakefile. The following example is a complete Gmakefile which compiles code that uses this library:

```
  Gmakefile for $(GISLIB)
  OBJ = main.o sub1.o sub2.o
pgm:  $(OBJ) $(GISLIB)
  $(CC) $(LDFLAGS) -o $@ $(OBJ) $(GISLIB)
  $(GISLIB): # in case the library changes
```

See *11 Compiling and Installing GRASS Modules (p. 69)* for a complete discussion of Gmakefiles.

12.23 Timestamp functions

```
#include "gis.h"
```

This structure is defined in gis.h, but there should be no reason to access its elements directly:

```
struct TimeStamp
DateTime dt[2]; /* two datetimes */
int count;
;
```

Using the G_*_timestamp routines reads/writes a timestamp file in the cell_misc/rastername or dig_misc/vectorname mapset element.

A TimeStamp can be one DateTime, or two DateTimes representing a range. When preparing to write a TimeStamp, the programmer should use one of:

G_set_timestamp to set a single DateTime

G_set_timestamp_range to set two DateTimes.

int
G_read_raster_timestamp (char *name, char *mapset, struct TimeStamp *ts)

Read raster timestamp

Returns 1 on success. 0 or negative on error.

int
G_read_vector_timestamp (char *name, char *mapset, struct TimeStamp *ts)

Read vector timestamp

Returns 1 on success. 0 or negative on error.

int
G_get_timestamps (struct TimeStamp *ts, DateTime *dt1, DateTime *dt2, int *count)

copy TimeStamp into Datetimes

Use to copy the TimeStamp information into Datetimes, so the members of struct TimeStamp shouldn't be accessed directly.
count=0 means no datetimes were copied
count=1 means 1 datetime was copied into dt1
count=2 means 2 datetimes were copied

int
G_init_timestamp (struct TimeStamp *ts)

Sets ts->count = 0, to indicate no valid DateTimes are in TimeStamp.

int
G_set_timestamp (struct TimeStamp *ts, DateTime *dt)

Copies a single DateTime to a TimeStamp in preparation for writing. (overwrites any existing information in TimeStamp)

int
G_set_timestamp_range (struct TimeStamp *ts, DateTime *dt1, DateTime *dt2)

Copies two DateTimes (a range) to a TimeStamp in preparation for writing. (overwrites any existing information in TimeStamp)

int
G_write_raster_timestamp (char *name, struct TimeStamp *ts)

Returns:
1 on success.
-1 error - can't create timestamp file
-2 error - invalid datetime in ts

int
G_write_vector_timestamp (char *name, struct TimeStamp *ts)

Returns:
1 on success.
-1 error - can't create timestamp file
-2 error - invalid datetime in ts

int
G_format_timestamp (struct TimeStamp *ts, char *buf)

Returns:
1 on success
-1 error

int
G_scan_timestamp (struct TimeStamp *ts, char *buf)

Returns:
1 on success
-1 error

int
G_remove_raster_timestamp (char *name)

Only timestamp files in current mapset can be removed
Returns:
0 if no file
1 if successful
-1 on fail

int
G_remove_vector_timestamp (char *name)

Only timestamp files in current mapset can be removed
Returns:
0 if no file
1 if successful
-1 on fail

int *read grid3*
G_read_grid3_timestamp (char *name,char *mapset, struct TimeStamp *ts) *timestamp*

Returns 1 on success. 0 or negative on error.

int *remove grid3*
G_remove_grid3_timestamp (char *name) *timestamp*

Only timestamp files in current mapset can be removed
Returns:
0 if no file
1 if successful
-1 on fail

int *write grid3*
G_write_grid3_timestamp (char *name, struct TimeStamp *ts) *timestamp*

Returns:
1 on success.
-1 error - can't create timestamp file
-2 error - invalid datetime in ts

See *23 DateTime Library (p. 349)* for a complete discussion of GRASS datetime routines.

12.24 GRASS GIS Library Overview

Contents of directory src/libes/:

```
D          : display library
bitmap     : bitmap library for X Window Bitmaps
btree      : binary tree library
bwidget    : tcl/tk extra library
coorcnv    : coordinate conversion and datum support library
datetime   : DateTime library
dbmi       : database management interface database drivers
dig_atts   : library to read and write from attribute files
digitizer  : raw library for general digitizer support
display    : library for CELL driver
dlg        : library to manage DLG files
dspf       : G3D display files library
g3d        : G3D raster volume library
geom       : geometrical calculations, long-integer arithmetics, delaunay
             triangulation, planesweep algorithm, simulation of simplicity
             and others
gis        : main GRASS library
gmath      : generic mathematical functions (matrix, fft etc.) added by
             D D Gray (later to be extended, BLAS/LAPACK library)
ibtree     : integer clone of binary tree library (btree)
icon       : icon library (required for PS/Paint icons, read, write,
             manipulate icons)
image3     : extra imagery library
imagery    : imagery library
libimage   : image library required for SG3d/Iris, saving images in SGIs
             rgb-format
linkm      : linked list memory manager
lock       : locking mechanism for GRASS monitors and files
ogsf       : ported gsurf library (required for NVIZ)
proj       : PROJ4.4.x projection library
raster     : GRASS raster library
rowio      : row in/out library
rst_gmsl   : library for interpolation with regularized splines with tension
segment    : segment library
unused     : maybe treasures here?
vask       : Curses management library
vect32     : GRASS vector library
vect32_64  : new approach for a 32/64bit GRASS vector library (to be
             integrated in vect32)
```

13 Vector Library

13.1 Introduction to Vector Library

The *Vector Library* provides the GRASS programmer with routines to process the binary vector files. It is assumed that the reader has read *4 Database Structure (p. 17)* for a general description of GRASS databases, and *6 Vector Maps (p. 41)* for details about vector file formats in GRASS.

The routines in the *Vector Library* are presented in functional groupings, rather than in alphabetical order. The order of presentation will, it is hoped, provide a better understanding of how the library is to be used, as well as show the interrelationships among the various routines. Note that a good way to understand how to use these routines is to look at the source code for GRASS modules which use them.[1]

Note. All routines and global variables in this library, documented or undocumented, start with one of the following prefixes **Vect_** or **V1_** or **V2_** or **dig_.**[2] To avoid name conflicts, programmers should not create variables or routines in their own modules which use this prefix.[3]

An alphabetic index is provided in *A.4 Appendix D: Index to Vector Library (p. 447)*.

13.1.1 Include Files

The following file contains definitions and structures required by some of the routines in this library. The programmer should therefore include this file in any code that uses this library:[4].

#include "Vect.h"

13.1.2 Vector Arc Types

A complete discussion of GRASS vector terminology can be found in *6.1 What is a Vector Map Layer? (p. 41)* and the reader should review that section. Briefly, vector data are stored as arcs representing linear, area, or point features. These arc types are coded as LINE, AREA, and DOT

[1] *Some of these programs are v.in.ascii, v.out.ascii, v.to.rast, d.vect, p.map, and v.patch.*

[2] All names beginning with V#_ (where # is any primary number) are also reserved for future use.

[3] **Warning.** There are also four additional global variables and/or routines which do NOT *begin with these prefixes: debugf, Lines_In_Memory, Mem_L*ine_Ptr, and Mem_curr_position.

[4] The GRASS compilation process, described in *11 Compiling and Installing GRASS Modules (p. 69)*, automatically tells the C compiler how to find this and other GRASS header files. But also see *13.9 Loading the Vector Library (p. 230)*

respectively, (and are # defined in the file ”dig_defines.h”, which is automatically included by the file ”Vect.h”).

13.1.3 Levels of Access

There are two levels of read access to these vector files:

Level One provides simple access to the arc information contained in the vector files. There is no access to category or topology information at this level.[5]

Level Two provides full access to all the information contained in the vector file and its support files, including line, category, node, and area information. This level requires more from the programmer, more memory, and longer startup time.[6]

Note. Higher levels of access are planned, so when checking success return codes for a particular level of access (when calling Vect_open_old( ) for example), the programmer should use > = instead of = = for compatibility with future releases.

13.2 Changes in 4.0 from 3.0

[GRASS 5: Can we remove this or shorten?]

The 4.0 Vector Library changed significantly from the **Dig Library** used with GRASS 3.1. Below is an overview of why the changes were made, and how to module using the **new Vect Library.**

13.2.1 Problem

The Digit Library was a collage of subroutines created for developing the map development programs. Few of these subroutines were actually designed as a user access library. They required individuals to assume too much responsibility and control over what happened to the data file. Thus when it came time to change vector data file formats for GRASS 4.0, many modules also required modification. By using the FILE * structure as the tag for files, there was no means of expansion since the FILE * structure is not modifiable by GRASS. For example, there was no way to open supporting files since all that was passed in to dig_init( ) was a FILE * which had no file name associated with it.

The two different access levels for 3.0 vector files provided very different ways of calling the library; they offered little consistency for the user.

[5] The category information is available through the dig_att library, but there are no data structures to link them to the spatial features at this level.

[6] The routines in this library which process *arcs* are named using the word *line.* They should be named using the word *arc* instead. Since that would require modifying a lot of existing code, the names have not been changed.

The Digit Library was originally designed to only have one file open for read or write at a time. Although it was possible in some cases to get around this, one restriction was the global *head* structure. Since there was only one instance of this, there could only be one copy of that information, and thus, only one open vector file.

13.2.2 Solution

The solution to these problems was to design a new user library as an interface to the vector data files. This new library was designed to provide a simple consistent interface, which hides as much of the details of the data format as possible. It also can be extended for future enhancements without the need to change existing programs.

13.2.3 Approach

A new [new in GRASS 4.x] library VECTLIB has been created. It provides routines for opening, closing, reading, and writing vector files, as well as several support functions. The Digit Library has been removed, so that all existing modules will have to be converted to use the new library. Those routines that existed in the Digit Library and were not affected by these changes continue to exist in unmodified form, and are now included in the VECTLIB. Most of the commonly used routines have been discarded, and replaced by the new Vector routines.

The token that is used to identify each map is the *Map_info* structure. This structure was used by level two functions in GRASS 3.1. It maintains all information about an individual open file. This structure must be passed to most Vector subroutines. The *head* structure has gone away, as has the global instance of it which was also called *head.* All modules which used this global structure must now create their own local version of it. The structure that replaced *struct head* is *struct dig_head.*

There are still two levels of interface to the vector files (future releases may include more). Level one provides access only to arc (i.e. polyline) information and to the type of line (AREA, LINE, DOT). Level two provides access to polygons (areas), attributes, and network topology. There is now only one subroutine to open a file for read, Vect_open_old( ) and one for write, Vect_open_new( ). Vect_open_old( ) attempts to open a vector file at the highest possible level of access. It will return the number of the level at which it opened. Vect_open_new( ) *always* opens at level 1 only. If you require that a file be opened at a lower level (e.g. one), you can call the routine Vect_set_open_level(1); Vect_open_old( ) will then either open at level one or fail. If you instead require the highest level access possible, you should not use Vect_set_open_level( ), but instead check the return value of Vect_open_old( ) to make sure it is greater than or equal to the lowest level at which you need access. This allows for future levels to work without need for module change.

13.2.4 Implementation

There are two macros set up for use in the Gmakefile to support the Vector library:

EXTRA_CFLAGS = $(VECT_INCLUDE)

must exist in the Gmakefile for any module which uses the Vector library. NOTE: GRASS 3.1 required the line ***-I$(DIG_INCLUDE)**; do NOT use **-I** with **VECT_INCLUDE.***

$(VECTLIB)

is to be used on the link statement to include the vector library.[7] This basically replaces the **$(DIGLIB)** macro from 3.1. Currently this macro represents two different libraries which are in directories: *src/mapdev/Vlib* and *src/mapdev/diglib.* These will probably change in the future and are given only for aid in looking up include files or functions.

The basic format of a module that reads a vector file is:

```
#include "Vect.h" /* new include file */
struct Map_info Map; /* Map info */
struct line_pnts *Points; /* Poly-Line data */
G_gisinit (argv[0]); /* init GIS lib */
if (0 > Vect_open_old (&Map, name, mapset)) /* open file */
G_fatal_error ("Cannot open vector file");
Points = Vect_new_line_struct (\h'|209350u');
while (0 < Vect_read_next_line (&Map, Points)) /* loop reading
*/
{/* each line */
/* do something with Points */
}
Vect_destroy_line_struct (Points); /* remove allocation */
Vect_close (&Map); /* close up */
```

All *Vect_* routines work in the same way on any lev el of access unless otherwise noted. Routines that are designed for one level of access or another have the naming convention V#_ where # is an integer (currently 1 or 2). For example: V2_line_att ( ) is only valid with level 2 or higher access, and will return the attribute number for a specified line.

13.3 Opening and closing vector maps

open existing vector map

int
Vect_open_old (struct Map_info *Map, char *name, char *mapset)

This routine opens the vector map **name** in **mapset** for reading. It returns the level of successful open, or a negative value on failure.

open new vector map

int

[7] Because there are two libraries involved and there are some cross-dependencies, it may occasionally be necessary to specify $(VECTLIB) twice on the link statement in order to resolve all references.

Vect_open_new (struct Map_info *Map, char *name)

This routine opens the vector map **name** in the current mapset for writing. It returns the level of successful open which must be one, or a negative value on failure.

int
Vect_set_open_level (int level)

specify level for opening map

This routine allows you to specify at which **level** the map is to be opened. It is recommended that it only be used to force opening at level one(1). There is no return value.

int
Vect_close (struct Map_info *Map)

close a vector map

This routine closes an open vector map and cleans up the structures associated with it. It MUST be called before exiting the module. When used in conjunction with *Vect_open_new*, it will cause the final writing of the vector header before closing the vector map. The header data is in the structure **Map**->head, which also changed in 4.0 to be an instance of the structure (struct dig_head head) instead of a pointer (struct dig_head *head).

13.4 Reading and writing vector maps

int
Vect_read_next_line (struct Map_info *Map, struct line_pnts *Points)

read next vector line

This is the primary routine for reading through a vector map. It simply reads the next line from the map into the **Points** structure. This routine should not be used in conjunction with any other read_line routine. Return value is type of line, or
-2 on EOF
-1 on Error (generally out of memory)

This routine is modified by:
Vect_rewind()
Vect_set_constraint_region()
Vect_set_constraint_type()

This routine normally only reads lines that are ”alive” (as opposed to dead or deleted) from the vector map. This can be overridden using *Vect_set_constraint_type(Map,-1).*

rewind vector map for re-reading

int
Vect_rewind (struct Map_info *Map)

This routine will reset the read pointer to the beginning of the map. This only affects the routine*Vect_read_next_line.*

set restricted region to read vector arcs from

int
Vect_set_constraint_region (struct Map_info *Map, double n, double s, double e, double w)

This routine will set a restriction on reading only those lines which fall entirely *or* partially in the specified rectangular region. *Vect_read_next_line* is currently the only routine affected by this, and it does NOT currently cause line clipping. Constraints affect only the **Map** specified. They do not affect any other Maps that may be open.

specify types of arcs to read

int
Vect_set_constraint_type (struct Map_info *Map, int type)

This routine will set a restriction on reading only those lines which match the **type**s specified. This can be any combination of **type**s bitwise OR'ed together. For example: LINE | AREA would exclude any DOT (or future NEAT) line **type**s. *Vect_read_next_line* is currently the only routine affected by this. If **type** is set to -1, all lines will be read including deleted or *dead* lines. An example of this exists in *v.out.ascii*, where it is desirable to include all lines, (ie. not exclude deleted lines).
Constraints affect only the **Map** specified. They do not affect any other Maps that may be open.

unset any vector read constraints

int
Vect_remove_constraints (struct Map_info *Map)

Removes all constraints currently affecting **Map.**

write out arc to vector map

long
Vect_write_line (struct Map_info *Map, int type, struct line_pnts *Points)

This routine will write out a line to a vector map which has previously been opened for write by *Vect_open_new.* The **type** of line is one of: AREA, LINE, DOT It returns the offset into the file where the line started. If this number is negative or 0, there was an error.

int
V1_read_line (struct Map_info *Map, struct line_pnts *Points, long offset)

read vector arc by specifying offset

This routine will read a line from the vector map at the specified **offset** in the file. This function is available at level 1 or higher.
Return value is the same as *Vect_read_next_line.*

int
V2_read_line (struct Map_info *Map, struct line_pnts *Points, int line)

read vector arc by specifying line id

This routine will read a line from the vector map at the specified line index in the map. Refer to *V2_num_lines* for number of lines in the map. This function is available at level 2 or higher.
Return value is the same as *Vect_read_next_line.*

13.5 Data Structures

struct line_pnts *
Vect_new_line_struct (void)

create new initialized line points structure

This routine MUST be used to initialize any and all line_pnts structures. You cannot simply create a line_pnts structure and pass its address to routines. It must first be initialized. The correct procedure is: struct line_pnts *Points;

```
Points = Vect_new_line_struct( );
```

This routine will print an error message and exit with an error on out of memory condition.

int
Vect_destroy_line_struct (struct line_pnts *Points)

deallocate line points structure space

This routine will free any memory created for a line_pnts structure. You can use this when you are done with a line_pnts struct or when you need to free up unused memory. The structure must have been created by *Vect_new_line_struct.*

13.6 Data Conversion

int
Vect_copy_xy_to_pnts (struct line_pnts *Points, double *x, double *y, int n)

convert xy arrays to line_pnts structure

Since all Vector library routines require the use of the line_pnts structure, and many modules out there work with X and Y arrays of points, this routine was to created to copy **n** data pairs from **x,y** arrays to a line_pnts structure **Points.** It handles all memory management.

convert line_pnts structure to xy arrays

int
Vect_copy_pnts_to_xy (struct line_pnts *Points, double *x, double *y, int *n)

Since all Vector library routines require the use of the line_pnts structure, and many modules out there work with X and Y arrays of points, this routine was to created to copy data from a line_pnts structure **Points** into user supplied **x,y** arrays. The **x,y** arrays MUST each be large enough to hold **Points.**n_points doubles or memory corruption will occur. No bounds checking is done. Upon return **n** will contain the number of points copied.

copy vector header struct data

int
Vect_copy_head_data (struct dig_head *from, struct dig_head *to)

This routine should be used to copy data **from** one dig_head struct **to** another. For example, if a 3.1 module used to fill in a local dig_head struct and then called dig_write_head_binary( ) (which no longer exists), you would now call *Vect_copy_head_data (local_head, &Map.head)* to copy the data into the Map structure which would then be written out when *Vect_close* was called. This routine must used because there are now other fields in the head structure which applications programmers should not touch, and this module copies only those fields which are available to the programmer.

13.7 Miscellaneous

get point inside area and outside all islands

int
Vect_get_point_in_area (struct Map_info *Map, int area, double *X, double *Y)

This routine examines the area with index value *area* in a vector map and places in *X* and *Y*, the X- and Y- positions respectively of a suitable area point.
The method is as follows: Take a line and intersect it with the polygon and any islands. Sort the list of X values from these intersections. This will be a list of segments alternating IN/OUT/IN/OUT of the polygon. Pick the largest IN segment and take the midpoint.
Note. This function, works only for level 2 or higher. It returns 0 on success or -1 on error.

int
Vect_get_point_in_poly (struct line_pnts *Points, double *X, double *Y)

get point inside polygon

This routine finds a suitable area point in the ring described by the line struct *Points*, but without reference to any islands that may be present inside the area. This routine is useful where prior topological information on an area is not available.
The return value is 0 on success, -1 on failure.

int
Vect_get_point_in_poly_isl (struct line_pnts *APoints, struct line_pnts **IPoints, int n_isles, double *X, double *Y)

get point inside polygon but outside the islands specifiled in IPoints.

This routine finds a suitable area point in the ring described by the line struct *APoints* and the interior islands specified in the array of line structs *IPoints*. This routine is useful where prior topological information on an area is not available, as for example in import filters.
The method for finding the area point is as follows: Take a line and intersect it with the polygon and any islands. Sort the list of X values from these intersections. This will be a list of segments alternating IN/OUT/IN/OUT of the polygon. Pick the largest IN segment and take the midpoint.
The return value is 0 on success, -1 on failure.

int
Vect_point_in_islands (struct Map_info *Map, int area, double x, double y)

find if a given point in an area is inside an island.

This routine determines if the given point *(x,y)* is located inside an island of area with index *area*.
The return value is 0 (false) if this condition is not met, 1 (true) if it is, and -1 if the procedure fails.
Note. This function, works only for level 2 or higher.

int
Vect_get_area_points (struct Map_info *Map, int area, struct line_pnts *Points)

get defining points for area polygon

This routine replaces dig_get_area( ). It will fill in the Points structure with the list of points which describe an area in clockwise order. Points at the junction of constituent lines are included only once.
Note. This function, works only for level 2 or higher. It returns the number of points or -1 on error.

int
Vect_get_isle_points (struct Map_info *Map, int isle, struct line_pnts *Points)

get defining points for isle perimeter

This routine fills in the Points structure with the list of points which describe the perimeter of a primary island in counter-clockwise order. Points at the junction of constituent lines are included only once.
Note. This function, works only for level 2 or higher. It returns the number of points or -1 on error.

get number of arcs in vector map

int
V2_num_lines (struct Map_info *Map)

Return total number of lines in the vector **Map.**
Note. The line indexes are numbers from 1 to n, where n is the number of lines in the vector map, as returned by this routine.

get number of areas in vector map

int
V2_num_areas (struct Map_info *Map)

Return total number of areas in the vector **Map.**
Note. The area indexes are numbers from 1 to n, where n is the number of areas in the vector map, as returned by this routine.

get number of islands in vector map

int
V2_num_islands (struct Map_info *Map)

Return total number of islands in the vector **Map.**
Note. The islands indexes are numbers from 1 to n, where n is the number of islands in the vector map, as returned by this routine.

get attribute number of arc

int
V2_line_att (struct Map_info *Map, int line)

Given arc index n, return its attribute number.
Returns 0 if not labeled or on error.

get attribute number of area

int
V2_area_att (struct Map_info *Map, int area)

Given area index n, return its attribute number.
Returns 0 if not labeled or on error.

get area info from id

int
V2_get_area (struct Map_info *Map, int n, P_AREA **pa)

Given area index n, the P_AREA information for the area is read into a private structure. A pointer to this structure is placed in pa. The pointer pa is valid until the next call to this routine. Note that *pa does not need to point to anything on entry. Returns 0 if found, or negative on error.

get bounding box of area

int
V2_get_area_bbox (struct Map_info *Map, int area, double *n, double *s, double *e, double *w)

Given area index n, set n (north, s (south), e (east), and w (west) to the values of the bounding box for the area.
Returns 0 if ok, or -1 on error.

get bounding box of arc

int
V2_get_line_bbox (struct Map_info *Map, int line, double *n, double *s, double *e, double *w)

Given arc index n, set n (north, s (south), e (east), and w (west) to the values of the bounding box for the arc.
Returns 0 if ok, or -1 on error.

print header info to stdout

int
Vect_print_header (struct Map_info *Map)

This routine replaces dig_print_header( ), and simply displays selected information from the header. Namely organization, map name, source date, and original scale.

get open level of vector map

int
Vect_level (struct Map_info *Map)

This routine will return the number of the level at which a **Map** is opened at or -1 if **Map** is not opened.[8]

[8] The levels correspond to the 3.1 level 1 and level 2 accesses.

13.8 Routines that remain from GRASS 3.1

find which area point is in

int
dig_point_to_area (struct Map_info *Map, double x, double y)

Returns the index of the area containing the point **x,y**, or 0 if none found.

is point in area?

double
dig_point_in_area (struct Map_info *Map, double x, double y, P_AREA *pa)

Given a filled P_AREA structure **pa**, determine if **x,y** is within the area. The structure **pa** can be filled with *V2_get_area.*
Returns 0.0 if **x,y** is not in the area, the positive minimum distance to the nearest area edge if **x,y** is inside the area, or -1.0 on error.

find which arc point is closest to

int
dig_point_to_line (struct Map_info *Map, double x, double y, char type)

Returns the index of the arc which is nearest to the point **x,y.** The point **x,y** must be within the arc's bounding box. Set **type** to a combination of LINE, AREA, or DOT (eg. LINE | AREA), or (char) -1 if you want to search all arc types.

find distance of point to line

int
dig_check_dist (struct Map_info *Map, intn, double x, double y, double *d)

Computes **d**, the square of the minimum distance from point **x,y** to arc **n**R. Returns the number of the segment that was closest, or -1 on error. The segment number, in combination with *V2_read_line* can be used to determine the endpoints of the closest line-segment in the arc.

13.9 Loading the Vector Library

The library is loaded by specifying $(VECTLIB) in the Gmakefile. The following example is a complete Gmakefile which compiles code that uses this library:

Gmakefile using $(VECTLIB)

```
OBJ = main.o sub1.o sub2.o

EXTRA_CFLAGS = $(VECT_INCLUDE)

$(BIN_MAIN_CMD)/pgm: $(OBJ) $(VECTLIB) $(GISLIB)

$(CC) $(LDFLAGS) -o $@ $(OBJ) $(VECTLIB) $(GISLIB)

$(VECTLIB): # in case the library changes
```

Note. EXTRA_CFLAGS tells the C compiler where additional # include files are located. This is necessary since the required # include files do not currently live in the normal GRASS # include directory. Notice that **-I** must **not** be provided before the $(VECT_INCLUDE)

Note. Because $(VECTLIB) currently references two distinct libraries, on occasion it may be necessary to specify it twice on the link command because of library cross-references.

See *11 Compiling and Installing GRASS Modules (p. 69)* for a complete discussion of Gmakefiles.

14 Imagery Library

14.1 Introduction to Imagery Library

The *Imagery Library* was created for version 3.0 of GRASS to support integrated image processing directly in GRASS. It contains routines that provide access to the *group* database structure which was also introduced in GRASS 3.0 for the same purpose.[1] It is assumed that the reader has read *4 Database Structure (p. 17)* for a general description of GRASS databases, *8 Image Data: Groups (p. 53)* for a description of imagery groups, and *5 Raster Maps (p. 27)* for details about map layers in GRASS. The routines in the *14 Imagery Library (p. 233)* are presented in functional groupings, rather than in alphabetical order. The order of presentation will, it is hoped, provide a better understanding of how the library is to be used, as well as show the interrelationships among the various routines. Note that a good way to understand how to use these routines is to look at the source code for GRASS modules which use them.[2] Most routines in this library require that the header file "imagery.h" be included in any code using these routines.[3] Therefore, programmers should always include this file when writing code using routines from this library:

#include "imagery.h"

This header file includes the "gis.h" header file as well.

Note. All routines and global variables in this library, documented or undocumented, start with the prefix **I_.** To avoid name conflicts, programmers should not create variables or routines in their own modules which use this prefix.

An alphabetic index is provided in *A.5 Appendix E: Index to Imagery Library (p. 448)*

14.2 Group Processing

The group is the key database structure which permits integration of image processing in GRASS.

[1] Since this is a new library, it is expected to grow. It is hoped that image analysis functions will be added to complement the database functions already in the library.

[2] See *8.4 Imagery Modules (p. 58)* for a list of some imagery programs.

[3] The GRASS compilation process, described in *11 Compiling and Installing GRASS Modules (p. 69)*, automatically tells the C compiler how to find this and other GRASS header files.

14.2.1 Prompting for a Group

The following routines interactively prompt the user for a group name in the current mapset.[4] In each, the **prompt** string will be printed as the first line of the full prompt which asks the user to enter a group name. If **prompt** is the empty string "", then an appropriate prompt will be substituted. The name that the user enters is copied into the **group** buffer.[5] These routines have a built-in 'list' capability which allows the user to get a list of existing groups.

The user is required to enter a valid group name, or else hit the RETURN key to cancel the request. If the user enters an invalid response, a message is printed, and the user is prompted again. If the user cancels the request, 0 is returned; otherwise, 1 is returned.

prompt for an existing group

int
I_ask_group_old (char *prompt, char *group)

Asks the user to enter the name of an existing **group** in the current mapset.

prompt for new group

int
I_ask_group_new (char *prompt, char *group)

Asks the user to enter a name for a **group** which does not exist in the current mapset.

prompt for any valid group name

int
I_ask_group_any (char *prompt, char *group)

Asks the user to enter a valid **group** name. The **group** may or may not exist in the current mapset.

Note. The user is not warned if the **group** exists. The programmer should use *I_find_group* to determine if the **group** exists.

Here is an example of how to use these routines. Note that the programmer must handle the 0 return properly:

```
char group[50];
if ( !  I_ask_group_any ("Enter group to be processed", group))
exit(0);
```

[4] This library only works with groups in the current mapset. Other mapsets, even those in the user's mapset search path, are ignored.

[5] The size of **group** should be large enough to hold any GRASS file name. Most systems allow file names to be quite long. It is recommended that **name** be declared *char group.*

14.2.2 Finding Groups in the Database

Sometimes it is necessary to determine if a given group already exists. The following routine provides this service:

int
I_find_group (char *group) *does group exist?*

Returns 1 if the specified **group** exists in the current mapset; 0 otherwise.

14.2.3 REF File

These routines provide access to the information contained in the REF file for groups and subgroups, as well as routines to update this information. They use the *Ref* structure, which is defined in the "imagery.h" header file; see *14.4 Imagery Library Data Structures (p. 239)*.

The contents of the REF file are read or updated by the following routines:

int
I_get_group_ref (char *group, struct Ref *ref) *read group REF file*

Reads the contents of the REF file for the specified **group** into the **ref** structure.
Returns 1 if successful; 0 otherwise (but no error messages are printed).

int
I_put_group_ref (char *group, struct Ref *ref) *write group REF file*

Writes the contents of the **ref** structure to the REF file for the specified **group.**
Returns 1 if successful; 0 otherwise (and prints a diagnostic error).
Note. This routine will create the **group**, if it does not already exist.

int
I_get_subgroup_ref (char *group, char *subgroup, struct Ref *ref) *read subgroup REF file*

Reads the contents of the REF file for the specified **subgroup** of the specified **group** into the **ref** structure.
Returns 1 if successful; 0 otherwise (but no error messages are printed).

write subgroup REF file

int
I_put_subgroup_ref (char *group, char *subgroup, struct Ref *ref)

Writes the contents of the **ref** structure into the REF file for the specified **subgroup** of the specified **group.**
Returns 1 if successful; 0 otherwise (and prints a diagnostic error).
Note. This routine will create the **subgroup**, if it does not already exist.

These next routines manipulate the *Ref* structure:

initialize Ref structure

int
I_init_group_ref (struct Ref *ref)

This routine initializes the **ref** structure for other library calls which require a *Ref* structure. This routine must be called before any use of the structure can be made.
Note. The routines I_get_group_ref and I_get_subgroup_ref call this routine automatically.

add file name to Ref structure

int
I_add_file_to_group_ref (char *name, char *mapset, struct Ref *ref)

This routine adds the file **name** and **mapset** to the list contained in the **ref** structure, if it is not already in the list. The **ref** structure must have been properly initialized. This routine is used by programs, such as *i.maxlik*, to add to the group new raster files created from files already in the group.
Returns the index into the *file* array within the **ref** structure for the file after insertion; see *14.4 Imagery Library Data Structures (p. 239)*.

copy Ref lists

int
I_transfer_group_ref_file (struct Ref *src, int n, struct Ref *dst)

This routine is used to copy file names from one *Ref* structure to another. The name and mapset for file **n** from the **src** structure are copied into the **dst** structure (which must be properly initialized).
For example, the following code copies one *Ref* structure to another:

```
struct Ref src,dst;
int n;
/* some code to get information into src */
...
I_init_group_ref (&dst);
for (n = 0; n < src.nfiles; n++)
I_transfer_group_ref_file (&src, n, &dst);
```

This routine is used by *i.points* to create the REF file for a subgroup.

int
I_free_group_ref (struct Ref *ref) *free Ref structure*

This routine frees memory allocated to the **ref** structure.

14.2.4 TARGET File

The following two routines read and write the TARGET file.

int
I_get_target (char *group, char *location, char *mapset) *read target information*

Reads the target **location** and **mapset** from the TARGET file for the specified group. Returns 1 if successful; 0 otherwise (and prints a diagnostic error). This routine is used by *i.points* and *i.rectify* and probably should not be used by other programs.
Note. This routine does **not** validate the target information.

int
I_put_target (char *group, char *location, char *mapset) *write target information*

Writes the target **location** and **mapset** to the TARGET file for the specified **group.**
Returns 1 if successful; 0 otherwise (but no error messages are printed).
This routine is used by *i.target* and probably should not be used by other programs.
Note. This routine does **not** validate the target information.

14.2.5 POINTS File

The following routines read and write the POINTS file, which contains the image registration control points. This file is created and updated by the module *i.points*,and read by *i.rectify.*

These routines use the *Control_Points* structure, which is defined in the "imagery.h" *header file*; see *14.4 Imagery Library Data Structures (p. 239).*

Note. The interface to the *Control_Points* structure provided by the routines below is incomplete. A routine to initialize the structure is needed.

int
I_get_control_points (char *group, struct Control_Points *cp) *read group control points*

Reads the control points from the POINTS file for the **group** into the **cp** structure. Returns 1 if successful; 0 otherwise (and prints a diagnostic error).
Note. An error message is printed if the POINTS file is invalid, or does not exist.

add new control point

int
I_new_control_point (struct Control_Points *cp, double e1, double n1, double e2, double n2, int status)

Once the control points have been read into the **cp** structure, this routine adds new points to it. The new control point is given by **e1** (column) and **n1** (row) on the image, and the **e2** (east) and **n2** (north) for the target database. The value of **status** should be 1 if the point is a valid point; 0 otherwise.[6]

write group control points

int
I_put_control_points (char *group, struct Control_Points *cp)

Writes the control points from the **cp** structure to the POINTS file for the specified group.
Note. Points in **cp** with a negative *status* are not written to the POINTS file.

14.3 Loading the Imagery Library

The library is loaded by specifying $(IMAGERYLIB) in the Gmakefile. The following example is a complete Gmakefile which compiles code that uses this library:

Gmakefile for $(IMAGERYLIB)

```
OBJ = main.o sub1.o sub2.o

pgm: $(OBJ) $(IMAGERYLIB) $(GISLIB)

$(CC) $(LDFLAGS) -o $@ $(OBJ) $(IMAGERYLIB) $(GISLIB)

$(IMAGERYLIB): # in case the library changes

$(GISLIB): # in case the library changes
```

Note. This library must be loaded with $(GISLIB) since it uses routines from that library. See *12 GIS Library (p. 79)* or details on that library. See *11 Compiling and Installing GRASS Modules (p. 69)* for a complete discussion of Gmakefiles.

[6] Use of this routine implies that the point is probably good, so **status** should be set to 1.

14.4 Imagery Library Data Structures

Some of the data structures in the ”imagery.h” header file are described below.

14.4.1 struct Ref

The *Ref* structure is used to hold the information from the REF file for groups and subgroups. The structure is:

```
struct Ref

{

int nfiles; /* number of REF files */

struct Ref_Files

{

char name[30]; /* REF file name */

char mapset[30]; /* REF file mapset */

} *file;

struct Ref_Color

{

unsigned char *table; /* color table for min-max values */

unsigned char *index; /* data translation index */

unsigned char *buf; /* data buffer for reading color file */

int fd; /* for image i/o */

CELL min, max; /* min,max CELL values */

int n; /* index into Ref_Files */

} red, grn, blu;

};
```

The *Ref* structure has *nfiles* (the number of raster files), *file* (the name and mapset of each file), and *red,grn,blu* (color information for the group or subgroup [7]).

[7] The *red,grn,blu* elements are expected to change as the imagery code develops. Do **not** reference them. Pretend they do not exist.

There is no function interface to the *nfiles* and *file* elements in the structure. This means that the programmer must reference the elements of the structure directly.[8] The name and *mapset for the i th file are file[i].name, and file[i].mapset.*

For example, to print out the names of the raster files in the structure:

int i;

struct Ref ref;

.

..

/* some code to get the REF file for a group into **ref** */

..

for (i = 0; i < ref.nfiles; i++)

fprintf(stdout, "%s in %s\n", ref.file[i].name, ref.file[i].mapset);

14.4.2 struct Control_Points

The *Control_Points* structure is used to hold the control points from the group POINTS file. The structure is:

struct

Control_Points

{

int count; /* number of control points */

double *e1; /* image east (column) */

double *n1; /* image north (row) */

double *e2; /* target east */

double *n2; /* target north */

int *status; /* status of control point */

};

The number of control points is *count.*

[8] The *nfiles* and *file* elements are not expected to change in the future.

Control point *i* is *e1* [i], *n1* [i], *e2* [i], *n2* [i], and its status is *status* [i].

15 Raster Graphics Library

15.1 Introduction

The *Raster Graphics Library* provides the programmer with access to the GRASS graphics devices. **All video graphics calls are made through this library (directly or indirectly).** No standard/portable GRASS video graphics module drives any video display directly. This library provides a powerful, but limited number of graphics capabilities to the programmer. The tremendous benefit of this approach is seen in the ease with which GRASS graphics applications modules port to new machines or devices. Because no device-dependent code exists in application programs, virtually all GRASS graphics modules port without modification. Each graphics device must be provided a driver (or translator program). At run-time, GRASS graphics modules rendezvous with a user-selected driver module. Two significant prices are paid in this approach to graphics: 1) graphics displays run significantly slower, and 2) the programmer does not have access to fancy (and sometimes more efficient) resident library routines that have been specially created for the device.

This library uses a couple of simple concepts. First, there is the idea of a current screen location. There is nothing which appears on the graphics monitor to indicate the current location, but many graphic commands begin their graphics at this location. It can, of course, be set explicitly. Second, there is always a current color. Many graphic commands will do their work in the currently chosen color. The programmer always works in the screen coordinate system. Unlike many graphics libraries developed to support CAD, there is no concept of a world coordinate system. The programmer must address graphics requests to explicit screen locations. This is necessary, especially in the interest of fast raster graphics.

The upper left hand corner of the screen is the origin. The actual pixel rows and columns which define the edge of the video surface are returned with calls to *R_screen_left*, *R_screen_rite, R_screen_bot, and R_screen_top.*

Note. All routines and global variables in this library, documented or undocumented, start with the prefix **R_.** To avoid name conflicts, programmers should not create variables or routines in their own modules which use this prefix.

An alphabetic index is provided in *A.7 Appendix G: Index to Raster Graphics Library (p. 452).*

15.2 Connecting to the Driver

Before any other graphics calls can be made, a successful connection to a running and selected graphics driver must be made.

initialize graphics int **R_open_driver ( )**

Initializes connection to current graphics driver. Refer to GRASS User's Manual entries on the *d.mon* command. If connection cannot be made, the application module sends a message to the user stating that a driver has not been selected or could not be opened. Note that only one application module can be connected to a graphics driver at once.
After all graphics have been completed, the driver should be closed.

terminate graphics int **R_close_driver ( )**

This routine breaks the connection with the graphics driver opened by R_open_driver( ).

15.3 Colors

GRASS is highly dependent on color for distinguishing between different categories. No graphic patterning is supported in any automatic way. There are two color modes. Fixed color refers to set and immutable color look-up tables on the hardware device. In some cases this is necessary because the graphics device does not contain programmer definable color look-up tables (LUT). Floating colors use the LUTs of the graphics device often in an interactive mode with the user. The basic impact on the user is that under the fixed mode, multiple maps can be displayed on the device with apparently no color interference between maps. Under float mode, the user may interactively manipulate the hardware color tables (using modules such as *d.colors*). Other than the fact that in float mode no more colors may be used than color registers available on the user's chosen driver, there are no other programming repercussions.

select fixed color table int **R_color_table_fixed ( )**

Selects a fixed color table to be used for subsequent color calls. It is expected that the user will follow this call with a call to erase and reinitialize the entire graphics screen.
Returns 0 if successful, non-zero if unsuccessful.

int
R_color_table_float ( )

select floating color table

Selects a float color table to be used for subsequent color calls. It is expected that the user will follow this call with a call to erase and reinitialize the entire graphics screen.
Returns 0 if successful, non-zero if unsuccessful.

Colors are set using integer values in the range of 0-255 to set the **red, green**, and **blue** intensities. In float mode, these values are used to directly modify the hardware color look-up tables and instantaneously modify the appearance of colors on the monitor. In fixed mode, these values modify secondary look-up tables in the devices driver module so that the colors involved point to the closest available color on the device.

int
R_reset_color (unsigned char red, unsigned char green, unsigned char blu, int num)

define single color

Sets color number **num** to the intensities represented by **red, green**, and **blue.**

int
R_reset_colors (int min, int max, unsigned char *red, unsigned char *green, unsigned char *blue)

define multiple colors

Sets color numbers **min** through **max** to the intensities represented in the arrays **red, green, and blue.**

int
R_color (int color)

select color

Selects the **color** to be used in subsequent draw commands.

int
R_standard_color (int color)

select standard color

Selects the standard **color** to be used in subsequent draw commands. The **color** value is best retrieved using *D_translate_color.* See *16 Display Graphics Library (p. 255).*

int
R_RGB_color (int red, int green, int blue)

select color

When in float mode (see *R_color_table_float*), this call selects the color most closely matched to the **red, green**, and **blue** intensities requested. These values must be in the range of 0-255.

15.4 Basic Graphics

Several calls are common to nearly all graphics systems. Routines exist to determine screen dimensions, as well as routines for moving, drawing, and erasing.

bottom of screen

int
R_screen_bot ( )

Returns the pixel row number of the bottom of the screen.

top of screen

int
R_screen_top ( )

Returns the pixel row number of the top of the screen.

screen left edge

int
R_screen_left ( )

Returns the pixel column number of the left edge of the screen.

screen right edge

int
R_screen_rite ( )

Returns the pixel column number of the right edge of the screen.

move current location

int
R_move_abs (int x, int y)

Move the current location to the absolute screen coordinate **x,y.** Nothing is drawn on the screen.

move current location

int
R_move_rel (int dx, int dy)

Shift the current screen location by the values in **dx** and **dy**:

```
Newx = Oldx + dx;
Newy = Oldy + dy;
```

Nothing is drawn on the screen.

draw line

int
R_cont_abs (int x, int y)

Draw a line using the current color, selected via *R_color*, from the current location to the location specified by **x,y.** The current location is updated to **x,y.**

draw line

int
R_cont_rel (int dx, int dy)

Draw a line using the current color, selected via *R_color*, from the current location to the relative location specified by **dx** and **dy.** The current location is updated:

```
Newx = Oldx + dx;
Newy = Oldy + dy;
```

fill a box

int
R_box_abs (int x1, int y1, int x2, int y2)

A box is drawn in the current color using the coordinates **x1,y1** and **x2,y2** as opposite corners of the box. The current location is updated to **x2,y2.**

fill a box

int
R_box_rel (int dx, int dy)

A box is drawn in the current color using the current location as one corner and the current location plus **dx** and **dy** as the opposite corner of the box. The current location is updated:

```
Newx = Oldx + dx;
Newy = Oldy + dy;
```

erase screen

int
R_erase ( )

Erases the entire screen to black.

flush graphics

int
R_flush ( )

Send all pending graphics commands to the graphics driver. This is done automatically when graphics input requests are made.

synchronize graphics

int
R_stabilize ( )

Send all pending graphics commands to the graphics driver and cause all pending graphics to be drawn (provided the driver is written to comply). This routine does more than *R_flush* and in many instances is the more appropriate routine fo the two to use.

15.5 Poly Calls

In many cases strings of points are used to describe a complex line, a series of dots, or a solid polygon. Absolute and relative calls are provided for each of these operations.

draw a series of dots

int
R_polydots_abs (int *x, int *y, int num)

Pixels at the **num** absolute positions in the **x** and **y** arrays are turned to the current color. The current position is left updated to the position of the last dot.

draw a series of dots

int
R_polydots_rel (int *x, int *y, int num)

Pixels at the **num** relative positions in the **x** and **y** arrays are turned to the current color. The first position is relative to the starting current location; the succeeding positions are then relative to the previous position. The current position is updated to the position of the last dot.

draw a closed polygon

int
R_polygon_abs (int *x, int *y,int num)

The **num** absolute positions in the **x** and **y** arrays outline a closed polygon which is filled with the current color. The current position is left updated to the position of the last point.

draw a closed polygon

int
R_polygon_rel (int *x, int *y, int num)

The **num** relative positions in the **x** and **y** arrays outline a closed polygon which is filled with the current color. The first position is relative to the starting current location; the succeeding positions are then relative to the previous position. The current position is updated to the position of the last point.

draw an open polygon

int
R_polyline_abs (int *x, int *y, int num)

The **num** absolute positions in the **x** and **y** arrays are used to generate a multisegment line (often curved). This line is drawn with the current color. The current position is left updated to the position of the last point.
Note. It is not assumed that the line is closed, i.e., no line is drawn from the last point to the first point.

draw an open polygon

int
R_polyline_rel (int *x, int *y, int num)

The **num** relative positions in the **x** and **y** arrays are used to generate a multisegment line (often curved). The first position is relative to the starting current location; the succeeding positions are then relative to the previous position. The current position is updated to the position of the last point. This line is drawn with the current color.
Note. No line is drawn between the last point and the first point.

15.6 Raster Calls

GRASS requires efficient drawing of raster information to the display device. These calls provide that capability.

draw a raster

int
R_raster (int num, int nrows, int withzero, int *raster)

Starting at the current position, the **num** colors represented in the **raster** array are drawn for **nrows** consecutive pixel rows. The **withzero** flag is used to indicate whether 0 values are to be treated as a color (1) or should be ignored (0). If ignored, those screen pixels in these locations are not modified. This option is useful for graphic overlays.

initialize graphics int
R_set_RGB_color (unsigned char red[256], unsigned char green[256], unsigned char blue[256])

The three 256 member arrays, **red, green**, and **blue,** establish look-up tables which translate the raw image values supplied in *R_RGB_raster* to color intensity values which are then displayed on the video screen. These two commands are tailor-made for imagery data coming off sensors which give values in the range of 0-255.

draw a raster int
R_RGB_raster (int num, int nrows, unsigned char *red, unsigned char *green, unsigned char *blue, int withzero)

This is useful only in fixed color mode (see *R_color_table_fixed*). Starting at the current position, the **num** colors represented by the intensities described in the **red, green**, and **blue** arrays are drawn for **nrows** consecutive pixel rows. The raw values in these arrays are in the range of 0-255. They are used to map into the intensity maps which were previously set with *R_set_RGB_color.* The **withzero** flag is used to indicate whether 0 values are to be treated as a color (1) or should be ignored (0). If ignored, those screen pixels in these locations are not modified. This option is useful for graphic overlays.

15.7 Text

These calls provide access to built-in vector fonts which may be sized and clipped to the programmer's specifications.

set text clipping frame int
R_set_window (int top, int bottom, int left, int right)

Subsequent calls to *R_text* will have text strings clipped to the screen frame defined by **top, bottom, left, right.**

choose font int

R_font (char *font)

Set current font to **font.** Available fonts are:

Font Name	Description
cyrilc	cyrillic
gothgbt	Gothic Great Britain triplex
gothgrt	Gothic German triplex
gothitt	Gothic Italian triplex
greekc	Greek complex
greekcs	Greek complex script
greekp	Greek plain
greeks	Greek simplex
italicc	Italian complex
italiccs	Italian complex small
italict	Italian triplex
romanc	Roman complex
romancs	Roman complex small
romand	Roman duplex
romanp	Roman plain
romans	Roman simplex
romant	Roman triplex
scriptc	Script complex
scripts	Script simplex

int *set text size*
R_text_size (int width, int height)

Sets text pixel width and height to **width** and **height.**

int *write text*
R_text (char *text)

Writes **text** in the current color and font, at the current text width and height, starting at the current screen location.

int *get text extents*
R_get_text_box (char *text, int *top, int *bottom, int *left, int *right)

The extent of the area enclosing the **text** is returned in the integer pointers **top, bottom, left**, and **right.** No text is actually drawn. This is useful for capturing the text extent so that the text location can be prepared with proper background or border.

15.8 GRASS font support

The current mechanism of GRASS 5.0 font support is this (all files are in the directory src/display/devices/lib unless stated otherwise):

1. A client calls R_font(), which sends the filename ($GISBASE/fonts/font_name) to the display driver using the FONT command. See src/libes/raster/Font.c.
2. The display driver receives the FONT command and the filename. See command.c.
3. It passes the filename to Font_get() (Font_get.c), which passes it to init_font() (font.c), which reads the file into memory.
4. A client draws text by calling R_text (src/libes/raster/Text.c), which sends the string to the display driver using the TEXT command.
5. The display driver receives the TEXT command and the string. See command.c (again).
6. It passes the string to Text() (Text.c) which calls soft_text() with the string and several stored parameters.
7. soft_text() (Text2.c) calls drawchar() (same file), to draw each character.
8. drawchar() calls get_char_vects() (font.c) to retrieve the actual vector definitions. It then draws the character using text_move() and text_draw() (same file), which use the Move_abs() and Cont_abs() functions (these are implemented separately by each display driver, e.g. XDRIVER).

15.9 User Input

The raster library provides mouse (or other pointing device) input from the user. This can be accomplished with a pointer, a rubber-band line or a rubber-band box. Upon pressing one of three mouse buttons, the current mouse location and the button pressed are returned.

get mouse location using pointer

int
R_get_location_with_pointer (int *nx, int *ny, int *button)

A cursor is put on the screen at the location specified by the coordinate found at the **nx,ny** pointers. This cursor tracks the mouse (or other pointing device) until one of three mouse buttons are pressed. Upon pressing, the cursor is removed from the screen, the current mouse coordinates are returned by the **nx** and **ny** pointers, and the mouse button (1 for left, 2 for middle, and 3 for right) is returned in the **button** pointer.

get mouse location using a line

int
R_get_location_with_line (int x, int y, int *nx, int *ny, int *button)

Similar to *R_get_location_with_pointer* except the pointer is replaced by a line which has one end fixed at the coordinate identified by the **x,y** values. The other end of the line is initialized at the coordinate identified by the **nx,ny** pointers. This end then tracks the mouse until a button is pressed. The mouse button (1 for left, 2 for middle, and 3 for right) is returned in the **button** pointer.

get mouse location using a box

int
R_get_location_with_box (int x, int y, int *nx, int *ny, int *button)

Identical to *R_get_location_with_line* except a rubber-band box is used instead of a rubber-band line.

15.10 Loading the Raster Graphics Library

The library is loaded by specifying $(RASTERLIB) in the Gmakefile. The following example is a complete Gmakefile which compiles code that uses this library:

Gmakefile for $(RASTERLIB)

```
OBJ = main.o sub1.o sub2.o

pgm: $(OBJ) $(RASTERLIB) $(GISLIB)

$(CC) $(LDFLAGS) -o $@ $(OBJ) $(RASTERLIB) $(GISLIB)

$(RASTERLIB): # in case the library changes

$(GISLIB): # in case the library changes
```

Note. This library must be loaded with $(GISLIB) since it uses routines from that library. See *12 GIS Library (p. 79)* for details on that library. This library is usually loaded with the $(DISPLAYLIB). See *16 Display Graphics Library (p. 255)* for details on that library.

See *11 Compiling and Installing GRASS Modules (p. 69)* for a complete discussion of Gmakefiles.

16 Display Graphics Library

16.1 Introduction

This library provides a wide assortment of higher level graphics commands which in turn use the graphics raster library primitives. It is highly recommended that this section be used to understand how some of the GRASS graphics commands operate. Such modules like *d.vect, d.graph*, and *d.rast* demonstrate how these routines work together. The routines fall into four basic sets: 1) frame[1] creation and management, 2) coordinate conversion routines, 3) specialized efficient raster display routines, and 4) assorted miscellaneous routines like pop-up menus and line clipping.

Note. All routines and global variables in this library, documented or undocumented, start with the prefix **D_.** To avoid name conflicts, programmers should not create variables or routines in their own modules which use this prefix.

An alphabetic index is provided in *A.6 Appendix F: Index to Display Graphics Library (p. 449)*.

16.2 Library Initialization

The following routine performs a required setup procedure. Its use is encouraged and simplifies the use of this library.

int
D_setup (int clear)

initialize/create a frame

This routine performs a series of initialization steps for the current frame. It also creates a full screen frame if there is no current frame. The **clear** flag, if set to 1, tells this routine to clear any information associated with the frame: graphics as well as region information.
This routine relieves the programmer of having to perform the following idiomatic function call sequence

```
struct Cell_head region;
```

[1] In previous versions of GRASS, these were called graphic windows. To reduce ambiguity for users, these are now called graphic *frames*. However, for backward compatiblity (and general programmer confusion) the routines described here still retain their original names - the word "window" is still used in the naming of these routines.

```
  char name[128];
  int T,B,L,R;
  /* get current frame, create full_screen frame if no current
frame */
  if (D_get_cur_wind(name)) {
  T =R_screen_top( );
  B =R_screen_bot( );
  L =R_screen_left( );
  R =R_screen_rite( );
  strcpy (name, "full_screen");
  D_new_window (name, T, B, L, R);
  }
  if (D_set_cur_wind(name)) G_fatal_error("Current graphics frame
not available") ;
  if (D_get_screen_window(&T, &B, &L, &R)) G_fatal_error("Getting
graphics coordinates") ;
  /* clear the frame, if requested to do so */
  if (clear) {
  D_clear_window( );
  R_standard_color(D_translate_color("black"));
  R_box_abs (L, T, R, B);
  }
  /* Set the map region associated with graphics frame */
  G_get_set_window (&region);
  if (D_check_map_window(&region)) G_fatal_error("Setting graphics
coordinates"); if(G_set_window (&region) < 0) G_fatal_error ("Invalid
graphics region coordinates");
  /* Determine conversion factors */
  if (D_do_conversions(&region, T, B, L, R)) G_fatal_error("Error
calculating graphics-region conversions")
  /* set text clipping, for good measure, and set a starting location
*/
  R_set_window (T,B,L,R);
  R_move_abs(0,0);
  D_move_abs(0,0);
```

16.3 Frame Management

The following set of routines create, destroy, and otherwise manage graphic frames.

create new graphics frame

int
D_new_window (char *name, int top, int bottom, int left, int right)

Creates a new frame **name** with coordinates **top, bottom, left**, and **right.** If **name** is the empty string "" (i.e., ***name** = = 0), the routine returns a unique string in **name.**

int
D_set_cur_wind (char *name)

set current graphics frame

Selects the frame **name** to be the current frame. The previous current frame (if there was one) is outlined in grey. The selected current frame is outlined in white.

int
D_get_cur_wind (char *name)

identify current graphics frame

Captures the name of the current frame in string **name.**

int
D_show_window (int color)

outlines current frame

Outlines current frame in **color.** Appropriate colors are found in $GISBASE/src/D/libes/colors.h[2] and are spelled with lowercase letters.

int
D_get_screen_window (int *top, int *bottom, int *left, int *right)

retrieve current frame coordinates

Returns current frame's coordinates in the pointers **top, bottom, left**, and **right.**

int
D_check_map_window (struct Cell_head *region)

assign/retrieve current map region

Graphics frames can have GRASS map regions associated with them. This routine passes the map **region** to the current graphics frame. If a GRASS region is already associated with the graphics frame, its information is copied into **region** for use by the calling module. Otherwise **region** is associated with the current graphics frame.
Note this routine is called by *D_setup.*

int
D_reset_screen_window (int top, int bottom, int left, int right)

resets current frame position

Re-establishes the screen position of a frame at the location specified by **top, bottom, left, and right.**

[2] $GISBASE is the directory where GRASS is installed. See *10.1 UNIX Environment (p. 65)* for details.

give current time to frame

int
D_timestamp ( )

Timestamp the current frame. This is used primarily to identify which frames are on top of other, specified frames.

erase current frame

int
D_erase_window ( )

Erases the frame on the screen using the currently selected color.

remove a frame

int
D_remove_window ( )

Removes any trace of the current frame.

clears information about current frame

int
D_clear_window ( )

Removes all information about the current frame. This includes the map region and the frame content lists.

16.4 Frame Contents Management

This special set of graphics frame management routines maintains lists of frame contents.

add command to frame display list

int
D_add_to_list (char *string)

Adds **string** to list of screen contents. By convention, **string** is a command string which could be used to recreate a part of the graphics contents. This should be done for all screen graphics except for the display of raster maps. The *D_set_cell_name* routine,the *D_set_dig_name* routine and the *D_set_site_name* routine are used for this special case.

add raster map name to display list

int
D_set_cell_name (char *name)

Stores the raster[3] map **name** in the information associated with the current frame.

int
D_get_cell_name (char *name)

retrieve raster map name

Returns the **name** of the raster map associated with the current frame.

int
D_set_dig_name (char *name)

add vector map name to display list

Stores the vector map **name** in the information associated with the current frame.

int
D_get_dig_name (char *name)

retrieve vector map name

Returns the **name** of the vector map associated with the current frame.

int
D_set_site_name (char *name)

add site map name to display list

Stores the site map **name** in the information associated with the current frame.

int
D_get_site_name (char *name)

retrieve site map name

Returns the **name** of the site map associated with the current frame.

Note: **R_pad_freelist( )** should be called to free memory allocated before.

int
D_clear_window ( )

clear frame display lists

Removes all display information lists associated with the current frame.

[3] As with the change from *window* to *frame*, GRASS 4.0 changed word usage from *cell* to *raster.* For compatibility with existing code, the routines have not changed their names.

16.5 Coordinate Transformation Routines

These routines provide coordinate transformation information. GRASS graphics modules typically work with the following three coordinate systems:

Coordinate system Origin

Array upperleft (NW)

Display screen upper left (NW)

Earth lower left (SW)

Display screen coordinates are the physical coordinates of the display screen and are referred to as *x* and *y.* Earth region coordinates are from the GRASS database regions and are referred to as *east* and *north.* Array coordinates are the columns and rows relative to the GRASS region and are referred to as *column* and *row.* The routine *D_do_conversions* is called to establish the relationships between these different systems. Then a wide variety of accompanying calls provide access to conversion factors as well as conversion routines.

initialize conversions

int
D_do_conversions (struct Cell_head *region, int top, int bottom, int left, int right)

The relationship between the earth **region** and the **top, bottom, left**, and **right** screen coordinates is established, which then allows conversions between all three coordinate systems to be performed.
Note this routine is called by *D_setup.*

In the following routines, a value in one of the coordinate systems is converted to the equivalent value in a different coordinate system. The routines are named based on the coordinates systems involved. Display screen coordinates are represented by *d*, array coordinates by *a,* and earth coordinates by *u* (which used to stand for UTM).

earth to array (north)

double
D_u_to_a_row (double north)

Returns a *row* value in the array coordinate system when provided the corresponding **north** value in the earth coordinate system.

earth to array (east

double
D_u_to_a_col (double east)

Returns a *column* value in the array coordinate system when provided the corresponding **east** value in the earth coordinate system.

double
D_a_to_d_row (double row)

array to screen (row)

Returns a *y* value in the screen coordinate system when provided the corresponding **row** value in the array coordinate system.

double
D_a_to_d_col (double column)

array to screen (column)

Returns an *x* value in the screen coordinate system when provided the corresponding **column** value in the array coordinate system.

double
D_u_to_d_row (double north)

earth to screen (north)

Returns a *y* value in the screen coordinate system when provided the corresponding **north** value in the earth coordinate system.

double
D_u_to_d_col (double east)

earth to screen (east)

Returns an *x* value in the screen coordinate system when provided the corresponding **east** value in the earth coordinate system.

double
D_d_to_u_row (double y)

screen to earth (y)

Returns a *north* value in the earth coordinate system when provided the corresponding **y** value in the screen coordinate system.

double
D_d_to_u_col (double x)

screen to earth (x)

Returns an *east* value in the earth coordinate system when provided the corresponding **x** value in the screen coordinate system.

screen to array (y)
double
D_d_to_a_row (double y)

Returns a *row* value in the array coordinate system when provided the corresponding **y** value in the screen coordinate system.

screen to array (x)
double
D_d_to_a_col (double x)

Returns a *column* value in the array coordinate system when provided the corresponding **x** value in the screen coordinate system.

reset raster color value
int
D_reset_color (CELL data, int r, int g, int b)

Modifies the hardware colormap, provided that the graphics are not using fixed more colors. The hardware color register corresponding to the raster data value is set to the combined values of **r,g,b.** This routine may only be called after a call to *D_set_colors. D_reset_color* is for use by modules such as d.colors. Returns 1 if the hardware colormap was updated, 0 if not. A 0 value will result if either a fixed color table transition is in effect, or because the data is not in the color range set by the call *D_set_colors.*

verify a range of colors
int
D_check_colormap_size (CELL min, CELL max, int *ncolors)

This routine determines if the range of colors fits into the hardware colormap. If it does, then the colors can be loaded directly into the hardware colormap and color toggling will be possible. Otherwise a fixed lookup scheme must be used, and color toggling will **not** be possible.
If the colors will fit, **ncolors** is set to the required number of colors (computed as max-min+2) and 1 is returned. Otherwise **ncolors** is set to the number of hardware colors and 0 is returned.

change to hardware color
void
D_lookup_colors (CELL *data, int n, struct Colors *colors)

The **n** data values are changed to their corresponding hardware color number. The colors structure must be the same one that was passed to *D_set_colors.*

void
D_color (CELL cat, struct Colors *colors)

select raster color for line

D_color specifies a raster color to use for line drawing. See *R_color* for a related routine.

16.6 Raster Graphics

The display of raster graphics is very different from the display of vector graphics. While vector graphics routines can efficiently make use of world coordinates, the efficient rendering of raster images requires the programmer to work within the coordinate system of the graphics device. These routines make it easy to do just that. The application of these routines may be inspected in such commands as *d.rast*, *r.combine* and *r.weight* which display graphics results to the screen.

int
D_set_colors (struct Colors *colors)

establish raster colors for graphics

This routine sets the colors to be used for raster graphics. The **colors** structure must be either be read using *G_read_colors* or otherwise prepared using the routines described in *12.10.3 Raster Color Table (p. 116)*.
Return values are 1 if the colors will fit into the hardware color map; 0 otherwise (in which case a fixed color approximation based on these colors will be applied). These return codes are not error codes, just information.
Note. Due to the way this routine behaves, it is **not** correct to assume that a raster category value can be used to index the color registers. The routines *D_lookup_colors* or *D_color* must be used for that purpose.

int
D_cell_draw_setup (int top, int bottom, int left, int right)

prepare for raster graphic

The raster display subsystem establishes conversion parameters based on the screen extent defined by **top, bottom, left**, and **right**, all of which are obtainable from *D_get_screen_window for the current frame.*

int
D_draw_cell (int row, CELL *raster, struct Colors *colors)

render a raster row

The **row** gives the map array row. The **raster** array provides the categories for each raster value in that row. The **colors** structure must be the same as the one passed to *D_set_colors.*
This routine is called consecutively with the information necessary to draw a raster image from north to south. No rows can be skipped. All screen pixel rows which represent the current map array row are rendered. The routine returns the map array row which is needed to draw the next screen pixel row.

configure raster overlay mode

int
D_set_overlay_mode (int flag)

This routine determines if *D_draw_cell* draws in overlay mode (locations with category 0 are left untouched) or not (colored with the color for category 0). Set **flag** to 1 (TRUE) for overlay mode; 0 (FALSE) otherwise.

low level raster plotting

int
D_raster (CELL *raster, int n, int repeat, struct Colors *colors)

This low-level routine plots raster data. The **raster** array has **n** values. The raster is plotted **repeat** times, one row below the other. The **colors** structure must be the *same one passed to D_set_colors.*
Note. This routine does not perform resampling or placement. *D_draw_cell* does resampling and placement and then calls this routine to do the actual plotting.

Here is an example of how these routines are used to plot a raster map. The input parameters are the raster map name and mapset and an overlay flag.

```
#include <stdlib.h>

#include "gis.h"
#include "raster.h"
#include "display.h"

void plot_raster_map(char *name, char *mapset, int overlay)
{
        struct Colors colors;
        CELL *raster;
        int row, fd, top, bottom, left, right;

        /* perform plotting setup */
        D_setup(0);
        D_get_screen_window(&top, &bottom, &left, &right);

        if (D_cell_draw_setup(top, bottom, left, right))
                G_fatal_error("D_cell_draw_setup");
```

```
        raster = G_allocate_cell_buf();

        /* open raster map, read and set the colors */
        if((fd = G_open_cell_old(name, mapset)) < 0)
                G_fatal_error("G_open_cell_old");
        if (G_read_colors(name, mapset, &colors) < 0)
                G_fatal_error("G_read_colors");
        D_set_colors(&colors);

        /* plot */
        D_set_overlay_mode(overlay);
        for (row=0; row >= 0; )
        {
                if (G_get_map_row(fd, raster, row) < 0)
                        G_fatal_error("G_get_map_row");
                row = D_draw_cell(row, raster, &colors);
        }
        G_close_cell(fd);
        G_free_colors(&colors);
        G_free(raster);
}

int main(int argc, char **argv)
{
        char name[] = "elevation.dem";
        char *mapset;

        G_gisinit(argv[0]);

        mapset = G_find_cell2(name, "");

        if (R_open_driver() != 0)
                G_fatal_error("R_open_driver");

        plot_raster_map(name, mapset, 0);

        R_close_driver();

        return 0;
}
```

16.7 Window Clipping

This section describes a routine which is quite useful in many settings. Window clipping is used for graphics display and digitizing.

int
D_clip (double s, double n, double w, double e, double *x, double *y, double *c_x, double *c_y)

clip coordinates to window

A line represented by the coordinates **x, y** and **c_x, c_y** is clipped to the window defined by **s** (south), **n** (north), **w** (west), and **e** (east). Note that the following constraints must be true:
w <e
s <n
The **x** and **c_x** are values to be compared to **w** and **e.** The **y** and **c_y** are values to be compared to **s** and **n.**
The **x** and **c_x** values returned lie between **w** and **e.** The **y** and **c_y** values returned lie between **s** and **n.**

16.8 Pop-up Menus

pop-up menu int
D_popup (int bcolor, int tcolor, int dcolor, int top, int left, int size, char *options[])

This routine provides a pop-up type menu on the graphics screen. The **bcolor** specifies the background color. The **tcolor** is the text color. The **dcolor** specifies the color of the line used to divide the menu items. The **top** and **left** specify the placement of the top left corner of the menu on the screen. 0,0 is at the bottom left of the screen, and 100,100 is at the top right. The **size** of the text is given as a percentage of the vertical size of the screen. The **options** array is a NULL terminated array of character strings. The first is a menu title and the rest are the menu options (i.e., options[0] is the menu title, and options[1], options[2], etc., are the menu options). The last option must be the NULL pointer.
The coordinates of the bottom right of the menu are calculated based on the **top left** coordinates, the **size**, the number of **options**, and the longest option text length. If necessary, the menu coordinates are adjusted to make sure the menu is on the screen.
D_popup( ) does the following:

1. Current screen contents under the menu are saved.
2. Area is blanked with the background color and fringed with the text color.
3. Menu options are drawn using the current font.
4. User uses the mouse to choose the desired option.
5. Menu is erased and screen is restored with the original contents.
6. Number of the selected option is returned to the calling module.

16.9 Colors

set colors in driver int
D_reset_colors (struct Colors *colors)

Turns color information provided in the **colors** structure into color requests to the graphics driver. These colors are for raster graphics, not lines or text. See *12.10.3 Raster Color Table (p. 116)* for GIS Library routines which use this structure.

int
D_translate_color (char *name)

color name to number

Takes a color **name** in ascii and returns the color number for that color. Returns 0 if color is not known. The color number returned is for lines and text, not raster graphics.

16.10 Loading the Display Graphics Library

The library is loaded by specifying $(DISPLAYLIB), $(RASTERLIB) and $(GISLIB) in the Gmakefile. The following example is a complete Gmakefile which compiles code that uses this library:

Gmakefile for $(DISPLAYLIB)

```
OBJ = main.o sub1.o sub2.o

pgm: $(OBJ) $(DISPLAYLIB) $(RASTERLIB) $(GISLIB)
	$(CC) $(LDFLAGS) -o $@ $(OBJ) $(DISPLAYLIB) \
	$(RASTERLIB) $(GISLIB)

$(DISPLAYLIB): # in case the library changes

$(RASTERLIB): # in case the library changes

$(GISLIB): # in case the library changes
```

Note. This library uses routines in $(RASTERLIB). See *15 Raster Graphics Library (p. 243)* for details on that library. Also $(RASTERLIB) uses routines in $(GISLIB). See *12 GIS Library (p. 79)* for details on that library. See *11 Compiling and Installing GRASS Modules (p. 69)* for a complete discussion of Gmakefiles.

16.11 Vector Graphics / Plotting Routines

This section describes routines in GISLIB and the DISPLAYLIB libraries to support plotting of vector data. The best source for an example of how they are used is the GRASS *d.vect* module.

16.11.1 DISPLAYLIB routines

graphics frame setup int
D_setup (int clear)

Performs a full setup for the current graphics frame: 1) Makes sure there is a current graphics frame (will create a full-screen one, if not); 2) Sets the region coordinates so that the graphics frame and the active module region agree (may change active module region to do this); and 3) performs graphic frame/region coordinate conversion initialization.
If **clear** is true, the frame is cleared (same as running *d.erase.*) Otherwise, it is not cleared.

set clipping window int
D_set_clip_window (int top, int bottom, int left, int right)

Sets the clipping window to the pixel window that corresponds to the current database region. This is the default.

set clipping window to map window int
D_set_clip_window_to_map_window ( )

Sets the clipping window to the pixel window that corresponds to the current database region. This is the default.

line to x,y int
D_cont_abs (int x, int y)

Draws a line from the current position to pixel location **x,y.** Any part of the line that falls outside the clipping window is not drawn.
Note. The new position is **x,y**, even if it falls outside the clipping window. Returns 0 if the line was contained entirely in the clipping window, 1 if the line had to be clipped to draw it.

line to x,y int
D_cont_rel (int x, int y)

Equivalent to *D_cont_abs*(curx+x, cury+y) where **curx, cury** is the current pixel location.

int *move to pixel*
D_move_abs (int x, int y)

Move without drawing to pixel location **x,y**, even if it falls outside the clipping window.

int *move to pixel*
D_move_rel (int x, int y)

Equivalent to *D_move_abs*(curx+x, cury+y) where **curx, cury** is the current pixel location.

17 Lock Library

17.1 Introduction

This library provides an advisory locking mechanism. It is based on the idea that a process will write a process id into a file to create the lock, and subsequent processes will obey the lock if the file still exists and the process whose id is written in the file is still running.

17.2 Lock Routine Synopes

int *create a lock*
lock_file (char *file, int pid)

This routine decides if the lock can be set and, if so, sets the lock. If **file** does not exist, the lock is set by creating the file and writing the **pid** (process id) into the **file.** If **file** exists, the lock may still be active, or it may have been abandoned. To determine this, an integer is read out of the file. This integer is taken to be the process id for the process which created the lock. If this process is still running, the lock is still active and the lock request is denied. Otherwise the lock is considered to have been abandoned, and the lock is set by writing the **pid** into the **file.**
Return codes:
1 ok, lock request was successful
0 sorry, another process already has the file locked
-1 error. could not create the file
-2 error. could not read the file
-3 error. could not write the file

int *remove a lock*
unlock_file (char *file)

This routine releases the lock by unlinking **file.** This routine does NOT check to see that the process unlocking the file is the one which created the lock. The file is simply unlinked. Programs should of course unlock the lock if they created it. (Note, however, that the mechanism correctly handles abandoned locks.)

Return codes:
1 ok. lock file was removed
0 ok. lock file was never there
-1 error. lock file remained after attempt to remove it.

sectionUse and Limitations

It is worth noting that the process id used to lock the file does not have to be the process id of the process which actually creates the lock. It could be the process id of a parent process. The GRASS start-up shells, for example, invoke an auxiliary "locking" module that is told the file name and the process id to use. The start-up shells simply use a hidden file in the user's home directory as the lock file,[1] and their own process id as the locking pid, but let the auxiliary module actually do the locking (since the lock must be done by a program, not a shell script). The only consideration is that the parent process not exit and abandon the lock.

Warning. Locking based on process ids requires that all processes which access the lock file run on the same cpu. It will not work under a network environment since a process id alone (without some kind of host identifier) is not sufficient to identify a process.

17.3 Loading the Lock Library

The library is loaded by specifying $(LOCKLIB) in the Gmakefile. The following example is a complete Gmakefile which compiles code that uses this library:

Gmakefile for $(LOCKLIB)

```
OBJ = main.o sub1.o sub2.o

pgm: $(OBJ) $(LOCKLIB)

$(CC) $(LDFLAGS) -o $@ $(OBJ) $(LOCKLIB)

$(LOCKLIB): # in case the library changes
```

See *11 Compiling and Installing GRASS Modules (p. 69)* for a complete discussion of Gmakefiles.

[1] This file is .gislock (GRASS 4.x) and .gislock5 (GRASS 5.x).

18 Rowio Library

18.1 Introduction

Sometimes it is necessary to process large files which contain data in a matrix format and keep more than one row of the data in memory at a time. For example, suppose a module were required to look at five rows of data of input to produce one row of output (neighborhood function). It would be necessary to allocate five memory buffers, read five rows of data into them, and process the data in the five buffers. Then the next row of data would be read into the first buffer, overwriting the first row, and the five buffers would again be processed, etc. This memory management complicates the programming somewhat and is peripheral to the function being developed.

The *Rowio Library* routines handle this memory management. These routines need to know the number of rows of data that are to be held in memory and how many bytes are in each row. They must be given a file descriptor open for reading. In order to abstract the file i/o from the memory management, the programmer also supplies a subroutine which will be called to do the actual reading of the file. The library routines efficiently see to it that the rows requested by the module are in memory.

Also, if the row buffers are to be written back to the file, there is a mechanism for handling this management as well.

Note. All routines and global variables in this library, documented or undocumented, start with the prefix **ro wio_.** To avoid name conflicts, programmers should not create variables or routines in their own modules which use this prefix.

An alphabetic index is provided in *A.8 Appendix H: Index to Rowio Library (p. 453)*.

18.2 Rowio Routine Synopses

The routines in the *Rowio Library* are described below. They use a data structure called RO WIO which is defined in the header file ”rowio.h” that must be included in any code using these routines:[1]

```
# include ”rowio.h”
```

[1] The GRASS compilation process, described in *11 Compiling and Installing GRASS Modules (p. 69)*, automatically tells the C compiler how to find this and other GRASS header files.

configure rowio structure

int
rowio_setup (ROWIO *r, int fd, int nrows, int len, int (*getrow)( ), int (*putrow)( ))

Rowio_setup( ) initializes the ROWIO structure **r** and allocates the required memory buffers. The file descriptor **fd** must be open for reading. The number of rows to be held in memory is **nrows.** The length in bytes of each row is **len.** The routine which will be called to read data from the file is **getrow( )** and must be provided by the programmer. If the application requires that the rows be written back into the file if changed, the file descriptor **fd** must be open for write as well, and the programmer must provide a **putrow( )** routine to write the data into the file. If no writing of the file is to occur, specify NULL for **putrow( ).**
Return codes:
1 ok
-1 there is not enough memory for buffer allocation

The **getrow( )** routine will be called as follows:

```
getrow (fd, buf, n, len)
int fd;
char *buf;
int n, len;
```

When called, **getrow( )** should read data for row **n** from file descriptor **fd** into **buf** for **len** bytes. It should return 1 if the data is read ok, 0 if not.

The **putrow( )** routine will be called as follows:

```
putrow (fd, buf, n, len)
int fd;
char *buf;
int n, len;
```

When called, **putrow( )** should write data for row **n** to file descriptor **fd** from **buf** for **len** bytes. It should return 1 if the data is written ok, 0 if not.

read a row

char *
rowio_get (ROWIO *r, int n)

Rowio_get( ) returns a buffer which holds the data for row **n** from the file associated with ROWIO structure **r.** If the row requested is not in memory, the **getrow( )** routine specified in *rowio_setup* is called to read row **n** into memory and a pointer to the memory buffer containing the row is returned. If the data currently in the buffer had been changed by *rowio_put*, the **putrow( )** routine specified in *rowio_setup* is

called first to write the changed row to disk. If row **n** is already in memory, no disk read is done. The pointer to the data is simply returned.
Return codes:
NULL **n** is negative, or
getrow( ) returned 0 (indicating an error condition).
!NULL pointer to buffer containing row **n.**

int *forget a row*
rowio_forget (ROWIO *r, int n)

Rowio_forget( ) tells the routines that the next request for row **n** must be satisfied by reading the file, even if the row is in memory.
For example, this routine should be called if the buffer returned by *rowio_get* is later modified directly without also writing it to the file. See *18.3 Rowio Programming Considerations (p. 276).*

int *get file descriptor*
rowio_fileno (ROWIO *r)

Rowio_fileno( ) returns the file descriptor associated with the ROWIO structure.

int *free allocated memory*
rowio_release (ROWIO *r)

Rowio_release( ) frees all the memory allocated for ROWIO structure **r.** It does not close the file descriptor associated with the structure.

int *write a row*
rowio_put (ROWIO *r, char *buf, int n)

Rowio_put( ) writes the buffer **buf**, which holds the data for row **n**, into the ROWIO structure **r.** If the row requested is currently in memory, the buffer is simply copied into the structure and marked as having been changed. It will be written out later. Otherwise it is written immediately. Note that when the row is finally written to disk, the **putrow( )** routine specified in *rowio_setup* is called to write row **n** to the file. **rowio_flush** (r) force pending updates to disk ROWIO *r;
Rowio_flush( ) forces all rows modified by *rowio_put* to be written to the file. This routine must be called before closing the file or releasing the rowio structure if rowio_put( ) has been called.

18.3 Rowio Programming Considerations

If the contents of the row buffer returned by rowio_get( ) are modified, the programmer must either write the modified buffer back into the file or call rowio_forget( ). If this is not done, the data for the row will not be correct if requested again. The reason is that if the row is still in memory when it is requested a second time, the new data will be returned. If it is not in memory, the file will be read to get the row and the old data will be returned. If the modified row data is written back into the file, these routines will behave correctly and can be used to edit files. If it is not written back into the file, rowio_forget( ) must be called to force the row to be read from the file when it is next requested.

Rowio_get( ) returns NULL if getrow( ) returns 0 (indicating an error reading the file), or if the row requested is less than 0. The calling sequence for rowio_get( ) does not permit error codes to be returned. If error codes are needed, they can be recorded by getrow( ) in global variables for the rest of the module to check.

18.4 Loading the Rowio Library

The library is loaded by specifying $(ROWIOLIB) in the Gmakefile. The following example is a complete Gmakefile which compiles code that uses this library:

Gmakefile for $(ROWIOLIB)

```
OBJ = main.o sub1.o sub2.o

pgm: $(OBJ) $(ROWIOLIB)

$(CC) $(LDFLAGS) -o $@ $(OBJ) $(ROWIOLIB)

$(ROWIOLIB): # in case the library changes
```

See *11 Compiling and Installing GRASS Modules (p. 69)* for a complete discussion of Gmakefiles.

19 Segment Library

19.1 Introduction

Large data files which contain data in a matrix format often need to be accessed in a nonsequential or random manner. This requirement complicates the programming.

Methods for accessing the data are to:

(1) read the entire data file into memory and process the data as a two-dimensional matrix,

(2) perform direct access i/o to the data file for every data value to be accessed, or

(3) read only portions of the data file into memory as needed.

Method (1) greatly simplifies the programming effort since i/o is done once and data access is simple array referencing. However, it has the disadvantage that large amounts of memory may be required to hold the data. The memory may not be available, or if it is, system paging of the module may severely degrade performance. Method (2) is not much more complicated to code and requires no significant amount of memory to hold the data. But the i/o involved will certainly degrade performance. Method (3) is a mixture of (1) and (2). Memory requirements are fixed and data is read from the data file only when not already in memory. Howev er the programming is more complex.

The routines provided in this library are an implementation of method (3). They are based on the idea that if the original matrix were segmented or partitioned into smaller matrices these segments could be managed to reduce both the memory required and the i/o. Data access along connected paths through the matrix, (i.e., moving up or down one row and left or right one column) should benefit.

In most applications, the original data is not in the segmented format. The data must be transformed from the nonsegmented format to the segmented format. This means reading the original data matrix row by row and writing each row to a new file with the segmentation organization. This step corresponds to the i/o step of method (1).

Then data can be retrieved from the segment file through routines by specifying the row and column of the original matrix. Behind the scenes, the data is paged into memory as needed and the requested data is returned to the caller.

Note. All routines and global variables in this library, documented or undocumented, start with the prefix **segment_.** To avoid name conflicts, programmers should not create variables or routines in their own modules which use this prefix.

An alphabetic index is provided in A.9 Appendix I: Index to Segment Library (p. 454).

19.2 Segment Routines

The routines in the *Segment Library* are described below, more or less in the order they would logically be used in a module. They use a data structure called SEGMENT which is defined in the header file "segment.h" that must be included in any code using these routines:[1]

include "segment.h"

The first step is to create a file which is properly formatted for use by the *Segment Library* routines:

format a segment file

int
segment_format (int fd, int nrows, int ncols, int srows, int scols, int len)

The segmentation routines require a disk file to be used for paging segments in and out of memory. This routine formats the file open for write on file descriptor **fd** for use as a segment file. A segment file must be formatted before it can be processed by other segment routines. The configuration parameters **nrows, ncols, srows, scols**, and **len** are written to the beginning of the segment file which is then filled with zeros.
The corresponding nonsegmented data matrix, which is to be transferred to the segment file, is **nrows** by **ncols.** The segment file is to be formed of segments which are **srows** by **scols.** The data items have length **len** bytes. For example, if the *data type is int*, ***len*** *is sizeof(int).*
Return codes are: 1 ok; else -1 could not seek or write *fd*, or -3 illegal configuration parameter(s).

The next step is to initialize a SEGMENT structure to be associated with a segment file formatted by *segment_format.*

initialize segment structure

int
segment_init (SEGMENT *seg, int fd, int nsegs)

Initializes the **seg** structure. The file on **fd** is a segment file created by *segment_format* and must be open for reading and writing. The segment file configuration parameters *nrows, ncols, srows, scols*, and *len*, as written to the file by

[1] The GRASS compilation process, described in 11 Compiling and Installing GRASS Modules (p. 69), automatically tells the C compiler how to find this and other GRASS header files.

segment_format, are read from the file and stored in the **seg** structure. **Nsegs** specifies the number of segments that will be retained in memory. The minimum value allowed is 1.
Note. The size of a segment is *scols*srows*len* plus a few bytes for managing each segment.
Return codes are: 1 if ok; else -1 could not seek or read segment file, or -2 out of memory.

Then data can be written from another file to the segment file row by row:

int *write row to*
segment_put_row (SEGMENT *seg, char *buf, int row) *segment file*

Transfers nonsegmented matrix data, row by row, into a segment file. **Seg** is the segment structure that was configured from a call to *segment_init.* **Buf** should contain *ncols*len* bytes of data to be transferred to the segment file. **Row** specifies the row from the data matrix being transferred.
Return codes are: 1 if ok; else -1 could not seek or write segment file.

Then data can be read or written to the segment file randomly:

int *get value from*
segment_get (SEGMENT *seg, char *value, int row, int col) *segment file*

Provides random read access to the segmented data. It gets *len* bytes of data into **value** from the segment file **seg** for the corresponding **row** and **col** in the original data matrix.
Return codes are: 1 if ok; else -1 could not seek or read segment file.

int *put value to*
segment_put (SEGMENT *seg, char *value, int row, int col) *segment file*

Provides random write access to the segmented data. It copies *len* bytes of data from **value** into the segment structure **seg** for the corresponding **row** and **col** in the original data matrix.
The data is not written to disk immediately. It is stored in a memory segment until the segment routines decide to page the segment to disk.
Return codes are: 1 if ok; else -1 could not seek or write segment file.

After random reading and writing is finished, the pending updates must be flushed to disk:

flush pending updates to disk

int
segment_flush (SEGMENT *seg)

Forces all pending updates generated by *segment_put* to be written to the segment file **seg.** Must be called after the final segment_put( ) to force all pending updates to disk. Must also be called before the first call to *segment_get_row.*

Now the data in segment file can be read row by row and transferred to a normal sequential data file:

read row from segment file

int
segment_get_row (SEGMENT *seg, char *buf, int row)

Transfers data from a segment file, row by row, into memory (which can then be written to a regular matrix file). **Seg** is the segment structure that was configured from a call to *segment_init.* **Buf** will be filled with *ncols*len* bytes of data corresponding to the **row** in the data matrix.
Return codes are: 1 if ok; else -1 could not seek or read segment file.

Finally, memory allocated in the SEGMENT structure is freed:

free allocated memory

int
segment_release (SEGMENT *seg)

Releases the allocated memory associated with the segment file **seg.** Does not close the file. Does not flush the data which may be pending from previous *segment_put* calls.

19.3 How to Use the Library Routines

The following should provide the programmer with a good idea of how to use the *Segment Library* routines. The examples assume that the data is integer. The first step is the creation and formatting of a segment file. A file is created, formatted and then closed:

```
fd = creat (file,0666);
segment_format (fd, nrows, ncols, srows, scols, sizeof(int));
close(fd);
```

The next step is the conversion of the nonsegmented matrix data into segment file format. The segment file is reopened for read and write, initialized, and then data read row by row from the original data file and put into the segment file:

```
int buf[NCOLS];
SEGMENT seg;
fd = open (file, 2); segment_init (&seg, fd, nseg)
for (row = 0; row < nrows; row++)
{
<code to get original matrix data for row into buf>
segment_put_row (&seg, buf, row);
}
```

Of course if the intention is only to add new values rather than update existing values, the step which transfers data from the original matrix to the segment file, using segment_put_row( ), could be omitted, since *segment_format* will fill the segment file with zeros.

The data can now be accessed directly using *segment_get.* For example, to get the value at a given row and column:

```
int value;
SEGMENT seg;
segment_get (&seg, &value, row, col);
```

Similarly *segment_put* can be used to change data values in the segment file:

```
int value;
SEGMENT seg;
value = 10;
segment_put (&seg, &value, row, col);
```

Warning. It is an easy mistake to pass a value directly to segment_put( ). The following should be avoided:

```
segment_put (&seg, 10, row, col); /* this will not work */
```

Once the random access processing is complete, the data would be extracted from the segment file and written to a nonsegmented matrix data file as follows:

```
segment_flush (&seg);
for (row = 0; row < nrows; row++)
{
segment_get_row (&seg, buf, row);
<code to put buf into a matrix data file for row>
}
```

Finally, the memory allocated for use by the segment routines would be released and the file closed:

```
segment_release (&seg);
close (fd);
```

Note. The *Segment Library* does not know the name of the segment file. It does not attempt to remove the file. If the file is only temporary, the programmer should remove the file after closing it.

19.4 Loading the Segment Library

The library is loaded by specifying $(SEGMENTLIB) in the Gmakefile. The following example is a complete Gmakefile which compiles code that uses this library:

Gmakefile for $(SEGMENTLIB)

OBJ = main.o sub1.o sub2.o

pgm: $(OBJ) $(SEGMENTLIB)

$(CC) $(LDFLAGS) -o $@ $(OBJ) $(SEGMENTLIB)

$(SEGMENTLIB): # in case the library changes

See *11 Compiling and Installing GRASS Modules (p. 69)* for a complete discussion of Gmakefiles.

20 Vask Library

20.1 Introduction

The *Vask Library* (visual-ask) provides an easy means to communicate with a user one page at a time. That is, a page of text can be provided to the user with information and question prompts. The user is allowed to move the cursor[1] from prompt to prompt answering questions in any desired order. Users' answers are confined to the programmer-specified screen locations.

This interface is used in many interactive GRASS modules.[2] For the user, the *Vask Library* provides a very consistent and simple interface. It is also fairly simple and easy for the programmer to use.

Note. All routines and global variables in this library, documented or undocumented, start with the prefix **V_.** To avoid name conflicts, programmers should not create variables or routines in their own modules which use this prefix.

An alphabetic index is provided in *A.10 Appendix J: Index to Vask Library (p. 454)*.

20.2 Vask Routine Synopses

The routines in the *Vask Library* are described below, more or less in the order they would logically be used in a module. The *Vask Library* maintains a private data space for recording the screen description. With the exception of V_call( ), which does all the screen painting and user interaction, *vask* routines only modify the screen description and do not update the screen itself.

int
V_clear () — *initialize screen description*

This routine initializes the screen description information, and must be called before each new screen layout description.

[1] The functions in this library make use of the curses library and termcap descriptios. As when using vi, the user must have the TERM variable set.

[2] The GRASS *g.region* command is a good example, as are *r.reclass* and *r.mask*.

add line of text to screen int
V_line (int num, char *text)

This routine is used to place lines of text on the screen. **Row** is an integer value of 0-22 specifying the row on the screen where the **text** is placed. The top row on the screen is row 0.
Warning. V_line( ) does not copy the text to the screen description. It only saves the text address. This implies that each call to V_line( ) must use a different text buffer.

define screen constant int
V_const (*Ctype* *value, char type, int row, int col, int len)

Ctype is one of int, long, float, double, or char.

define screen question int
V_ques (*Ctype* *value, char type, int row, int col, int len)

Ctype is one of int, long, float, double, or char.
These two calls use the same syntax. V_const( ) and V_ques( ) specify that the contents of memory at the address of **value** are to be displayed on the screen at location **row, col** for **len** characters. V_ques( ) further specifies that this screen location is a prompt field. The user will be allowed to change the field on the screen and thus change the **value** itself. V_const( ) does not define a prompt field, and thus the user will not be able to change these values.
Value is a pointer to an int, long, float, double, or char string. **Type** specifies what type value points to: 'i' (int), 'l' (long), 'f' (float), 'd' (double), or 's' (character string). **Row** is an integer value of 0-22 specifying the row on the screen where the value is placed. The top row on the screen is row 0. **Col** is an integer value of 0-79 specifying the column on the screen where the value is placed. The leftmost column on the screen is column 0. **Len** specifies the number of columns that the value will use.
Note that the size of a character array passed to V_ques( ) must be at least one byte longer than the length of the prompt field to allow for NULL termination. Currently, you are limited to 20 constants and 80 variables.
Warning. These routines store the address of **value** and not the value itself. This implies that different variables must be used for different calls. Programmers will instinctively use different variables with V_ques( ), but it is a stumbling block for V_const( ). Also, the programmer must initialize **value** prior to calling these routines.[3]

[3] Technically **value** needs to be initialized before the call to V_call( ) since V_const( ) and V_ques( ) only store the address of **value.** V_call( ) looks up the values and places them on the screen.

int
V_float_accuracy (int num)

set number of decimal places

V_float_accuracy( ) defines the number of decimal places in which floats and doubles are displayed or accepted. **Num** is an integer value defining the number of decimal places to be used. This routine affects subsequent calls to V_const( ) and V_ques( ). Various inputs or displayed constants can be represented with different numbers of decimal places within the same screen display by making different calls to V_float_accuracy( ) before calls to V_ques( ) or V_const( ). V_clear( ) resets the number of decimal places to the default (which is unlimited).

int
V_call ()

interact with the user

V_call( ) clears the screen and writes the text and data values specified by V_line( ), V_ques( ) and V_const( ) to the screen. It interfaces with the user, collecting user responses in the V_ques( ) fields until the user is satisfied. A message is automatically supplied on line number 23, explaining to the user to enter an ESC when all inputs have been supplied as desired. V_call( ) ends when the user hits ESC and returns a value of 1 (but see V_intrpt_ok( ) below). No error checking is done by V_call( ). Instead, all variables used in V_ques( ) calls must be checked upon return from V_call( ). If the user has supplied inappropriate information, the user can be informed, and the input prompted for again by further calls to V_call( ).

int
V_intrpt_ok ()

allow ctrl-c

V_call( ) normally only allows the ESC character to end the interactive input session. Sometimes it is desirable to allow the user to cancel the session. To provide this alternate means of exit, the programmer can call V_intrpt_ok( ) before V_call( ). This allows the user to enter Ctrl-C, which causes V_call( ) to return a value of 0 instead of 1.

A message is automatically supplied to the user on line 23 saying to use Ctrl-C to cancel the input session. The normal message accompanying V_call( ) is moved up to line 22.

Note. When V_intrpt_ok( ) is called, the programmer must limit the use of V_line( ), V_ques( ), and V_const( ) to lines 0-21.

int
V_intrpt_msg (char *text)

change ctrl-c message

A call to V_intrpt_msg( ) changes the default V_intrpt_ok( ) message from (OR <Ctrl-C> TO CANCEL) to (OR <Ctrl-C> TO *msg*). The message is (re)set to the default by V_clear( ).

20.3 An Example Program

Following is the code for a simple module which will prompt the user to enter an integer, a floating point number, and a character string.

```
# define LEN 15

main( )

{

int i ; /* the variables */

float f ;

char s[LEN] ;

i=0; /*initialize the variables */

f = 0.0 ;

*s = 0 ;

V_clear( ) ; /* clear vask info */

V_line( 5, " Enter an Integer ") ; /* the text */

V_line( 7, " Enter a Decimal ") ;

V_line( 9, " Enter a character string ") ;

V_ques ( &i, 'i', 5, 30, 5) ; /* the prompt fields */

V_ques ( &f, 'f', 7, 30, 5) ;

V_ques ( s, 's', 9, 30, LEN - 1) ;

V_intrpt_ok( ); /* allow ctrl-c */

if (!V_call( )) /* display and get user input */

exit(1); /* exit if ctrl-c */

fprintf(stdout, "%d %f %s\n", i, f, s) ; /* ESC, so print results */
```

exit(0);

}

The user is presented with the following screen:

Enter an Integer 0 ____

Enter a Decimal 0.00 _

Enter a character string _____

AFTER COMPLETING ALL ANSWERS, HIT <ESC><ENTER> TO CONTINUE (OR <Ctrl-C> TO CANCEL)

The user has several options.

<CR> moves the cursor to the next prompt field.

CTRL-K moves the cursor to the previous prompt field.

CTRL-H moves the cursor backward nondestructively within the field.

CTRL-L moves the cursor forward nondestructively within the field. CTRL-A writes a copy of the screen to a file named *visual_ask* in the user's home directory.

ESC returns control to the calling module with a return value of 1.

CTRL-C returns control to the calling module with a return value of 0. Displayable ascii characters typed by the user are accepted and displayed. Control characters (other than those with special meaning listed above) are ignored.

20.4 Loading the Vask Library

Compilations must specify the vask, curses, and termcap libraries. The library is loaded by specifying $(VASK) and $(VASKLIB) in the Gmakefile. The following example is a complete Gmakefile which compiles code that uses this library:

Gmakefile for $(VASK)

OBJ = main.o sub1.o sub2.o

pgm: $(OBJ) $(VASKLIB)

$(CC) $(LDFLAGS) -o $@ $(OBJ) $(VASK)

$(VASKLIB): # in case the library changes

Note. The target *pgm* depends on the object files $(OBJ) and the *Vask Library* $(VASKLIB). This is done so that modifications to any of the $(OBJ) files or to the $(VASKLIB) itself will force module reloading. Linking against the Vask library is performed using $(VASKLIB) $(CURSES), it specifies both the UNIX curses and termcap/terminfo libraries as well as $(VASKLIB).

See *11 Compiling and Installing GRASS Modules (p. 69)* for a complete discussion of Gmakefiles.

20.5 Programming Considerations

The order of movement from prompt field to prompt field is dependent on the ordering of calls to V_ques( ), not on the line numbers used within each call. Information cannot be entered beyond the edges of the prompt fields. Thus, the user response is limited by the number of spaces in the prompt field provided in the call to V_ques( ). Some interpretation of input occurs during the interactive information gathering session. When the user enters <CR> to move to the next prompt field, the contents of the current field are read and rewritten according to the value type associated with the field. For example, nonnumeric responses (e.g., ”abc”) in an integer field will get turned to a 0, and floating point numbers will be truncated (e.g., 54.87 will become 54).

No error checking (other than matching input with variable type for that input field) is done by V_call( ). This must be done, by the programmer, upon return from V_call( ). Calls to V_line( ), V_ques( ), and V_const( ) store only pointers, not contents of memory. At the time of the call to V_call( ), the contents of memory at these addresses are copied into the appropriate places of the screen description. Care should be taken to use distinct pointers for different fields and lines of text. For example, the following mistake should be avoided:

```
char

text[100];

V_clear( );

sprintf(text,” Welcome to GRASS ”);

V_line(3,text);

sprintf(text,” which is a product of the US Army CERL ”);

V_line(5,text);

V_call( );
```

since this results in the following (unintended) screen:

which is a product of the GRASS Development Team

which is a product of the GRASS Development Team

AFTER COMPLETING ALL ANSWERS, HIT <ESC><ENTER> TO CONTINUE

(OR <Ctrl-C> TO CANCEL)

Warning. Due to a problem in a routine within the curses library,[4] the Vask routines use the curses library in a somewhat unorthodox way. This avoided the problem within curses, but means that the programmer cannot mix the use of the *Vask Library* with direct calls to curses routines. Any module using the *Vask Library* should not call curses **library routines directly.**

[4] Specifically, memory allocated by initscr( ) was not freed by endwin( ).

21 Projection and Datum support

21.1 Supported projections

GRASS Projection software is based on PROJ4 (developed until 1995 by USGS)

(http://www.remotesensing.org/proj/).

For cartographic GRASS functions see also *12.7.3 Projection Information (p. 96)*.

```
ll -- Lat/Lon
utm -- Universe Transverse Mercator
stp -- State Plane
aea -- Albers Equal Area
lcc -- Lambert Conformal Conic
merc -- Mercator
tmerc -- Transverse Mercator
leac -- Lambert Equal Area Conic
laea -- Lambert Azimuthal Equal Area
aeqd -- Azimuthal Equidistant
airy -- Airy
aitoff -- Aitoff
alsk -- Mod. Stererographics of Alaska
apian -- Apian Globular I
august -- August Epicycloidal
bacon -- Bacon Globular
bipc -- Bipolar conic of western hemisphere
boggs -- Boggs Eumorphic
bonne -- Bonne (Werner lat_1=90)
cass -- Cassini
cc -- Central Cylindrical
cea -- Equal Area Cylindrical
chamb -- Chamberlin Trimetric
collg -- Collignon
crast -- Craster Parabolic (Putnins P4)
denoy -- Denoyer Semi-Elliptical
eck1 -- Eckert I
eck2 -- Eckert II
eck3 -- Eckert III
eck4 -- Eckert IV
eck5 -- Eckert V
eck6 -- Eckert VI
eqc -- Equidistant Cylindrical (Plate Caree)
eqdc -- Equidistant Conic
euler -- Euler
fahey -- Fahey
fouc -- Foucaut
```

```
fouc_s -- Foucaut Sinusoidal
gall -- Gall (Gall Stereographic)
gins8 -- Ginsburg VIII (TsNIIGAiK)
gn_sinu -- General Sinusoidal Series
gnom -- Gnomonic
goode -- Goode Homolosine
gs48 -- Mod. Stererographics of 48 U.S.
gs50 -- Mod. Stererographics of 50 U.S.
hammer -- Hammer & Eckert-Greifendorff
hatano -- Hatano Asymmetrical Equal Area
imw_p -- International Map of the World Polyconic
kav5 -- Kavraisky V
kav7 -- Kavraisky VII
labrd -- Laborde
lagrng -- Lagrange
larr -- Larrivee
lask -- Laskowski
lee_os -- Lee Oblated Stereographic
loxim -- Loximuthal
lsat -- Space oblique for LANDSAT
mbt_s -- McBryde-Thomas Flat-Polar Sine (No. 1)
mbt_fps -- McBryde-Thomas Flat-Pole Sine (No. 2)
mbtfpp -- McBride-Thomas Flat-Polar Parabolic
mbtfpq -- McBryde-Thomas Flat-Polar Quartic
mbtfps -- McBryde-Thomas Flat-Polar Sinusoidal
mil_os -- Miller Oblated Stereographic
mill -- Miller Cylindrical
mpoly -- Modified Polyconic
moll -- Mollweide
murd1 -- Murdoch I
murd2 -- Murdoch II
murd3 -- Murdoch III
nell -- Nell
nell_h -- Nell-Hammer
nicol -- Nicolosi Globular
nsper -- Near-sided perspective
nzmg -- New Zealand Map Grid
ob_tran -- General Oblique Transformation
ocea -- Oblique Cylindrical Equal Area
oea -- Oblated Equal Area
omerc -- Oblique Mercator
ortel -- Ortelius Oval
ortho -- Orthographic
pconic -- Perspective Conic
poly -- Polyconic (American)
putp1 -- Putnins P1
putp2 -- Putnins P2
putp3 -- Putnins P3
putp3p -- Putnins P3'
putp4p -- Putnins P4'
putp5 -- Putnins P5
putp5p -- Putnins P5'
putp6 -- Putnins P6
putp6p -- Putnins P6'
qua_aut -- Quartic Authalic
robin -- Robinson
rpoly -- Rectangular Polyconic
sinu -- Sinusoidal (Sanson-Flamsteed)
somerc -- Swiss. Obl. Mercator
```

```
stere -- Stereographic
tcc -- Transverse Central Cylindrical
tcea -- Transverse Cylindrical Equal Area
tissot -- Tissot
tpeqd -- Two Point Equidistant
tpers -- Tilted perspective
ups -- Universal Polar Stereographic
urm5 -- Urmaev V
urmfps -- Urmaev Flat-Polar Sinusoidal
vandg -- van der Grinten (I)
vandg2 -- van der Grinten II
vandg3 -- van der Grinten III
vandg4 -- van der Grinten IV
vitk1 -- Vitkovsky I
wag1 -- Wagner I (Kavraisky VI)
wag2 -- Wagner II
wag3 -- Wagner III
wag4 -- Wagner IV
wag5 -- Wagner V
wag6 -- Wagner VI
wag7 -- Wagner VII
weren -- Werenskiold I
wink1 -- Winkel I
wink2 -- Winkel II
wintri -- Winkel Tripel
```

21.2 GRASS and the PROJ4 projection library

GRASS utilizes the PROJ4 library developed by Gerald Evenden/USGS (Cartographic Projection Procedures for the UNIX Environment – A User's Manual, (Evenden, 1990, Open-file report 90-284).). However, the PROJ4 functions are commonly used through wrapper functions in a GRASS environment.

Internally to the PROJ.4 library, projection may involve transformation to and from geodetic coordinates (latitude and longitude), and numerical corrections to account for different datums.

This is transparent for the user as input and output parameters are either read from PROJ_INFO & PROJ_UNITS files (v.proj, r.proj, s.proj) or read from user prompts (m.proj2).

In GRASS the wrapper functions in src/libes/proj/get_proj.c makes the preparations to set up the parameter strings and init the info-structures, while do_proj.c contains the actual projection calls.

21.2.1 Include Files

All modules using the PROJ4 library should include the header file **projects.h** like this:

```
#include "projects.h"
```

In the Gmakefile a reference to $(GPROJLIB) is required.

21.2.2 Initialization

Initialize pj_info

int
pj_zero_proj (struct pj_info *info)

This function is deprecated and it may be removed in the future.
Initialization of PROJ info structure to "no data". Use of this function is not necessary since its contents are duplicated inside both pj_get_kv and pj_get_string, one or other of which must be called to set up the projection parameters.

Get projection key values

int
pj_get_kv(struct pj_info *info, struct Key_Value *in_proj_keys, struct Key_Value *in_units_keys)

Get projection key values from current location settings (PERMANENT/PROJ_INFO and PERMANENT/PROJ_UNITS files).

Read in projection settings

int
pj_get_string(struct pj_info *info, char *str)

Reads in projection settings.

21.2.3 Projection of coordinate pairs

Project x,y

int
pj_do_proj(double *x, double *y, struct pj_info *info_in, struct pj_info *info_out)

Project the given coordinate pair (x, y) from a projection defined in info_in to projection defined in info_out.

Project x,y,h

int
pj_do_transform(int count, double *x, double *y, double *h, struct pj_info *info_in, struct pj_info *info_out)

Project the given coordinate triple (x, y, h) from a projection defined in info_in to projection defined in info_out. x, y and h should be three arrays of equal length; count is the number of points to be transformed.

Both pj_do_proj and pj_do_transform will perform datum transformation if one of the datum identifiers (datum, (dx, dy, and dz), towgs84, or nadgrids) is supplied for both the input and output projections.

21.2.4 Programming Example

Below is outlined a draft PROJ4 programming example using the GRASS-PROJ4 environment. It reads the current location projection information and converts a given coordinate pair matching this projection to lat/long.

```
#include <math.h>
#include <string.h>
#include <stdio.h>
#include "gis.h"
#include "projects.h"

int main (int argc, char **argv)
{
    struct Key_Value *in_proj_info, *in_unit_info; /* input projection */
    struct pj_info iproj;       /* input map proj parameters  */
    struct pj_info oproj;       /* output map proj parameters  */
    UV data;
    double longitude, latitude; /* coordinate pair to be transformed */


    G_gisinit(argv[0]);

    /* preset coordinates, must match location projection settings */
    longitude=3578000.0; /* Transverse Mercator, Gauss-Krueger */
    latitude=5770000.0;

    fprintf(stderr, "IN: longitude: %f, latitude: %f\n", longitude, latitude);


    /* if location/coordinate pair are not in lat/long format, transform them: */
    if ((G_projection() != PROJECTION_LL))
    {
     fprintf(stderr, "Transforming input coordinates to lat/long\n");

     /* read current projection info */
     if ((in_proj_info = G_get_projinfo()) == NULL)
        G_fatal_error("Can't get projection info of current location");

     if ((in_unit_info = G_get_projunits()) == NULL)
        G_fatal_error("Can't get projection units of current location");

     if (pj_get_kv(&iproj, in_proj_info, in_unit_info) < 0)
        G_fatal_error("Can't get projection key values of current location");
```

```
    /* set output projection to lat/long*/
    pj_get_string(&oproj, (char *)NULL);


    /* Now do the transform:
     * order:       in/outx    in/outy  in_info  out_info */
    if(pj_do_proj(&longitude, &latitude, &iproj, &oproj) < 0)
    {
      fprintf(stderr,"Error in pj_do_proj\n");
      exit(0);
    }
  }

  fprintf(stderr, "OUT: longitude: %f, latitude: %f\n", longitude, latitude);
}
```

21.3 Coordinate Conversion Library (coorcnv)

Most of the information presented here was derived from the source code of the coordinate conversion library and from the manuals and source code of the modules using this library. There are no hints on the authors name in the code. The modules have been written by Michael Shapiro of US Army CERL (the library too?).

21.3.1 Introduction to the Coordinate Conversion Library

The *Coordinate Conversion Library* provides functions for latitude-longitude calculations, e. g. for projecting latitude-longitude to universal transverse mercator (UTM) and transverse mercator (tm), for inverse projection to latitude-longitude, for calculating datum shifts on latitude-longitude coordinates, for scanning coordinates from strings and converting of latitude-longitude coordinates to geocentric coordinates and vice versa.

Some of this functionality is already included in the *12 GIS Library (p. 79)* and in the PROJ.4 projection library. Some functions are wrappers for appropriate functions from the GIS Library, this was introduced to maintain compatibility with old programming code while introducing support for centralized tables for ellipsoid and map datum parameters.

The library is not fully tested, so use this with care.

Parts of the library are in parallel to functions from the GIS Library and the PROJ.4 library in GRASS. No checks have been done so far to validate that the functions give numerically identical results.

21.3.1.1 Include Files

All modules using the coorcnv library should include the header file **CC.h** like this:

```
#include "CC.h"
```

The modules using this library must always be linked with the math library and the GIS Library, as the library itself needs internally functions from the GIS Library.

21.3.2 Future plans for enhanced map datum support

Update (January 2003): Datum transformation is handled automatically by the PROJ.4 library. Already datum, dx, dy and dz are valid keywords in the PROJ_INFO file. This will be extended to allow towgs84 (for 7-parameter transformation *or* 3-parameter—alternative notation to dx dy dz) and nadgrids (to use tables for accurate localised shifting between NAD27 and NAD83 datums).

The GRASS datum.table format will be extended to include the towgs84 and nadgrids parameters in addition to dx dy and dz, where available or contributed by users, and in the end only underlying parameters, not ellipsoid or datum names will be passed to PROJ.4. This will allow for historical differences in naming conventions. None of the GRASS ellipsoid or datum names may ever be changed as this would break existing installations.

```
%% remove if not applicable or make formatted text
This are the neccessary steps to get
full map datum support within GRASS:

- change coorcnv library calls to use \$GISBASE/etc/datum.table
done.

- change coorcnv library to use ellipsoids in
\$GISBASE/etc/ellipse.table
done.

- add datum support functions to coorcnv library:
datum shift parameters:
+ CC\_get\_datum\_by\_name
+ CC\_get\_datum\_by\_nbr
x CC\_datum\_name
x CC\_datum\_description
+ CC\_datum\_ellipsoid
x CC\_datum\_shift -> CC\_get\_datum\_parameters
+ CC\_get\_datum\_parameters ( or CC\_get\_datum\_parameters3?)
  CC\_get\_datum\_parameters7

spheroid/ellipsoid parameters:
(only wrappers for libgis calls!)
* CC\_get\_spheroid
* CC\_spheroid\_name
+ CC\_get\_spheroid\_by\_name
+ CC\_get\_spheroid\_by\_nbr
(reworked an new gislib calls)
+ G\_ellipsoid\_name
+ G\_get\_spheroid\_by\_name (for f parameter)
+ G\_ellipsoid\_name
+ G\_ellipsoid\_description
```

```
* G\_get\_ellipsoid\_by\_name
* G\_get\_ellipsoid\_parameters

datum shift routines:
+ CC\_datum\_shift\_Molodensky
+ CC\_datum\_to\_datum\_shift\_M
+ CC\_datum\_shift\_CC
+ CC\_datum\_to\_datum\_shift\_CC
+ CC\_datum\_shift\_BursaWolf
+ CC\_datum\_to\_datum\_shift\_BW

others:
+ CC\_geo2lld
+ CC\_lld2geo

symbols:
? not known
x changable, because not used in any module
* existing, not changable
+ new
  not yet implemented
mostly done, needs testing.

- change m.datum.shift to new library calls
done, needs testing.

- add functions for low-level datum support to gislib
(in analogy to the ellipsoid and projection functions):
add metadata (datum name and parameters) to PROJ\_INFO
to every PERMAMENT mapset of locations.
This is done via g.setproj i assume (please correct me
if i am wrong!)

proposed format:
datum:  acronym
ellips: acronym -> already there
dx: -> shifting parameter dx relative to wgs84
dy: -> dy
dz: -> dz
a:  -> already in PROJ\_INFO
e2: -> already in PROJ\_INFO
f:  -> from datum.table
e\_desc: ellipsoid description
d\_desc: datum description

- add a function to read the datum parameters for the
actual location (parallel to G\_database\_projection\_name
and G\_get\_ellipsoid\_parameters)
+ G\_database\_datum\_name()
+ G\_get\_datum\_parameters(a, e2, f, dx, dy, dz)
+ G\_get\_datum\_parameters7(a, e2, f, dx, dy, dz, rx, ry, rz, m)

- add G\_ask\_datum\_name for interactive use in modules
+ G\_ask\_datum\_name()
done.

- change all modules that create new locations to
ask the user for a map datum and add to PROJ\_INFO file.
This is done via g.setproj?
```

```
- change all modules that do projection or data-import
to use datum conversion:
v.proj
r.proj
s.proj
r.in.*
v.in.*
etc.

Please note that i am only involved with the first steps of
changing the library to support map datums. The changes in
the modules must be done by other GRASS developers or
the module maintainers.

Andreas Lange, 05/2000,
andreas.lange@rhein-main.de
```

21.3.2.1 The map datum database

All map datums known within GRASS must be listed in the map datum database in the file $GISBASE/etc/datum.table. The ellipsoid-acronym is a reference to the ellipsoid database in the file $GISBASE/etc/ellipse.table.

The format of the map datum file is as follows:

```
acronym "description" ellipsoid-acronym dx= dy= dz=
```

The acronym is a short name (single word) datum specifier, the description gives a long name and reference to the map datum (enclosed in double quotes).

The ellipsoid-acronym is the short name of the ellipsoid used with this map datum and referenced in the ellipse.table.

dx, dy and dz are the datum shift parameters, which are applied to convert from local map datum to wgs84 datum. The reverse calculation can be done with signs reversed.

Comments are signalled by a '#' at first position of the line, empty lines are discarded.

A sample entry:

```
  # World Geodetic System 1984 wgs84 "World Geodetic System 1984"
wgs84 dx=0.0 dy=0.0 dz=0.0
```

If you need additional map datums add them to the file. Please comment the new entry and cite a reference for the values.

You can not use the parameters for the Molodensky datum shift formula with any other datum shift formula. The Bursa-Wolf datum transformation needs 7 parameters (3 xyz-shift, 3 xyz-rotational, 1 scale factor), which can not be used with any other formula. Specifically do not

use the 3 xyz-shift parameters for the Bursa-Wolf transformation with the Molodensky formula, as the parameters are not independent from another.

21.3.2.2 The ellipsoid database/table

21.3.3 Datum-shift related functions

Functions which provide datum shift values from the database and calculate the datum shift with these parameters.

21.3.3.1 Reading datum parameters from database

?? ?? *(p.* **??***)*

The following functions are provided to read information from the central map datum table (datum.table), see also the function CC_get_spheroid in section *12.8.6 Miscellaneous (p. 104)* for reading from the central ellipsoid table (ellipse.table). The transformation parameters in datum.table are meant to transform from local datum to wgs84, reverse the sign for the reverse transformation from wgs84 to the local datum.

Get datum shift parameters

int
CC_datum_shift(const char *name, double *dx, double *dy, double *dz)

This routine sets the datum shift parameters **dx, dy and dz** (to be interpreted relative to the wgs84 map datum) to the value from the datum table corresponding to the name pointed to by the variable **name**. Returns 1 on success, 0 on failure (e. g. invalid name, not found in datum table).

Get datum shift parameters and ellipsoid

int
CC_get_datum_parameters(const char *name, char *ellps, double *dx, double *dy, double *dz)

This functions sets the datum shift parameters **dx, dy and dz** and the name of the ellipsoid used with the map datum (in **ellps**) from the map datum table. Returns 1 on success, 0 on failure.

*Get datum name for **n**th datum*

char *
CC_datum_name(int n)

Returns a pointer to the name of the **n***th* map datum from the datum table, if **n** is below 0 or greater than the number of entries in the datum table returns a NULL pointer.

char *
CC_datum_description(int n)

*Get datum description for **n**th datum*

Returns a pointer to the textual description for the **n***th* map datum from the datum table. Returns a NULL pointer on failure.

char *
CC_datum_ellipsoid(int n)

*Get ellipsoid name for **n**th datum*

Returns a pointer to the name of the ellipsoid used with the **n***th* map datum from the datum table. Returns a NULL pointer on failure.

int
CC_get_datum_by_name(const char *name)

Get number of map datum

Get the number of the named map datum (**name**) from the datum table, returns the number in the table on success, -1 on failure, e. g. if the datum is not found in the table.

char *
CC_get_datum_by_nbr(int n)

*Get name of **n**th map datum*

Get the name of the **n***th* map datum from the datum table, returns a pointer to a string containing the name on success, a NULL pointer on failure, e. g. if the number is smaller than 0 or greater than the number of entries in the datum table.

21.3.3.2 Calculating a datum shift (Block Shift, Molodensky and Bursa-Wolf transformation)

The following functions provide low-level and high-level interfaces for calculationg map datum shifts on latitude-longitude coordinates. Sources for the formula are cited for the different functions. Some more comments are found in the source code itself.

21.3.3.3 Some hints on accuracy

Generally the accuracy depends on the transformation method used and the accuracy of the parameters supplied to the transformation function. You always must check if the formula is applicable to your problem and supplies the needed accuracy. The author of the functions refuse any responsibility for problems that may arise out of the improper use of the functions.

The following valus can give you an idea of the accuracy of the different methods used in the library:

block shift with cartesian coordinates 10 m

molodensky transformation 5 m

bursa-wolf transformation 1 m

3d similarity transformation 1 m (needs additional national similarity parameters)

multiple regression transformation, other methods up to 10 cm (generally not needed for GRASS)

All transformations need correct input and output ellipsoid for the calculation of **Rm** and **Rn**. Wrong ellipsoid parameters will lead to wrong datum shifts.

In the GRASS GIS the height above the ellipsoid is not known in most cases. Set the value for **Sh** to 0 for the use with the transformation functions. Of course the value for the destination height (**Dh**) makes no sense in this case and should not be displayed to the user.

The ellipsoid flattening **f** is given to the function as the reciprocal value (**1/f**), even if this is not signalled by the variable name. This value is aquired from the function G_get_spheroid_by_name( ) from the *12 GIS Library (p. 79)* or from the function CC_get_spheroid_by_name( ) from the *21.3 Coordinate Conversion Library (coorcnv) (p. 296)*.

datum shift with block shift transformation

int
CC_datum_shift_CC(double Sphi, double Slam, double Sh, double Sa, double Se2, double *Dphi, double *Dlam, double *Dh, double Da, double De2, double dx, double dy, double dz)

This function performs a datum shift with the block shift transformation. The values for the source ellipsoid (**Sa, Se2**) and the destination ellipsoid (**Da, De2**) must be provided to the function as well as the shifting parameters (**dx, dy, dz**). Get these parameters with calls to the CC_get_datum_shift_parameters( ) function from the map datum table and CC_get_spheroid_by_name( ) from the ellipsoid table. In most cases the hight above the ellipsoid (**Sh**) is not known and should be set to 0. Obviously the calculated value for **Dh** should not be used in this case.

datum shift with block shift transformation

int

CC_datum_to_datum_shift_CC(int Sdatum, double Sphi, double Slam, double Sh, int Ddatum, double *Dphi, double *Dlam, double *Dh)

This function provides a high level access to the block shift transformation. The numbers in **Sdatum** and **Ddatum** are the index to the map datum table for source and destination map datum and should be set with the CC_get_datum_by_name( ) function. See the introduction for a comment on **Sh** and **Dh**.

A full description of the molodensky transformation is found at: http://www.utexas.edu/depts/grg/gcraft/

http://www.anzlic.org.au/icsm/gdatum/molodens.html

int
CC_datum_shift_Molodensky(double Sphi, double Slam, double Sh, double Sa, double Se2, double rSf, double *Dphi, double *Dlam, double *Dh, double Da, double De2, double rDf, double dx, double dy, double dz)

datum shift with Molodensky transformation

This function provides low-level access to the Molodensky transformation. You must provide the appropritiate values for the parameters for ellipsoid (**Sa, Se2, rSf, Da, De2, rDf**) and the shifting parameters (**dx, dy, dz**). See the introduction for a comment on **Sh** and **Dh**.

int
CC_datum_to_datum_shift_M(int Sdatum, double Sphi, double Slam, double Sh, int Ddatum, double *Dphi, double *Dlam, double *Dh)

datum shift with Molodensky transformation

High-level access to the datum shift with Molodensky transformation. This function provides a shorthand acces to the CC_datum_shift_Molodensky( ) function. The numbers in **Sdatum** and **Ddatum** are the index to the map datum table for source and destination map datum and should be set with the CC_get_datum_by_name( ) function. See the introduction for a comment on **Sh** and **Dh**.

For a description of the Bursa Wolf transformation see: http://www.posc.org/Epicentre.2\protect\T1\te

CAVEAT: check which sign convention you must use as european and australian/american users use different systems. If you have no rotational parameters and/or no scaling parameter you should use Molodensky or block transformation. The Scale value is expressed as *ppm* (parts per million). Check if you really need an accuracy as high as with this!

int
CC_datum_shift_BursaWolf(double Sphi, double Slam, double Sh, double Sa, double Se2, double *Dphi, double *Dlam, double *Dh, double Da, double De2, double dx, double dy, double dz, double Rx, double Ry, double Rz, double Scale)

datum shift with Bursa Wolf transformation

This function is used to transform latitude-longitude coordinates from one map datum to another map datum with the Bursa Wolf transformation (also known as 3d similarity transformation. The source coordinates are given in **Sphi, Slam** and **Sh**, the destination coordinates are stored in **Dphi, Dlam** and **Dh**. The correct values for the source and target ellipsoid must be set in **Sa, Se2, Da** and **De2**. These values may be obtained from the ellipsoid table with the function CC_get_spheroid( ). The 7 parameters (3 shifting parameters: **dx, dy, dz**, 3 rotational parameters **Rx, Ry, Rz** and the scaling parameter**Scale**) must be provided to the function. The shifting parameters are in meters and the rotational parameters are in radians, the scaling parameter is expressed in *ppm* (parts per million). The function returns 0 on error, 1 on succes.

datum shift with Bursa Wolf transformation

int
CC_datum_to_datum_shift_BW(int Sdatum, double Sphi, double Slam, double Sh, int Ddatum, double *Dphi, double *Dlam, double *Dh)

This function provides a high-level interface for the Bursa Wolf datum shift function CC_datum_shift_BursaWolf( ). The latitude, longitude and ellipsoid heigth (**Sphi, Slam, Sh**) are transformed from the map datum given as an index in **Sdatum** to the the map datum indexed by **Ddatum**, the resulting values are stored in **Dphi, Dlam** and **Dh**. The function returns 0 on error, 1 on success. This is currently not fully supported as the datum database provides no values for a 7 parameter datum shift.

21.3.4 Latitude-Longitude related functions

21.3.4.1 Formatting of latitude-longitude coordinates

format latitude coordinate

int
CC_lat_format(double lat, char *buf)

This function formats the latitude (in seconds) in **lat** into the buffer **buf** as a string in the format *dd.mm.ssH*, where hemisphere *H* is N for northern and S for southern hemisphere. A longitude less than zero is southern, greater than zero is northern hemisphere.

format longitude coordinate

int
CC_lon_format(double lon, char *buf)

This function formats the longitude (in seconds) in **lon** into the buffer **buf** as a string in the format *ddd.mm.ssH*, where hemisphere *H* is W for western hemisphere and

E for eastern hemisphere. A longitude less than zero is eastern hemisphere, greater than 0 is western hemisphere.

int
CC_lat_parts(double lat, int *deg, int *min, double *sec, char *hemisphere)

format latitude coordinate parts

The latitude in seconds in the variable **lat** is formatted into the **deg, min, sec** and **hemisphere** parts given as a pointer.

int
CC_lon_parts(double lon, int *deg, int *min, double *sec, char *hemisphere)

format longitude coordinate parts

The longitude in seconds in the variable **lon** is formatted into the **deg, min, sec** and **hemisphere** parts given as a pointer.

21.3.4.2 Conversion of Latitude-Longitude coordinates to geocentric coordinates

int
CC_ll2geo(double a, double e2, double lat, double lon, double h, double *x, double *y, double *z)

Convert latitude-longitude (radians) to geocentric

Converts latitude and longitude (expressed in radians) to geocentric coordinates. Inputs are the major axis of the local spheroid **a**, the eccentricity squared of the spheroid **e2**, the latitude in arc seconds in **lat**, the longitude in arc seconds in **lon** and the heigth above the spheroid in **h** at this point. The geocentric coordinates are written into the variables given by the pointers to **x**, **y** and **z**. Usually in GRASS the height above the ellipsoid **h** is not known. It should be approximated by 0 or the average elevation above sea level. Obviously this introduces an error in the calculation of the coordinates. The function should return always 0.

int
CC_lld2geo(double a, double e2, double lat, double lon, double h, double *x, double *y, double *z)

convert latitude-longitude (degree) to gecentric

The same as the preceding function G_ll2geo( ), but **lat** and **lon** are given as seconds degree. This function returns 1 in any case.

21.3.4.3 Conversion of geocentric coordinates to Latitude-Longitude coordinates

convert geocentric to latitude-longitude (radians)

int
CC_geo2ll(double a, double e2, double x, double y, double z, double *lat, double *lon, double *h, int n, double stop_delta)

Converts geocentric coordinates to geographic coordinates (in radians, arc seconds). The function expects the major axis of the local spheroid in **a**, the square of eccentricity in **e2**, the geocentric coordinates in meters in **x**, **y** and **z**, the maximum number of iterations when solving the equation (should be set to 50) in **n** and a **stop_delta** value for difference to stop the calculation (should be set to 1e-11). The function returns the number of iterations remaining or 0 if it did not converge. The values are stored in the variables pointed to by **lat** and **lon**, the height above the ellipsoid in meters is stored in **h**.

convert geocentric to latitude-longitude (degree)

int
CC_geo2lld(double a, double e2, double x, double y, double z, double *lat, double *lon, double *h)

This function converts the geocentric coordinates in **x**, **y** and **z** to the geographic latitude and longitude (in seconds degree) pointed to by **lat** and **lon**. The heigth above the ellipsoid is stored in **h** (in meters). This function is a wrapper for the CC_geo2ll( ) function, it supplies default values for the **stop_delta** and the number of iterations **n** to that function and calculates seconds degree instead of radians. It returns 1 if the calculation could be solved with 500 iterations or 0 if the iteration could not be solved with the maximum number of iterations.

21.3.4.4 Scanning of Latitude-Longitude coordinates in strings

scan for latitude string

int
CC_lat_scan(char *string, double *lat)

This functions scans for the latitude in a string of the format *ddd.hh.ssH* (where the hemisphere may be S or N) and sets the result in degree in the variable pointed to by **lat**. It returns 1 on success, 0 on error.

scan for longitude string

int
CC_lon_scan(char *string, double *lon)

This functions scans for the longitude in a string of the format *ddd.hh.ssH* (where the hemisphere may be E or W) and sets the result in degree in the variable pointed to by **lon**. It returns 1 on success, 0 on error.

21.3.4.5 Reading ellipsoid parameters from the database

12.8.6 Miscellaneous (p. 104)

The terms ellipsoid and spheroid are used interchangeable throughout this documentation. Where possible without breaking backward compatibility the functions from the Coordinate Conversion library are named with the term "spheroid" and the functions from the GIS Library with the term "ellipsoid".

int
CC_get_spheroid(const char *name, double *a, double *e2) *get spheroid parameters*

This function sets the ellipsoid parameters for the major axis **a** and the eccentricity squared **e2** to the values for the ellipsoid named by the variable **name**. If the named spheroid is not found in the ellipsoid table, the function returns 0, on success 1 is returned. This function is a wrapper to the function G_get_ellipsoid_by_name( ) from GIS library, which should be used instead.

char
***CC_spheroid_name(int n)** *get spheroid name*

This function returns a pointer to the abbreviated name of the **n***t*h spheroid/ellipsoid from the datum table. If the name is not found in the table a NULL pointer is returned. This function is retained only for backward compatibility with the existing code. The function G_ellipsoid_name( ) from GIS Library or the function CC_get_spheroid_by_number( ) from this library should be used instead.

int
CC_get_spheroid_by_name(const char *name, double *a, double *e2, double *f) *get spheroid parameters*

This function sets the spheroid parameters for major axis **a**, the eccentricity squared **e2** and the inverse flattening **f** for the ellipsoid named by the variable name. If the named spheroid is not found in the ellipsoid table, the function returns 0, on success 1 is returned. This function was introduced to provide the datum shift functions with the value for the reciprocal flattening **f**.

char *get spheroid name*

C_get_spheroid_by_nbr(int n)

This function returns a pointer to the abbreviated name of the **n***th* spheroid/ellipsoid from the datum table. If the name is not found in the table a NULL pointer is returned. This is only a wrapper for the function G_ellipsoid_name( ) from the GIS Library.

21.3.5 Projection and inverse projection, UTM, Transverse Mercator

21.3.5.1 Inverse projection from transverse mercator to latitude-longitude and vice versa

set spheroid int
CC_tm2ll_spheroid(char *name)

This function must be called prior to calling any CC_tm2ll( ) function. It sets the spheroid parameters for the ellipse in **name**, returns -1 on an unrecongnized spheroid, -2 on an internal error and 1 on success.

set ellipsoid parameters int
CC_tm2ll_spheroid_parameters(double a, double e2)

This function is called by CC_tm2ll_spheroid( ) to set the ellipsoid major axis **a** and the eccentricity squared **e2**. It can be called directly for unknown ellipsoids. Returns -2 on illegal values for a or e2, 1 on success.

set longitude of central meridian int
CC_tm2ll_zone(int zone)

This function must be called before invoking CC_tm2ll_north( ) to set the zone. The zone value **zone** must be non-zero, positive means northern hemisphere, negative means southern hemisphere. It is used to set the longitude of the central meridian used for tm to latitude-longitude conversions.

set tm northing int
CC_tm2ll_north(double northing)

This functions sets the tm
textbfnorthing and must be called before CC_tm2ll( ) is used. It returns -1 if the related rectifying latitude exceeds 1.47 radians, 1 on success.

int *set tm easting*
CC_tm2ll(double easting, double *lat, double *lon)

This function computes the latitude and longitude in **lat** and **lon** from the **easting**. The function CC_tm2ll_north must be called first to set the northing value. Returns -1 if the longitude is more than .16 radians from the center of the tm zone, 1 otherwise, i. e. on success.

int *compute latitude-longitude from easting*
CC_ll2tm(double lat, double lon, double *easting, double *northing, int *zone)

Compute the **easting**, **northing** and **zone** from latitude-longitude values in **lat** and **lon**. If the **zone** is not zero, the point is forced into this zone, otherwise the zone is computed from the latitude-longitude coordinates. Returns -1 if the latitude is above 84 degrees, -2 if the longitude is too far from the center of the zone and 1 on success.

21.3.5.2 Inverse projection from UTM to latitude-longitude and vice versa

int *utm to latitude-longitude conversion initialization*
CC_u2ll_spheroid(char *name)

This function must be called first to set the spheroid parameters for the ellipse with the function CC_spheroid_name( ). The function returns 1 on success, -1 on an unrecognized spheroid and -2 on an internal error.

int *get spheroid parameters for ellipse*
CC_u2ll_spheroid_parameters(double a, double e2)

This routine is called by CC_u2ll_spheroid( ) to set the ellipsoid parameters for the major axis **a** and the eccentricity squared **e2**. The function can be called directly for unknown ellipsoids and returns 1 on success, -2 on illegal values for **a** or **e2**.

int *set utm zone*
CC_u2ll_zone(int zone)

This function must be called before the execution of CC_u2ll_north( ) to set the longitude of the central meridian for utm to latitude-longitude conversions. The value for **zone** must be non-zero, positive value means northern hemisphere, negative value means southern hemisphere. Always returns 0.

set the utm north int
CC_u2ll_north(double northing)

This routine must be called befor CC_u2ll( ) is invoked in order to set the utm north. Returns 1 on success, -1 if the related rectifying latitude exceeds 1.47 radians.

compute latitude-longitude from east int
CC_u2ll(double easting, double *lat, double *lon)

This functions computes latitude-longitude in **lat** and **lon** from the **easting**, CC_u2ll_north( ) must be already called with the **northing** value. Returns 1 on success, -1 if lon is more than .16 radians from the center of the utm zone.

compute utm easting and northing from latitude-longitude int
CC_ll2u(double lat, double lon, double *easting, double *northing, int *zone)

This function computes the utm **easting** and **northing** from the latitude-longitude coordinates in **lat** and **lon**. If the value for **zone** is set to anything else than zeor (verify?!), the point is forced into this zone, otherwise the zone is computed internally. Returns 1 on success, -1 if the latitude is above 84 degrees, -2 if the longitude is too far from the center of the utm zone.

21.3.6 changes to gislib

21.3.6.1 Miscellaneous

return ellopsoid name char *
G_ellipsoid_name (int n)

This routine returns a pointer to a string containg the name for the **n***th* ellipsoid in the GRASS ellipsoid table; NULL when **n** is below zero or too large. It can be used as follows:

```
int n ;
char *name ;
for ( n=0 ; name=G_ellipsoid_name(n) ; n++ )
fprintf(stdout, "%s\n", name);
```

get ellipsoid parameters by name int
G_get_ellipsoid_by_name (char *name, double *a, double *e2)

This routine returns the semi-major axis **a** (in meters) and eccentricity squared **e2** for the named ellipsoid. Returns 1 if **name** is a known ellipsoid, 0 otherwise.

int
G_get_ellipsoid_parameters (double *a, double *e2)

get ellipsoid parameters

This routine returns the semi-major axis **a** (in meters) and the eccentricity squared **e2** for the ellipsoid associated with the database. If there is no ellipsoid explicitly associated with the database, it returns the values for the WGS 84 ellipsoid.

int
G_get_spheroid_by_name(const char *name, double *a, double *e2, double *f)

get spheroid parameters by name

This function returns the semi-major axis **a** (in meters), the eccentricity squared **e2** and the inverse flattening **f** for the named ellipsoid. Returns 1 if **name** is a known ellipsoid, 0 otherwise.

char *
G_ellipsoid_name(int n)

get ellipsoid name

This function returns a pointer to the short name for the **n***th* ellipsoid. If **n** is less than 0 or greater than the number of known ellipsoids, it returns a NULL pointer.

char *
G_ellipsoid_description(int n)

*get description for **n**th ellipsoid*

This function returns a pointer to the description text for the **n***th* ellipsoid. If **n** is less than 0 or greater than the number of known ellipsoids, it returns a NULL pointer.

char *
G_database_datum_name()

get datum name for database

Returns a pointer to the name of the map datum of the current database. If there is no map datum explicitely associated with the acutal database, the standard map datum WGS84 is returned, on error a NULL pointer is returned.

get datum parameters from database

int
G_get_datum_parameters(double *a, double *e2, double *f, double *dx, double *dy, double *dz)

This function sets the datum parameters for the map datum of the current database. These are the semi-major axis **a** (in meters), the eccentricity squared **e2** and the inverse flattening **f** of the spheroid associated with the database and the x shift **dx**, the y shift **dy** and the z shift **dz** of the map datum associated with the database. If there is no map datum explicitely associated with the actual database, the standard values for the WGS84 spheroid and map datum are set. The funcion returns 1 on success, 0 if the default WGS84 parameters are set. If an error occurs, the function dies with a diagnostic message (HINT: to change, very bad practice to die in a library function!).

get datum parameters from database

int
G_get_datum_parameters7(double *a, double *e2, double *f, double *dx, double *dy, double *dz, double *rx, double *ry, double *rz, double *m)

This is a placeholder as the 7 parameter datum shift support is not implemented yet.

ask for a valid datum name

int
G_ask_datum_name(char *datum)

This function asks the user interactively for a valid datum name from the datum table. The datum name is stored in the character array pointed to by **datum**. The function returns 1 on sucess, -1 if no datum was entered on command line and 0 on internal error.

22 Grid3D raster volume library

Authors:

Roman Waupotitsch and Michael Shapiro Helena Mitasova, Bill Brown, Lubos Mitas, Jaro Hofierka

22.1 Directory Structure

The file format consists of a mapset element *grid3* which contains a directory for every map. The elements for each map are

```
3d region file
color file (color)
categories file (cats)
range file (range)
timestamp file /* not yet implemented */
cell file (cell)
header file (cellhd)
a directory containing display files (dsp)
```

There is also a *colr2* mechanism provided. *colr2* color tables are stored in *grid3/colr2/MAPSET/MAP*.

Note: color, categories, and the range can be used in the same way as in *2d* GRASS with the exception of reading and writng. *3d* read and write functions have to be used for this purpose.

22.2 Data File Format

- Cell-values can be either double or float.
- Values are written in XDR-format.
- NULL-values are stored in an embedded fashion.
- The cell-values are organized in *3d*-tiles.
- The tile dimensions can be chosen when a new map is opened.
- Every tile of a map has the same dimension except those which overlap the region boundaries.

- Compression is used to store tiles.

The data file has the following format:

```
xdr_int nofBytesLong;
xdr_int nofBytesUsed;
encoded_long indexOffset;
compressed_tile[] tiles;
compressed_encoded_long[] index;
```

22.2.1 Transportability of data file

All numbers stored in the data file are either XDR-encoded or encoded by some other method (for variables of type long only).

22.2.2 Tile Data NULL-values

G3D uses the same functions as *2d* GRASS to set and test NULL-values. The storage in the file is different though. NULL-values are stored with a special bit-pattern if maximum precision is chosen. They are stored by adding an additional bit if the precision is smaller.

22.2.3 Tile Data Compression

There are three methods of compression provided. The compression methods can either be those defined by default, set by environment variables or explicitly set at run-time.

```
Precision
RLE
```

Precision indicates how many of the mantissa bits should be stored on file. This number can be any value between 0 and 23 for floats and between 0 and 52 for doubles. Choosing a small precision is the most effective way to achieve good compression.

RLE takes advantage of possible repetitions of the exponents and the NULL-bit structure. Using RLE does not significantly increase the running time. If for some tile the non-RLEed version is smaller in size, RLE is not used for this tile.

The default and suggested setting is to use precision and RLE.

Additional compression is achieved by storing the extra NULL-bit in a separate bit-array. Using this scheme NULL-values need not actually be represented in the array of cell values. This array is stored together with the cell-values of the tile.

22.2.4 Tile Cache

Tiles can either be read and written directly or use an intermediate cache instead.

In non-cache mode the application should only use the functions

int
G3d_readTile ()

and

int
G3d_writeTile ()

to read and write tiles. The application can use one tile provided by the map structure as buffer. See `G3d_getTilePtr ()`.

In cache mode the application can access cell-values directly by their coordinates. The corresponding functions are

int
G3d_getValue ()

and

int
G3d_putValue ()

and their corresponding typed versions.

If the map is new then in addition to the memory-cache a file-cache is provided. This allows the application to write the cell-values in any arbitrary order. Tiles are written (flushed) to the data-file either at closing time or if explicitly requested.

If the map is new `G3d_getValue ()` can be used even if the tile which contains the cell has already been flushed to the data file. In this case the tile is simply read back into the memory-cache from the data file.

Explicitly flushing tiles can have the advantage that less disk space is occupied since tiles are stored in a uncompressed fashion in the file-cache. Flushing tiles explicitly can cause problems with accuracy though if precision is less than the maximum precision and an already flushed

value is used for computations later in the program.

The type of the cell-values of the tiles in memory can be chosen independently of the type of the tiles in the file. Here, once again one has to consider possible problems arising from mixing different precisions.

As an example consider the case where the data is stored in the file with double precision and the tiles are stored in memory in single precision. Then using `G3d_getValue ()` will actually return a double precision number whose precision is only 23 bits. It is therefore a good idea to use the types in the memory consistently.

22.2.5 Header File

The header file has the following format:

```
Proj: 1
Zone: 1
North: 2.0000000000000
South: 0.5000000000000
East: 4.0000000000000
West: 3.0000000000000
Top: 6.0000000000000
Bottom: 5.0000000000000
nofRows: 30
nofCols: 20
nofDepths: 14
e-w resol: 0.05
n-s resol: 0.05
t-b resol: 0.071428571
TileDimensionX: 8
TileDimensionY: 8
TileDimensionZ: 8
CellType: double
useCompression: 1
useRle: 1
Precision: -1
nofHeaderBytes: 12
useXdr: 1
hasIndex: 1
Units: none
```

Except for the first 14 fields the entries of the header file should not be modified. The precision value -1 indicates that maximum precision is used.

Binary files not in G3D format can be read by the library. The following actions have to be taken:

Make a new map directory in the *grid3* element of the mapset (say *mymap*). Copy the file into *mymap/cell* and generate a header file *mymap/cellhd*.

In the following example the relevant values of *mymap/cellhd* are shown:

```
TileDimensionX: A
TileDimensionY: B
TileDimensionZ: C
useCompression: 0
useRle: 0
Precision: -1
nofHeaderBytes: X
useXdr: 0
hasIndex: 0
```

The values of *A*, *B*, and *C* have to be chosen according to one of the following patterns:

```
A >= 1, B == 1, C == 1, or
A >= nofRows, B >= 1, C == 1, or
A >= nofRows, B >= nofCols, C >= 1.
```

A larger tile size reduces the number of tile-reads. If in the third pattern *C* is chosen larger than or equal to *nofDepths*, the entire region is considered one large tile.

The value *nofHeaderBytes* indicates the offset in the file to the first data entry.

For performance reasons it is a good idea to use function `G3d_retile()` before using the file in other applications.

22.2.6 Region Structure

```
typedef struct{

    double north, south;
    double east, west;
    double top, bottom;

    int rows, cols, depths;/* data dimensions in cells */

    double ns_res, ew_res, tb_res;

    int proj;  /* Projection (see gis.h) */
    int zone;  /* Projection zone (see gis.h) */

} G3D\_Region;
```

22.2.7 Windows

Window capability similar to that of *2d* GRASS is provided (compare *9.1 Region (p. 61)*). Additional features are the window for the third dimension as well as the possibility to choose a

different window for every map. The window can be specified at the time of opening an old map. It can be modified at any time later in the program. The resampling method can be the default nearest neighbor method as well as an application provided method.

The default *3d* window file is *WIND3* located in the mapset. Application programs should use `G3d_useWindowParams ()` to allow the user to overwrite this default.

The window file has the following format:

```
Proj: 1
Zone: 1
North: 2.0
South: 0.5
East: 4.0
West: 3.0
Top: 5.0
Bottom: 6.0
nofRows: 30
nofCols: 20
nofDepths: 14
e-w resol: 0.05000000000000000
n-s resol: 0.05000000000000000
t-b resol: 0.07142857142857142
```

Note: after reading the window file the fields *e-w*, *n-s*, and *t-b* are recomputed internally.

A note about windows and caching. Caching is performed on the level of tiles read from the file. There is no caching performed on resampled data. This is different from *2d* GRASS since resampling for a specific value is performed every time it is being accessed.

22.2.8 Masks

G3D provides a mask for the *3d* region. The mask structure is automatically initialized at the time the first file is opened. The same structure is used for all the files. The default for every file is that the mask is turned off. If masking should be performed, the application program has to turn on masking explicitly. If masking is turned on for a file, the cell-values of a tile are automatically checked against the mask. Values which are masked out, are set to NULL.

Note: changing the status of masking after one or more tiles have already been read does not affect the tiles which are already stored in the cache.

Any arbitrary g3d file can be used as mask file: NULL-values are interpreted as `"mask-out"`, all other values are interpreted as `"don't mask out"`. Using *r3.mask* to convert a g3d file into a mask file instead of simply copying (or renaming) the directory will significantly reduce to amount of disk space and the access time for the mask.

22.2.9 Include File

Exported G3D constants and structures can be found in *G3d.h*.

22.3 G3D Defaults

There are three methods to set default variables. First, the default can be set at compile time in *g3ddefault.c*. This value has lowest priority.

Second, the default can be set via an environment variable. Third, the value can be set explicitly set at run time. This value has highest priority.

There are also functions provided to query the value.

22.3.1 Cache Mode

22.3.1.1 Limiting the maximum cache size

The limit is specified in bytes. It is a limit on the size of cell-data stored in the cache and does not include the support structure.

Default G3D_CACHE_SIZE_MAX_DEFAULT. This is currently set to 2meg and can be changed at compilation time of the library.

Environment variable G3D_MAX_CACHE_SIZE.

void *Set cache limit*
G3d_setCacheLimit (int nBytes)

int *Get cache limit*
G3d_getCacheLimit (int nBytes)

22.3.1.2 Setting the cache size

This value specifies the number of tiles stored in the cache. It is the value used if at opening time of a map G3D_USE_CACHE_DEFAULT is used for the cache mode. Any other value used at opening time will supersede the default value. A default value of 0 indicates that non-cache mode should be used by default.

Default G3D_CACHE_SIZE_DEFAULT. This is currently set to 1000 and can be changed at compilation time of the library.

Environment variable G3D_DEFAULT_CACHE_SIZE.

void
G3d_setCacheSize (int nTiles)

int
G3d_getCacheSize ()

22.3.2 Compression

22.3.2.1 Toggling compression mode

This value specifies whether compression should be used while writing a new map. It does not have any effect on old maps.

Default G3D_COMPRESSION_DEFAULT. This is set to G3D_COMPRESSION. This default should not be changed.

Environment variables G3D_USE_COMPRESSION and G3D_NO_COMPRESSION.

See functions G3d_setCompressionMode () (cf. Section 22.3.2.3) and G3d_getCompressionMode () (cf. Section 22.3.2.3).

22.3.2.2 Toggling RLE compression

This value specifies whether RLE compression should be used (in addition to precision).

Default G3D_USE_RLE_DEFAULT. This is currently set to G3D_USE_RLE and can be changed at compilation time of the library.

Environment variables G3D_USE_RLE and G3D_NO_RLE.

See functions G3d_setCompressionMode () (cf. Section 22.3.2.3) and G3d_getCompressionMode () (cf. Section 22.3.2.3).

22.3.2.3 Setting the precision

This number specifies how many mantissa bits should be used when writing a cell value. The minimum value is 0. The maximum value is 23 or G3D_MAX_PRECISION for type G3D_FLOAT, it is 52 or G3D_MAX_PRECISION for type G3D_DOUBLE.

Default G3D_PRECISION_DEFAULT. This is set to G3D_MAX_PRECISION. This default should not be changed.

Environment variables G3D_PRECISION and G3D_MAX_PRECISION.

void
G3d_setCompressionMode (int doCompress, int doLzw, int doRle, int precision)

doCompress should be one of G3D_NO_COMPRESSION and G3D_COMPRESSION, *doRle* should be either G3D_NO_RLE or G3D_USE_RLE, and *precision* should be either G3D_MAX_PRECISION or a positive integer.

void
G3d_getCompressionMode (int *doCompress, int *doLzw, int *doRle, int *precision)

22.3.3 Tiles

22.3.3.1 Setting the tile dimensions

The dimensions are specified in number of cell.

Defaults G3D_TILE_X_DEFAULT, G3D_TILE_Y_DEFAULT, and G3D_TILE_Z_DEFAULT.
These are currently set to 8 and can be changed at compilation time of the library.

Environment variables G3D_TILE_DIMENSION_X, G3D_TILE_DIMENSION_Y, and G3D_TILE_DIMENSION_Z.

void
G3d_setTileDimension (int tileX, int tileY, int tileZ)

void
G3d_getTileDimension (int *tileX, int *tileY, int *tileZ)

22.3.3.2 Setting the tile cell-value type

Specifies which type is used to write cell-values on file. This type can be chosen independently of the type used to store cell-values in memory.

Default G3D_FILE_TYPE_DEFAULT. This is set to G3D_DOUBLE. This default should not be changed.

Environment variables G3D_WRITE_FLOAT and G3D_WRITE_DOUBLE.

void
G3d_setFileType (int type)

int
G3d_getFileType (int type)

22.3.4 Setting the window

The window is set from a *3d* window file.

The default *3d* window file is *WIND3* located in the current mapset.

Possible choices for *3d* window files are *name* which refers to a window file in the *3d* window database located at *windows3d* of the current mapset; or file names which are identified by a leading `"/"` or `"."`; or fully qualified names, i.e. *file@mapset* which refer to window files in the *3d* window database of mapset. Note, that names *WIND3* and *WIND3@mapset* do not specify the default window name in the (current) mapset but rather a window file in the window database of the (current) mapset.

Environment variable G3D_DEFAULT_WINDOW3D.

See functions

`G3d_useWindowParams ()`,

`G3d_setWindow ()`, and

`G3d_setWindowMap ()`.

22.3.5 Setting the Units

Default `"none"`.

No environment variable.

void
G3d_setUnit (unit) char *unit;

22.3.6 Error Handling: Setting the error function

This variable specifies the function which is invoked when an error (not a fatal error) occurs. For example setting the error function to `G3d_fatalError` simplifies debugging with dbx

and also might show errors which are missed because the application does not check the return value.

Default `G3d_skipError`.

Environment variables G3D_USE_FATAL_ERROR and G3D_USE_PRINT_ERROR.

void
G3d_setErrorFun (void (*fun)(char *))

The following 3 functions are possible choices for error functions.

void
G3d_skipError (char (*msg)(char *))

This function ignores the error.

void
G3d_printError (char (*msg)(char *))

This function prints the error message *msg* to *stderr* and returns.

void
G3d_fatalError (char (*msg)(char *))

This function prints the error message *msg* to *stderr*, flushes *stdout* and *stderr*, and terminates the program with a segementation fault.

22.4 G3D Function Index

22.4.1 Opening and Closing G3D Files

void *
G3d_openCellOld (char *name, char *mapset, G3D_Region *window, int type, int cache)

Opens existing g3d-file *name* in *mapset*.
Tiles are stored in memory with *type* which must be any of G3D_FLOAT, G3D_DOUBLE, or G3D_TILE_SAME_AS_FILE. *cache*

specifies the cache-mode used and must be either G3D_NO_CACHE, G3D_USE_CACHE_DEFAULT, G3D_USE_CACHE_X, G3D_USE_CACHE_Y, G3D_USE_CACHE_Z, G3D_USE_CACHE_XY, G3D_USE_CACHE_XZ, G3D_USE_CACHE_YZ, G3D_USE_CACHE_XYZ, the result of `G3d_cacheSizeEncode ()` (cf. Section 22.4.6), or any positive integer which specifies the number of tiles buffered in the cache. *window* sets the window-region for the map. It is either a pointer to a window structure or G3D_DEFAULT_WINDOW, which uses the window stored at initialization time or set via `G3d_setWindow ()` (cf. Section 22.4.16). To modify the window for the map after it has already been opened use `G3d_setWindowMap ()` (cf. Section 22.4.16).
Returns a pointer to the cell structure ... if successful, NULL ... otherwise.

void *
G3d_openCellNew (char *name, int type, int cache, G3D_Region *region)

Opens new g3d-file with *name* in the current mapset. Tiles are stored in memory with *type* which must be one of G3D_FLOAT, G3D_DOUBLE, or G3D_TILE_SAME_AS_FILE. *cache* specifies the cache-mode used and must be either G3D_NO_CACHE, G3D_USE_CACHE_DEFAULT, G3D_USE_CACHE_X, G3D_USE_CACHE_Y, G3D_USE_CACHE_Z, G3D_USE_CACHE_XY, G3D_USE_CACHE_XZ, G3D_USE_CACHE_YZ, G3D_USE_CACHE_XYZ, the result of `G3d_cacheSizeEncode ()` (cf. Section 22.4.6), or any positive integer which specifies the number of tiles buffered in the cache. *region* specifies the 3d region.
Returns a pointer to the cell structure ... if successful, NULL ... otherwise.

void *
G3d_openCellNewParam (char *name, int typeIntern, int cache, G3D_Region *region, int type, int doLzw, int doRle, int precision, int tileX, int tileY, int tileZ)

Opens new g3d-file with *name* in the current mapset. Tiles are stored in memory with *typeIntern* which must be one of G3D_FLOAT, G3D_DOUBLE, or G3D_TILE_SAME_AS_FILE. *cache* specifies the cache-mode used and must be either G3D_NO_CACHE, G3D_USE_CACHE_DEFAULT, G3D_USE_CACHE_X, G3D_USE_CACHE_Y, G3D_USE_CACHE_Z, G3D_USE_CACHE_XY, G3D_USE_CACHE_XZ, G3D_USE_CACHE_YZ, G3D_USE_CACHE_XYZ, the result of `G3d_cacheSizeEncode ()` (cf. Section 22.4.6), or any positive integer which specifies the number of tiles buffered in the cache. *region* specifies the 3d region.
In addition the properties of the new file have to be specified. It is assumed by default that compression is used. This function first sets the global default values to the specified values, and then restores the original global defaults. This function

can be used in conjunction with `G3d_setStandard3dInputParams ()` (cf. Section 22.4.18) and `G3d_getStandard3dParams ()`.
Returns a pointer to the cell structure ... if successful, NULL ... otherwise.

int
G3d_closeCell (void *map)

Closes g3d-file. If *map* is new and cache-mode is used for *map* then every tile which is not flushed before closing is flushed.
Returns 1 ... if successful, 0 ... otherwise.

22.4.2 Reading and Writing Tiles

These functions read or write data directly to the file (after performing the appropriate compression) without going through the cache. In order to avoid unexpected side-effects the use of these functions in cache mode is discouraged.

int
G3d_readTile (void *map, char *tileIndex, int tile, int type)

Reads tile with index *tileIndex* into the *tile* buffer. The cells are stored with type *type* which must be one of G3D_FLOAT and G3D_DOUBLE. If the tile with *tileIndex* is not stored on the file corresponding to *map*, and *tileIndex* is a valid index *tile* is filled with NULL-values.
Returns 1 ... if successful, 0 ... otherwise.

int
G3d_readTileFloat (void *map, char *tileIndex, int tile)

Is equivalent to G3d_readTile (map, tileIndex, tile, G3D_FLOAT).

int
G3d_readTileDouble (void *map, char *tileIndex, int tile)

Is equivalent to G3d_readTile (map, tileIndex, tile, G3D_DOUBLE).

int
G3d_writeTile (void *map, char *tileIndex, int tile, int type)

Writes tile with index *tileIndex* to the file corresponding to *map*. It is assumed that the cells in *tile* are of *type* which must be one of G3D_FLOAT and G3D_DOUBLE. The actual type used to write the tile depends on the type specified at the time when *map* is initialized.
A tile can only be written once. Subsequent attempts to write the same tile are ignored.
Returns 1 ... if successful, 2 ... if write request was ignored, 0 ... otherwise.

int
G3d_writeTileFloat (void *map, char *tileIndex, int tile)

Is equivalent to `G3d_writeTile (map, tileIndex, tile, G3D_FLOAT)`.

int
G3d_writeTileDouble (void *map, char *tileIndex, int tile)

Is equivalent to `G3d_writeTile (map, tileIndex, tile, G3D_DOUBLE)`.

22.4.3 Reading and Writing Cells

void
G3d_getValue (void *map, int x, int y, int z, char *value, int type)

Returns in **value* the cell-value of the cell with window-coordinate *(x, y, z)*. The value returned is of *type*.
This function invokes a fatal error if an error occurs.

float
G3d_getFloat (void *map, int x, int y, int z)

Is equivalent to `G3d_getValue (map, x, y, z, &value, G3D_FLOAT)`; return value.

double
G3d_getDouble (void *map, int x, int y, int z)

Is equivalent to `G3d_getValue (map, x, y, z, &value, G3D_DOUBLE);` return value.

void
G3d_getValueRegion (void *map, int x, int y, int z, char*value, int type)

Returns in **value* the cell-value of the cell with region-coordinate *(x, y, z)*. The value returned is of *type*. Here *region* means the coordinate in the cube of data in the file, i.e. ignoring geographic coordinates.
This function invokes a fatal error if an error occurs.

float
G3d_getFloatRegion (void *map, int x, int y, int z)

Is equivalent to `G3d_getValueRegion (map, x, y, z, &value, G3D_FLOAT);` return value.

double
G3d_getDoubleRegion (void *map, int x, int y, int z)

Is equivalent to `G3d_getValueRegion (map, x, y, z, &value, G3D_DOUBLE);` return value.

int
G3d_putValue (void *map, int x, int y, int z, char *value, int type)

After converting **value* of *type* into the type specified at the initialization time (i.e. *typeIntern*) this function writes the value into the tile buffer corresponding to cell-coordinate *(x, y, z)*.
Returns
1 ... if successful, 0 ... otherwise.

int
G3d_putFloat (void *map, int x, int y, int z, char *value)

Is equivalent to G3d_putValue (map, x, y, z, &value, G3D_FLOAT).

int
G3d_putDouble (void *map, int x, int y, int z, char *value)

Is equivalent to G3d_putValue (map, x, y, z, &value, G3D_DOUBLE).

22.4.4 Loading and Removing Tiles

char *
G3d_getTilePtr (void *map, int tileIndex)

This function returns a pointer to a tile which contains the data for the tile with index *tileIndex*. The type of the data stored in the tile depends on the type specified at the initialization time of *map*. The functionality is different depending on whether *map* is old or new and depending on the cache-mode of *map*.

If *map* is old and the cache is not used the tile with *tileIndex* is read from file and stored in the buffer provided by the map structure. The pointer to this buffer is returned. If the buffer already contains the tile with *tileIndex* reading is skipped. Data which was stored in earlier calls to `G3d_getTilePtr` is destroyed. If the tile with *tileIndex* is not stored on the file corresponding to *map*, and *tileIndex* is a valid index the buffer is filled with NULL-values.

If *map* is old and the cache is used the tile with *tileIndex* is read from file and stored in one of the cache buffers. The pointer to buffer is returned. If no free cache buffer is available an unlocked cache-buffer is freed up and the new tile is stored in its place. If the tile with *tileIndex* is not stored on the file corresponding to *map*, and *tileIndex* is a valid index the buffer is filled with NULL-values. If one of the cache buffers already contains the tile with *tileIndex* reading is skipped and the pointer to this buffer is returned.

If *map* is new and the cache is not used the functionality is the same as if *map* is old and the cache is not used. If the tile with *tileIndex* is already stored on file, it is read into the buffer, if not, the cells are set to null-values. If the buffer corresponding to the pointer is used for writing, subsequent calls to `G3d_getTilePtr` may destroy the values already stored in the buffer. Use `G3d_flushTile` to write the buffer to the file before reusing it for a different index. The use of this buffer as write buffer is discouraged.

If *map* is new and the cache is used the functionality is the same as if *map* is old and the cache is used with the following exception. If *tileIndex* is a valid index and the tile with this index is not found in the cache and is not stored on the file corresponding to *map*, then the file cache is queried next. If the file-cache contains the tile it is loaded into the cache (memory-cache). Only if the file-cache does not contain the tile it is filled with NULL-values. Tile contents of buffers are never

destroyed. If a cache buffer needs to be freed up, and the tile stored in the buffer has not been written to the file corresponding to *map* yet, the tile is copied into the file-cache.

Care has to be taken if this function is used in non-cache mode since it is implicitly invoked every time a read or write request is issued. The only I/O-functions for which it is safe to assume that they do not invoke `G3d_getTilePtr` are `G3d_readTile ()` and `G3d_writeTile ()` and their corresponding type-specific versions.
Returns a pointer to a buffer ... if successful, NULL ... otherwise.

int
G3d_tileLoad (void *map, int tileIndex)

Same functionality as `G3d_getTilePtr ()` but does not return the pointer.
Returns 1 ... if successful, 0 ... otherwise.

int
G3d_removeTile (void *map, inttileIndex)

Removes a tile from memory-cache if tile is in memory-cache. For new maps the application does not know whether the tile is in the memory-cache or in the file-cache. Therefore, for new maps this function should be preceded by `G3d_tileLoad ()`.
(Question: Is this a useful function?)
Returns 1 ... if successful, 0 ... otherwise.

22.4.5 Write Functions used in Cache Mode

int
G3d_flushTile (void *map, int tileIndex)

Writes the tile with *tileIndex* to the file corresponding to *map* and removes the tile from the cache (in non-cache mode the buffer provided by the map-structure is written).
If this tile has already been written before the write request is ignored. If the tile was never referred to before the invokation of G3d_flushTile, a tile filled with NULL-values is written.
Returns 1 ... if successful, 0 ... otherwise.

int
G3d_flushTileCube (void *map, int xMin, int yMin, int zMin, int xMax, int yMax, int zMax)

Writes the tiles with tile-coordinates contained in the axis-parallel cube with vertices *(xMin, yMin, zMin)* and *(xMax, yMax, zMax*). Tiles which are not stored in the cache are written as NULL-tiles. Write attempts for tiles which have already been written earlier are ignored.
Returns 1 ... if successful, 0 ... otherwise.

int
G3d_flushTilesInCube (void *map, int xMin, int yMin, int zMin, int xMax, int yMax, int zMax)

Writes those tiles for which *every* cell has coordinate contained in the axis-parallel cube defined by the vertices with cell-coordinates *(xMin, yMin, zMin)* and *(xMax, yMax, zMax)*.
Tiles which are not stored in the cache are written as NULL-tiles. Write attempts for tiles which have already been written earlier are ignored.
Returns 1 ... if successful, 0 ... otherwise.

22.4.6 Locking and Unlocking Tiles, and Cycles

int
G3d_lockTile (void *map, int tileIndex)

Locks tile with *tileIndex* in cache. If after locking fewer than the minimum number of unlocked tiles are unlocked, the lock request is ignored.
Returns 1 ... if successful, -1 ... if request is ignored, 0 ... otherwise.

int
G3d_unlockTile (void *map, int tileIndex)

Unlocks tile with *tileIndex*.
Returns 1 ... if successful, 0 ... otherwise.

int
G3d_unlockAll (void *map)

Unlocks every tile in cache of *map*.
Returns 1 ... if successful, 0 ... otherwise.

void
G3d_autolockOn (void *map)

Turns autolock mode on.

void
G3d_autolockOff (void *map)

Turns autolock mode Off.

void
G3d_minUnlocked (void *map, int minUnlocked)

Sets the minimum number of unlocked tiles to *minUnlocked*. This function should be used in combination with `G3d_unlockAll ()` in order to avoid situations where the new minimum is larger than the actual number of unlocked tiles.
minUnlocked must be one of G3D_USE_CACHE_X, G3D_USE_CACHE_Y, G3D_USE_CACHE_Z, G3D_USE_CACHE_XY, G3D_USE_CACHE_XZ, G3D_USE_CACHE_YZ, G3D_USE_CACHE_XYZ, the result of G3d_cacheSizeEncode () (cf. Section 22.4.6), or any positive integer which explicitly specifies the number of tiles.

int
G3d_beginCycle (void *map)

Starts a new cycle.
Returns 1 ... if successful, 0 ... otherwise.

int
G3d_endCycle (void *map)

Ends a cycle.
Returns 1 ... if successful, 0 ... otherwise.

int
G3d_cacheSizeEncode (int cacheCode, int n)

Returns a number which encodes multiplicity *n* of *cacheCode*. This value can be used to specify the size of the cache.
If *cacheCode* is the size (in tiles) of the cache the function returns *cacheCode * n*.
If *cacheCode* is G3D_USE_CACHE_DEFAULT the function returns G3D_USE_CACHE_DEFAULT.
If *cacheCode* is G3D_USE_CACHE_??? the function returns a value encoding G3D_USE_CACHE_??? and *n*. Here G3D_USE_CACHE_??? is one of G3D_USE_CACHE_X, G3D_USE_CACHE_Y, G3D_USE_CACHE_Z, G3D_USE_CACHE_XY, G3D_USE_CACHE_XZ, G3D_USE_CACHE_YZ, or G3D_USE_CACHE_XYZ, where e.g. G3D_USE_CACHE_X specifies that the cache should store as many tiles as there exist in one row along the x-axis of the tile cube, and G3D_USE_CACHE_XY specifies that the cache should store as many tiles as there exist in one slice of the tile cube with constant Z coordinate.

22.4.7 Reading Volumes

int
G3d_getVolume (void *map, double originNorth, double originWest, double originBottom, double vxNorth, double vxWest, double vxBottom, double vyNorth, double vyWest, double vyBottom, double vzNorth, double vzWest, double vzBottom, int nx, int ny, int nz, char *volumeBuf, int type)

Resamples the cube defined by *origin* and the 3 vertices *vx*, *vy*, and *vz* which are incident to the 3 edges adjacent to *origin*. The resampled cube is stored in *volumeBuf* which is a cube with dimensions *(nx, ny, nz)*.
The method of sampling is nearest neighbor sampling.
The values stored are of *type*.
Returns 1 ... if successful, 0 ... otherwise.

int
G3d_getAllignedVolume (void *map, double originNorth, double originWest, double originBottom, double lengthNorth, double lengthWest, double lengthBottom, int nx, int ny, int nz, char *volumeBuf, int type)

Resamples the axis-parallel cube defined by *origin* and the lengths of the 3 edges adjacent to *origin*. The resampled cube is stored in *volumeBuf* which is a cube with dimensions *(nx, ny, nz)*. The method of sampling is nearest neighbor sampling.
The values stored are of *type*.
Returns 1 ... if successful, 0 ... otherwise.

22.4.8 Allocating and Freeing Memory

void *
G3d_malloc (int nBytes)

Same as *malloc (nBytes)*, except that in case of error `G3d_error ()` is invoked.
Returns a pointer ... if successful, NULL ... otherwise.

void *
G3d_realloc (void *ptr, int nBytes)

Same as *realloc (ptr, nBytes)*, except that in case of error `G3d_error ()` is invoked.
Returns a pointer ... if successful, NULL ... otherwise.

void
G3d_free (void *ptr)

Same as *free (ptr)*.

char *
G3d_allocTilesType (void *map, int nofTiles, int type)

Allocates a vector of *nofTiles* tiles with the same dimensions as the tiles of *map* and large enough to store cell-values of *type*.
Returns a pointer to the vector ... if successful, NULL ... otherwise.

char *
G3d_allocTiles (void *map, int nofTiles)

Is equivalent to G3d_allocTilesType (map, nofTiles, G3d_fileTypeMap (map)).

void
G3d_freeTiles (char *tiles)

Is equivalent to `G3d_free (tiles);`

22.4.9 G3D Null Value Support

void
G3d_isNullValueNum (void *n, int type)

Returns 1 if the value of **n* is a NULL-value. Returns 0 otherwise.

void
G3d_setNullValue (void *c, int nofElts, int type)

Fills the vector pointed to by *c* with *nofElts* NULL-values of *type*.

void
G3d_setNullTileType (void *map, int tile, int type)

Assumes that *tile* is a tile with the same dimensions as the tiles of *map*. Fills *tile* with NULL-values of *type*.

void
G3d_setNullTile (void *map, int tile)

Is equivalent to G3d_setNullTileType (map, tile, G3d_fileTypeMap (map)).

22.4.10 G3D Map Header Information

void
G3d_getCoordsMap (void *map, int *rows, int *cols, int *depths)

Returns the size of the region of *map* in cells.

void
G3d_getRegionMap (void *map, int *north, int *south, int *east, int *west, int *top, int *bottom)

Returns the size of the region.

void
G3d_getRegionStructMap (void *map, G3D_Region *region)

Returns in *region* the region of *map*.

void
G3d_getTileDimensionsMap (void *map, int *x, int *y, int *z)

Returns the tile dimensions used for *map*.

void
G3d_getNofTilesMap (void *map, int *nx, int *ny, int *nz)

Returns the dimensions of the tile-cube used to tile the region of *map*. These numbers include partial tiles.

int
G3d_tileTypeMap (void *map)

Returns the type in which tiles of *map* are stored in memory.

int
G3d_fileTypeMap (void *map)

Returns the type with which tiles of *map* are stored on file.

int
G3d_tilePrecisionMap (void *map)

Returns the precision used to store *map*.

int
G3d_tileUseCacheMap (void *map)

Returns 1 if *map* uses cache, returns 0 otherwise.

void
G3d_printHeader (void *map)

Prints the header information of *map*.

22.4.11 G3D Tile Math

void
G3d_tileIndex2tile (void *map, int tileIndex, int *xTile, int *yTile, int *zTile)

Converts index *tileIndex* into tile-coordinates *(xTile, yTile, zTile)*.

int
G3d_tile2tileIndex (void *map, int xTile, int yTile, int zTile)

Returns tile-index corresponding to tile-coordinates *(xTile, yTile, zTile)*.

void
G3d_coord2tileCoord (void *map, int x, int y, int z, int *xTile, int *yTile, int *zTile, int *xOffs, int *yOffs, int *zOffs)

Converts cell-coordinates *(x, y, z)* into tile-coordinates *(xTile, yTile, zTile)* and the coordinate of the cell *(xOffs, yOffs, zOffs)* within the tile.

void
G3d_tileCoordOrigin (void *map, int xTile, int yTile, int zTile, int *x, int *y, int *z)

Computes the cell-coordinates *(x, y, z)* which correspond to the origin of the tile with tile-coordinates *(xTile, yTile, zTile)*.

void
G3d_tileIndexOrigin (void *map, int tileIndex, int *x, int *y, int *z)

Computes the cell-coordinates *(x, y, z)* which correspond to the origin of the tile with *tileIndex*.

void
G3d_coord2tileIndex (void *map, int x, int y, int z, int *tileIndex, int *offset)

Converts cell-coordinates *(x, y, z)* into *tileIndex* and the *offset* of the cell within the tile.

int
G3d_coordInRange (void *map, int x, int y, int z)

Returns 1 if cell-coordinate *(x, y, z)* is a coordinate inside the region. Returns 0 otherwise.

int
G3d_tileInRange (void *map, int x, int y, int z)

Returns 1 if tile-coordinate *(x, y, z)* is a coordinate inside tile cube. Returns 0 otherwise.

int
G3d_tileIndexInRange (void *map, int tileIndex)

Returns 1 if *tileIndex* is a valid index for *map*. Returns 0 otherwise.

int
G3d_isValidLocation (void *map, double north, double west, double bottom)

Returns 1 if region-coordinates *(north, west, bottom)* are inside the region of *map*. Returns 0 otherwise.

void
G3d_location2coord (void *map, double north, double west, double bottom, int *x, *y, *z)

Converts region-coordinates *(north, west, bottom)* into cell-coordinates *(x, y, z)*.

int
G3d_computeClippedTileDimensions (void *map, int tileIndex, int *rows, int *cols, int *depths, int *xRedundant, int *yRedundant, int *zRedundant)

Computes the dimensions of the tile when clipped to fit the region of *map*. The clipped dimensions are returned in *rows*, *cols*, *depths*. The complement is returned in *xRedundant*, *yRedundant*, and *zRedundant*. This function returns the number of cells in the clipped tile.

22.4.12 G3D Range Support

The map structure of G3D provides storage for the range. The range of a map is updated every time a cell is written to the file. When an old map is opened the range is not automatically loaded. The application has to invoke `G3d_range_load ()` (cf. Section 22.4.12) explicitly. In addition to these function the application can also use the standard grass functions to manipulate the range.

int
G3d_range_load (void *map)

Loads the range into the range structure of *map*.
Returns 1 ... if successful 0 ... otherwise.

void
G3d_range_min_max (void *map, double *min, double *max)

Returns in *min* and *max* the minimum and maximum values of the range.

int
G3d_range_write (void *map)

Writes the range which is stored in the range structure of *map*. (This function is invoked automatically when a new file is closed).
Returns 1 ... if successful 0 ... otherwise.

22.4.13 G3D Color Support

Applications can use the standard grass functions to work with colors, except for the file manipulations.

int
G3d_removeColor (char *name)

Removes the primary and/or secondary color file. See *G_remove_colr* for details. Returns always 0.

int
G3d_readColors (char *name, char *mapset, struct Colors *colors)

Reads color file for map *name* in *mapset* into the *colors* structure. See *G_read_colors* (*12.10.3 Raster Color Table (p. 116)*) for details and return values.

int
G3d_writeColors (char *name, char *mapset, struct Colors *colors)

Writes colors stored in *colors* structure into the color file for map *name* in *mapset*. See *G_write_colors* (*12.10.3 Raster Color Table (p. 116)*) for details and return values.

22.4.14 G3D Categories Support

Applications can use the standard grass functions to work with categories, except for the file manipulations.

int
G3d_readCats (char *name, char *mapset, struct Categories *pcats

Reads the categories file for map *name* in *mapset* and stores the categories in the *pcats* structure. See *G_read_cats* (*12.10.2 Raster Category File (p. 114)*) for details and return values.

int
G3d_writeCats (char *name, struct Categories *cats)

Writes the categories stored in the *cats* structure into the categories file for map *name* in the current mapset. See *G_write_cats* (*12.10.2 Raster Category File (p. 114)*) for details and return values.

22.4.15 G3D Mask Support

void
G3d_maskOn (void *map)

Turns on the mask for *map*. Do not invoke this function after the first tile has been read since the result might be inconsistent cell-values.

void
G3d_maskOff (void *map)

Turns off the mask for *map*. This is the default. Do not invoke this function after the first tile has been read since the result might be inconsistent cell-values.

int
G3d_maskIsOn (void *map)

Returns 1 if the mask for *map* is turned on. Returns 0 otherwise.

int
G3d_maskIsOff (void *map)

Returns 1 if the mask for *map* is turned off. Returns 0 otherwise.

The remaining functions in this section are for the explicit query of the mask and the masking of individual cells or tiles. These functions are used in the library and might have applications in situations where both the masked and non-masked value of a cell has to be known.

int
G3d_maskReopen (int cache)

This function should be used to adjust the cache size used for the 3d-mask. First the open 3d-mask is closed and then opened again with a cache size as specified with *cache*.
Returns 1 ... if successful 0 ... otherwise.

int
G3d_maskFileExists ()

Returns 1 if the 3d mask file exists.

int
G3d_maskMapExists ()

Returns 1 if the 3d mask is loaded.

char *
G3d_maskFile ()

Returns the name of the 3d mask file.

int
G3d_isMasked (int x, int y, int z)

Returns 1 if the cell with cell-coordinates *(x, y, z)* is masked out. Returns 0 otherwise.

void
G3d_maskNum (int x, int y, int z, void *value, int type)

Replaces the value stored in *value* with the NULL-value if *G3d_isMasked (x, y, z)* returns 1. Does nothing otherwise. *value* is assumed to be of*type*.

void
G3d_maskFloat (int x, int y, int z, float *value)

Same as *G3d_maskNum (x, y, z, value, G3D_FLOAT)*.

void
G3d_maskDouble (int x, int y, int z, double *value)

Same as *G3d_maskNum (x, y, z, value, G3D_DOUBLE)*.

void
G3d_maskTile (void *map, int tileIndex, char *tile, int type)

Replaces the values stored in *tile* (with *tileIndex*) for which *G3d_isMasked* returns 1 with NULL-values. Does not change the remaining values. The values are assumed to be of *type*. Whether replacement is performed or not only depends on location of the cells of the tile and not on the status of the mask for *map* (i.e. turned on or off).

22.4.16 G3D Window Support

void
G3d_setWindowMap (void *map, G3D_Region *window)

Sets the window for *map* to *window*. Can be used multiple times for the same map.

void
G3d_setWindow (G3D_Region *window)

Sets the default window used for every map opened later in the program. Can be used multiple times in the same program.

void
G3d_getWindow (G3D_Region *window)

Stores the current default window in *window*.

void *
G3d_windowPtr ()

Returns a pointer to the current default window. This pointer should not be (ab)used to modify the current window structure directly. It is provided to pass a window pointer when opening a map.

int
G3d_readWindow (G3D_Region *window, char *windowName)

Reads *window* from the file specified by *windowName*. The name is converted by the rules defined in window defaults. A NULL pointer indicates the *WIND3* file in the current mapset.
Returns 1 ... if successful 0 ... otherwise.

int
G3d_writeWindow (G3D_Region *window, char *windowName)

Writes *window* to the file specified by *windowName*. The name is converted by the rules defined in window defaults. A NULL pointer indicates the *WIND3* file in the current mapset.
Returns 1 ... if successful 0 ... otherwise.

void
G3d_useWindowParams ()

Allows the window to be set at run-time via the *region3* command line argument. This function has to be called before *G_parser ()*. See also window defaults.

void
G3d_setResamplingFun (void *map, void (*resampleFun)())

Sets the resampling function to be used by G3d_getValue () (cf. Section 22.4.3). This function is defined as follows:

void
G3d_customResampleFun (void *map, int row, int col, int depth, char *value, int type)

row, *col*, and *depth* are in region coordinates. The result is returned in *value* as *type* which is one of G3D_FLOAT or G3D_DOUBLE. Possible choices include G3d_nearestNeighbor () (cf. Section 22.4.16) and G3d_getValueRegion () (cf. Section 22.4.3).

void
G3d_nearestNeighbor (void *map, int row, int col, int depth, char *value, int type)

The default resampling function which uses nearest neighbor resampling.

void
G3d_getResamplingFun (void *map, void (resampleFun) ())**

Returns in *resampleFun* a pointer to the resampling function used by *map*.

void
G3d_getNearestNeighborFunPtr (void (nnFunPtr) ())**

Returns in *nnFunPtr* a pointer to G3d_nearestNeighbor () (cf. Section 22.4.16).

22.4.17 G3D Region

void
G3d_extract2dRegion (G3D_Region *region3d, struct Cell_head *region2d)

Returns in *region2d* the *2d* portion of *region3d*.

void
G3d_incorporate2dRegion (struct Cell_head *region2d, G3D_Region *region3d)

Replaces the *2d* portion of *region3d* with the values stored in *region2d*.

void
G3d_adjustRegion (G3D_Region *region)

Computes an adjusts the resolutions in the region structure from the region boundaries and number of cells per dimension.

void
G3d_adjustRegionRes (G3D_Region *region)

Computes an adjusts the number of cells per dimension in the region structure from the region boundaries and resolutions.

void
G3d_regionCopy (G3D_Region *regionDest, G3D_Region *regionSrc)

Copies the values of *regionSrc* into *regionDst*. (The unfortunate order of parameters was chosen in order to conform to the order used in *G_copy ()*).

void
G3d_getRegionValue (void *map, double north, double east, double top, char *value, int type)

Returns in *value* the value of the *map* which corresponds to region coordinates *(north, east, top)*. The value is resampled using the resampling function specified for *map*. The *value* is of *type*.

void
G3d_readRegionMap (char *name, char *mapset, G3D_Region *region)

Returns in *region* the region information for 3d cell *name@mapset*.

22.4.18 Miscellaneous Functions

void
G3d_g3dType2cellType (int g3dType)

Returns the GRASS floating point type which is equivalent to the G3D type of *g3dType*.

void
G3d_initDefaults ()

Initializes the default values described in G3D Defaults. Applications have to use this function only if they need to query the default values before the first file (either old or new) has been opened.

void
G3d_setStandard3dInputParams ()

Initializes a parameter structure for the subset of command line arguments which lets the user overwrite the default properties of the new file. Applications are encouraged to use this function in order to provide a uniform style. The command line arguments provided are the *type* of the cell values, the *precision*, the properties of the *compression*, and the dimension of the tiles (*tiledimension*). Every of these values defaults to the value described in G3D Defaults.
This function has to be used in conjunction with G3d_getStandard3dInputParams () (cf. Section 22.4.18).

int
G3d_getStandard3dInputParams (int *useTypeDefault, *type, *useLzwDefault, *doLzw, int *useRleDefault, *doRle, *usePrecisionDefault, *precision, int *useDimensionDefault, *tileX, *tileY, *tileZ

Returns the properties of the new file as chosen by the user via command line arguments. If the default is chosen the values of *useXxxxDefault* is 1, it is 0 otherwise. In addition, the corresponding parameters contain the default value if *useXxxxDefault* is 1, or the value specified by the user if *useXxxxDefault* is 0.
Function G3d_setStandard3dInputParams () (cf. Section 22.4.18) has to be used to initialize the internal parameter structure.
Returns 1 ... if successful, 0 ... otherwise.

int
G3d_makeMapsetMapDirectory (char *mapName)

Creates the 3d mapset element for map *mapName*.

int
G3d_filename (char *path, *elementName, *mapName, *mapset)

Returns in *path* the path for element *elementName* for map *mapName* in *mapset*.
Note, an error occurs if *mapName* is fully qualified.

See *12.23 Timestamp functions (p. 214)* for a complete discussion of GRASS datetime routines (reading, writing grid3d timestamps).

22.5 Sample G3D Applications

These functions were implemented to test the library. They are not very efficient but can be used as starting point for other applications. Some of them might actually be useful. They are available from GRASS 5 source code in src/libes/g3d/.

void
G3d_retile (void *map, char *nameOut, int tileX, int tileY, int tileZ)

Makes a copy of *map* with name *nameOut* which has tile dimensions *tileX*, *tileY*, *tileZ*.
The source code can be found in *retile.c*.

void
G3d_changePrecision (void *map, int precision, char *nameOut)

Makes a copy of *map* with name *nameOut* which is written with *precision*.
The source code can be found in *changeprecision.c*.

void
G3d_changeType (void *map, char *nameOut)

Makes a copy of *map* with name *nameOut* in which the cells are of type G3D_FLOAT if they are G3D_DOUBLE in *map*, and in G3D_DOUBLE otherwise.
The source code can be found in *changetype.c*.

void
G3d_compareFiles (char *f1, char *mapset1, char *f2, char *mapset2)

Compares the cell-values of file *f1* in mapset *mapset1* and file *f2* in mapset *mapset2*. The values are compared up to precision. Terminates in error if the files don't match. This function uses the more advanced features of the cache.
The source code can be found in *filecompare.c*.

void
G3d_getBlock (void *map, int x0, int y0, int z0, int nx, int ny, int nz, char *block, int type

Copies the cells contained in the block (cube) with vertices *(x0, y0, z0)* and *(x0 + nx - 1, y0 + ny - 1, z0 + nz - 1)* into *block*. The cell-values in *block* are of *type*.
The source code can be found in *getblock.c*.

void
G3d_writeAscii (void *map, char *fname)

Writes the cell-values of *map* in ascii format to file *fname*. The values are organized by horizontal slices.

See **?? ??** *(p. ??)* for a G3D Function Index.

See *11 Compiling and Installing GRASS Modules (p. 69)* for a complete discussion of Gmakefiles.

23 DateTime Library

This chapter describes the new DateTime library contributed by GMSL.

Authors: Michael Shapiro & Bill Brown

23.1 Introduction

This Library may be used to record, manipulate, and perform arithmetic on date and time information. It is anticipated that **GRASS database access routines** will utilize this library to "timestamp" data files and perform temporal analysis. This library could also be used to generate, format and compare dates for labels, titles, or site descriptions. It is used in r.timestamp and v.timestamp.

23.1.1 Relative vs. Absolute

Successfully using this library requires understanding the two basic modes of DateTimes:

1) Absolute DateTimes express a single time or date referenced to the Gregorian calendar (e.g. 14 Feb 1995), and

2) Relative DateTimes express a difference or length of time (e.g., 201 days 6 hours).

An interval for a DateTime is defined by its greatest unit (from) and its smallest unit (to). The absolute DateTime "14 Feb 1995" has the interval: from=year, to=day. There are specific rules for legal intervals. The mode and interval define the "type" of a DateTime. When doing DateTime artithmetic certain type combinations are not allowed because the result would be undefined.

23.1.2 Calendar Assumptions

This library uses the modern Gregorian calendar, correcting for leap years using the convention: ((year%4 == 0 && year%100 != 0) || year%400 == 0) , but also extrapolating those leap years back in time. There are no leap second corrections and there is no correction for the missing 11 days of September 1752 (in some locales) or prior corrections (in other locales). The year is always considered to start on January 1 and end 365 or 366 days later on December 31.

Include File

#include <datetime.h>

DateTime Structure typedef struct {

int mode; /* absolute or relative */

int from, to; /* range of values */

int positive; /* positive/negative datetime */

int year, month, day;

int hour, minute;

double second;

int fracsec; /* #decimal place in printed seconds */

int tz; /* timezone - minutes from UTC */

} DateTime;

DateTimes have a 3-part type and range qualifiers from.

mode: one of

#define DATETIME_ABSOLUTE 1

#define DATETIME_RELATIVE 2

from, to: one of #define DATETIME_YEAR 1

#define DATETIME_MONTH 2

#define DATETIME_DAY 3

#define DATETIME_HOUR 4

#define DATETIME_MINUTE 5

#define DATETIME_SECOND 6

- The values for the from/to #defines must increase from YEAR to SECOND In other words YEAR < MONTH < DAY < HOUR < MINUTE < SECOND. The idea is that the higher elements represent higher precision for a date/time. For example, having seconds in the time is more precise than if seconds are not present.
- There are some restrictions on legal values for from/to:
  - from <= to

  - if the 'mode' is ABSOLUTE, then 'from' must be YEAR
  - if the 'mode' is RELATIVE, then

year, month, day, hour, minute, second:

- These are non-negative values.

- For ABSOLUTE types, these must be valid date/time values:
- year
- a complete year (not just the last 2 digits)
- must be positive (since 0 isn't a legal year).

- month[1,12]
- day[1,n] where n depends on the year/month.
- hour[0,23]]
- minute[0,59]
- second[0.0,<60.0]

- For RELATIVE types, the value corresponding to 'from' is unrestricted (except that it can't be negative). The other values are restricted as follows:

  if from==YEAR, month is [0,11]

  if from==DAY, hour is [0,23], min is [0,59], sec is [0.0,<60.0]

  if from==HOUR, min is [0,59], sec is [0.0,<60.0]

  if from==MINUTE, sec is [0.0,<60.0] fracsec:
- This controls the number of decimal places to print after the seconds.]
- It is only used if the 'to' element is SECOND.
- It must be non-negative. tz:
- The time (hour/minute) in ABSOLUTE types is in local time.
- The specification of a timezone (tz) is an (subtractive)] offset to convert from local time to UTC.
- To get UTC from localtime: LT - TZ
- tz is expressed in minutes from -720 to 780

  (720 == 12 hours, 780 minutes == 13 hours).

  [See ANSI X3.51-1975, section 2.2.3]

- For a timezone to be allowed, the 'to' field must be one of {MINUTE, SECOND} positive:
- this indicates if the datetime value is to considered "positive" (!=0) or "negative" (==0)
- For mode ABSOLUTE, positive==0 means BC

23.2 DateTime library functions

23.2.1 ASCII Representation

The ascii representation of DateTime is:

```
ABSOLUTE:  15 Jan 1994 [bc] 10:35:23.456 -0500

RELATIVE:  [-] 2 years 5 months

           [-] 100 days 15 hours 25 minutes 35.34 seconds

The parts can be missing.

ABSOLUTE:  1994 [bc]

           Jan 1994 [bc]

           15 jan 1000 [bc]

           15 jan 1994 [bc] 10 [+0000]

           15 jan 1994 [bc] 10:00 [+0100]

           15 jan 1994 [bc] 10:00:23.34 [-0500]

RELATIVE:  [-] 2 years

           [-] 5 months

           [-] 2 years 5 months

           [-] 100 days

           [-] 15 hours 25 minutes 35.34 seconds

           [-] 100 days 25 minutes

           [-] 1000 hours 35.34 seconds

                etc.

NOTE: values missing between the from/to are assumed

to be zero; when scanning, they can be missing; when
```

```
formatting they will appear as 0 (to preserve the from/to):

    1000 hours 0 minutes 35.34 seconds

   0 days 10 hours 0 minutes

 NOTE: when scanning the from/to are determined by the

fields present. Compare:

   10 hours 0 minutes 35.34 seconds [from=HOUR,to=SECOND]

and

   0 days 10 hours 0 minutes 35.34 seconds [from=DAY,to=SECOND]
```

int
datetime_scan (DateTime *dt, char *string)

Convert the ascii string into a DateTime. This determines the mode/from/to based on the string, inits 'dt' and then sets values in 'dt' based on the [???]
Returns 0 if 'string' is legal, -1 if not.

void
datetime_format (DateTime *dt, char *string)

Convert 'dt' to a printable string. 'string' should be large enough to hold the result, perhaps 80 bytes.

23.2.2 Initializing, Creating and Checking DateTime Structures

int
datetime_set_type (DateTime *dt; int mode, from, to, fracsec)

- This routine must be called can be made with other datetime functions.
- initialize all the elements in dt.
- Set all values to zero except:
  tz (set to illegal value - 99*24)
  positive (set to 1 for positive)
- Set the type info in dt: mode, from, to, fracsec

- validate the mode/from/to/fracsec (according to the rules for the mode)
- return the return value from `datetime_check_type`(dt)

void
datetime_get_type (DateTime *dt; int *mode, *from, *to, *fracsec)

extract the mode, from, to, and fracsec out of dt.

int
datetime_check_type (DateTime *dt)

checks the mode/from/to/fracsec in dt.
Returns:

- 0: OK
- -1: mode is invalid - not one of {ABSOLUTE,RELATIVE}
- -2: from is invalid - not one of {YEAR,MONTH,DAY,HOUR,MINUTE,SECOND}
- -3: to is invalid - not one of {YEAR,MONTH,DAY,HOUR,MINUTE,SECOND}
- -4: from/to are reversed (from>to is illegal)
- -5: invalid from/to combination for RELATIVE mode:
  from in {YEAR,MONTH} but to is not, or
  from in {DAY,HOUR,MINUTE,SECOND} but to is not
- -6: from is invalid for ABSOLUTE mode (from != YEAR is illegal)
- -7: fracsec is negative (only if to==SECOND)

int
datetime_is_valid_type (DateTime *dt)

Returns:
1 if **datetime_check_type()** returns 0
0 if not.

int
datetime_change_from_to (DateTime *dt; int from, to; int round)

Changes the from/to of the type for dt. The 'from/to' must be legal values for the mode of dt; (if they are not legal, then the original values are preserved, dt is not changed).

Returns:
0 OK
-1 invalid 'dt'
-2 invalid 'from/to'

- round =
  negative implies floor() [decrease magnitude]
  0 implies normal rounding, [incr/decr magnitude]
  positive implies ceil() [increase magnitude]
- If dt.from < 'from' (losing "lower" elements), convert the "lost" values to the equivalent value for the new 'from' Lost elements are then set to zero. (This case can only occur for dt.mode relative):
  months += lost years * 12 ; years = 0
  hours += lost days * 24 ; days = 0
  minutes += lost hours * 60 ; hours = 0
  seconds += lost minutes * 60.0 ; minutes = 0
- If dt.from > 'from' (adding "lower" elements), the new elements are set to zero.
- If dt.to < 'to' (adding "higher" elements), the new elements are set to zero.
- If dt.to > 'to' (losing "higher" elements), the the new 'to' is adjusted according to the value for 'round' After rounding the "lost" elements are set to zero.

int
datetime_is_absolute (DateTime *dt)

Returns:
1 if dt.mode is absolute
0 if not (even if dt.mode is not defined)

int
datetime_is_relative (DateTime *dt)

Returns:
1 if dt.mode is relative
0 if not (even if dt.mode is not defined)

void
datetime_copy (DateTime *dst, *src)

Copies the DateTime [into/from ???] src

int
datetime_is_same (DateTime *dt1, *dt2)

Returns:
1 if dt1 is exactly the same as dt2
0 if they differ

23.2.3 Getting & Setting Values from DateTime Structure

These routines get/set elements of 'dt'. They return:

- 0 if OK
- -1 if the value being gotten or set is not a legal value
- -2 if the from/to for 'dt' doesn't include this value Values don't get set if they are invalid.

int
datetime_set_year (DateTime *dt, int year)

if dt.mode = ABSOLUTE, this also sets dt.day = 0

int
datetime_get_year (DateTime *dt, int *year)

int
datetime_set_month (DateTime *dt, int month)

if dt.mode = ABSOLUTE, this also sets dt.day = 0

int
datetime_get_month (DateTime *dt, int *month)

int
datetime_set_day (DateTime *dt, int day)

if dt.mode = ABSOLUTE, then the dt.year, dt.month:

```
  if (day > datetime_days_in_month (dt.year,
dt.month)) {error}
```

This implies that year/month must be set for ABSOLUTE datetimes.

int
datetime_get_day (DateTime *dt, int *day)

int
datetime_set_hour (DateTime *dt, int hour)

int
datetime_get_hour (DateTime *dt, int *hour)

int
datetime_set_minute (DateTime *dt, int minute)

int
datetime_get_minute (DateTime *dt, int *minute)

int
datetime_set_second (DateTime *dt, double second)

int
datetime_get_second (DateTime *dt, double *second)

int
datetime_set_fracsec (DateTime *dt, int fracsec)

int
datetime_get_fracsec (DateTime *dt, int *fracsec)

int
datetime_check_year (DateTime *dt, int year)

Returns:
0 is legal year for dt

-1 illegal year for this dt
-2 dt has no year component

int
datetime_check_month (DateTime *dt, int month)

Returns:
0 is legal month for dt
-1 illegal month for this dt
-2 dt has no month component

int
datetime_check_day (DateTime *dt, int day)

Returns:
0 is legal day for dt
-1 illegal day for this dt
-2 dt has no day component

Note: if dt.mode is ABSOLUTE, then dt.year and dt.month must also be legal, since the 'day' must be a legal value for the dt.year/dt.month

int
datetime_check_hour (DateTime *dt, int hour)

int
datetime_check_minute (DateTime *dt, int minute)

int
datetime_check_second (DateTime *dt, double second)

int
datetime_check_fracsec (DateTime *dt, int fracsec)

23.2.4 DateTime Arithmetic

These functions perform addition/subtraction on datetimes.

int
datetime_increment (DateTime *src, *incr)

This function changes the 'src' date/time data based on the 'incr'
The type (mode/from/to) of the 'src' can be anything.
The mode of the 'incr' must be RELATIVE, and the type (mode/from/to) for 'incr' must be a valid increment for 'src'. See **datetime_is_valid_increment()**, **datetime_check_increment()**
Returns:
0: OK
-1: 'incr' is invalid increment for 'src'
For src.mode ABSOLUTE,

- positive 'incr' moves into the future,
- negative 'incr' moves into the past.
- BC implies the year is negative, but all else is positive. Also, year==0 is illegal: adding 1 year to 1[bc] gives 1[ad]

The 'fracsec' in 'src' is preserved.
The 'from/to' of the 'src' is preserved.
A timezone in 'src' is allowed - it's presence is ignored.
NOTE: There is no datetime_decrement() To decrement, set the 'incr' negative.

void
datetime_set_positive (DateTime *dt)

Makes the DateTime positive. (A.D. for ABSOLUTE DateTimes)

void
datetime_set_negative (DateTime *dt)

Makes the DateTime negative. (B.C. for ABSOLUTE DateTimes)

void
datetime_invert_sign (DateTime *dt)

int
datetime_is_positive (DateTime *dt)

Returns:
1 if the Datetime is positive
0 otherwise

int
datetime_difference (DateTime *a, *b, *result)

This performs the formula: result = a - b;

- both a and b must be absolute.
- result will be relative
- If a is "earlier" than b, then result will be set negative.
- b must be no more "precise" than a.
  (a copy of b is "extended" to the precision of a)
- If result.to == SECOND, then result.fracsec is a.fracsec
- result will have the following from/to based on a.to: result a.to from to YEAR YEAR YEAR MONTH YEAR MONTH DAY DAY DAY HOUR DAY HOUR MINUTE DAY MINUTE SECOND DAY SECOND [LAYOUT ??? - see HTML]
- If either 'a' or 'b' has a timezone, both must have a timezone. The difference will account for the differences in the time zones.

int
datetime_is_valid_increment (DateTime *src, *incr)

Returns:
datetime_check_increment(src, incr) == 0

int
datetime_check_increment (DateTime *src, *incr)

This checks if the type of 'incr' is valid for incrementing/decrementing 'src'.
The type (mode/from/to) of the 'src' can be anything.
The incr.mode must be RELATIVE
A timezone in 'src' is allowed - it's presence is ignored.
To aid in setting the 'incr' type, see **datetime_get_increment_type()**.
Returns:

- 0 valid increment
- 1 src is not a legal DateTime, error code/msg are those set by **datetime_is_valid_type()**
- 2 incr is not a legal DateTime, error code/msg are those set by **datetime_is_valid_type()**
- -1 incr.mode not relative
- -2 incr more precise that src

- -3 illegal incr, must be YEAR-MONTH
- -4 illegal incr, must be DAY-SECOND

int
datetime_get_increment_type (DateTime *src; int *mode, *from, *to, *fracsec)

This returns the components of a type (mode/from/to/fracsec) that can be used to construct a DateTime object that can be used to increment the 'src'. Also see **datetime_set_increment_type()**.
returns:
0 dt is legal
!=0 why dt is illegal
Implemented as follows:

```
*mode = RELATIVE
*to = src.to
*fracsec = src.fracsec
if src.mode is ABSOLUTE
if src.to is in {YEAR,MONTH} then
*from = YEAR
if src.to is in {DAY,HOUR,MINUTE,SECOND} then
*from = DAY
if src.mode is RELATIVE, then
*from = src.from
```

int
datetime_set_increment_type (DateTime *src, *incr)

src must be legal
This is a convenience routine which is implemented as follows:

```
int mode, from ,to;
int fracsec;
if(datetime_get_increment_type(src, &mode, &from,
&to, &fracsec))
return datetime_get_error_code();
return datetime_set_type (incr, mode, from, to,
fracsec);
```

Timezone Timezones are represented in minutes from GMT in the range [-720,+780]. For a DateTime to have a timezone, it must be of type ABSOLUTE, and "to" must be in {MINUTE,SECOND}.

The next 3 functions return:

0: OK

-1: mode not ABSOLUTE

-2: dt.to not in {MINUTE,SECOND}

-3: minutes not valid - not in the range [-720,+780]

int
datetime_check_timezone (DateTime *dt, int minutes)

int
datetime_set_timezone (DateTime *dt, int minutes)

int
datetime_get_timezone (DateTime *dt, int *minutes)

int
datetime_is_valid_timezone (int minutes)

Returns:
1 OK: -720 <= minutes <= 780 (720 = 12 hours; 780 = 13 hours)
0 NOT OK

void
datetime_unset_timezone (DateTime *dt)

Remove timezone from 'dt'
dt.tz = 99*60 (some illegal value)

int
datetime_change_timezone (DateTime *dt; int minutes)

if dt has a timezone, increment dt by minutes-dt.tz MINUTES and set dt.tz = minutes
Returns:
0 OK
datetime_check_timezone (dt) if not
-4 if minutes invalid

int
datetime_change_to_utc (DateTime *dt)

Return **datetime_change_timezone** (dt, 0);

void
datetime_decompose_timezone (int tz, int *hour, int *minute)

tz = abs(tz)
*hour = tz/60
*minute = tz%60
Note: hour,minute are non-negative. Must look at sign of tz itself to see if the tz is negative offset or not. This routine would be used to format tz for output. For example if tz=-350 this would be hour=5 minute=50, but negative. Output might encode this as -0550: printf ("%s%02d%02d", tz<0?"-":"", hour, minute)

int
datetime_get_local_timezone (int *minutes)

Returns:
0 OK
-1 local timezone info not available

void
datetime_get_local_time (DateTime *dt)

set mode/from/to ABSOLUTE/YEAR/SECOND
set the local time into 'dt'
does not set timezone.

23.2.5 Utilities

int
datetime_days_in_month (int month, year)

int
datetime_is_leap_year (int year, ad)

int
datetime_days_in_year (int year, ad)

23.2.6 Error Handling

All datetime functions that return int status codes should return:

- 0 (or positive) if OK;
- a negative integer if not; and register the error with a call to **datetime_error()**

Applications can test for error by:

```
if (datetime_function() < 0) {process the error}
```

int
datetime_error (int code, char *msg)

record 'code' and 'msg' as error code/msg (in static variables)
code==0 will clear the error (ie set msg=NULL)
returns 'code' so that it can be used like:

```
return datetime_error (-1, "bad date");
```

char *
datetime_get_error_msg ()

returns pointer to static error msg (which is NULL if no error)

int
datetime_get_error_code ()

returns error code

void
datetime_clear_error ()

clears error code and message

23.2.7 Example Application

Follow these links for the source code for mini applications called **incr** and **dsub**. The **incr** application simply adds or subtracts a legal increment and a datetime. The **dsub** application simply adds or subtracts two datetimes given on the command line. Once compiled with the datetime library, the following entries and results may be duplicated:

```
  incr "3 feb 1" - "40 days"  25 Dec 1 bc
  incr "31 Jan 1 bc" + "360 days"  26 Jan 1
  incr "1 year 11 months" - "2 years"  - 0 years 1 month
  incr "- 1 year 11 months" + "2 years"  0 years 1 month
  incr "- 1 year 11 months" + "36 months"  1 year 1 month
  incr "1 day 23 hours 40 minutes" - "2 days"  - 0 days 0 hours
20 minutes
  incr "2 days 0 minutes" + "1 day 23 hours 40 minutes"  3 days
23 hours 40 minutes
  incr "- 2 days 0 minutes" + "1 day 23 hours 40 minutes"  - 0
days 0 hours 20 minutes
  incr "- 2 days 0 minutes 5 seconds" + "3 days 23 hours 40 minutes"
1 day 23 hours 39 minutes 55 seconds
  incr "1 day 23 hours 39 minutes 55 seconds" + "2 days 0 minutes
5 seconds"  3 days 23 hours 40 minutes 0 seconds
  incr "28 feb 1980" + "1 day"  29 Feb 1980
  incr "1 mar 1979" - "1 day"  28 Feb 1979
  incr "1 mar 1979" + "365 days"  29 Feb 1980
  dsub "4 jul 1776 12:00:00.123" "4 jul 1976 11:00" - 73047 days
22 hours 59 minutes 59.877 seconds
  dsub "4 jul 1976 11:00" "4 jul 1776 12:00:00.123" 73047 days
23 hours 0 minutes
  dsub "4 jul 1976 11:00:00.00" "4 jul 1776 12:00:00.123" 73047
days 22 hours 59 minutes 59.88 seconds
  dsub "4 jul 1976 11:00:00.000" "4 jul 1776 12:00:00.123" 73047
days 22 hours 59 minutes 59.877 seconds
  incr "1 apr 1963 00:00:00.000" - "1440 minutes 1.23 seconds"
30 Mar 1963 23:59:58.770
  dsub "15 Jul 1995 11:53:17" "6 nov 1958 08:23:11" 13400 days
3 hours 30 minutes 6 seconds
  incr "15 Jul 1995 11:53:17" - "13400 days 3 hours 30 minutes
6 seconds" 6 Nov 1958 08:23:11
  incr "0 seconds" + "1 day"  86400 seconds
```

24 gsurf Library for OpenGL programming (ogsf)

Author: Bill Brown GMSL/University of Illinois

24.1 Overview

The gsurf library, consisting of approximately 20,000 lines of C code, contains some 180 public functions and about twice as many internal functions for run-time data storage, manipulation, querying, and visualization using OpenGL. The library handles all drawing and lighting operations, including use of user-defined clipping planes and drawing various style "fences" on clipping planes when drawing multiple surfaces, and treats datasets as objects which can be used for various attributes of the rendering. It allows data sharing (e.g., same data for more than one attribute of same or different surfaces), separate masking for each surface, multiple surfaces, vector sets, or point sets, and will also allow multiple volumes. The library provides all query features such as 3D "point on surface", keyframe animation routines, and full state saving functionality. Database-specific routines for interfacing with the GRASS GISlib are kept isolated for easier library reuse with other databases. The gsurf library is not dependent upon any particular interface library, and has been used successfully with both Motif and Tcl/Tk. It is used for NVIZ visualization tool.

The library is designed to provide a unique "handle" or identifier number to the calling program for each new geographic object added to the model. The object could be a surface, vector set, or point set, which could each be defined by one or more database files. Once created, the application only needs to keep track of the "handles" to the objects (which are returned by the creation routines); for example, to draw a surface the application would make the call:

GS_draw_surf(id)

where *id* is the handle number. To associate a vector set with a surface and then draw all surfaces with the vector set draped over the one selected, the application would use the calls:

GV_select_surf(vid, sid)

GS_alldraw_surf( )

GV_draw_vect(vid)

where *vid* and *sid* are the handles for the vector set and surface. Similarly, to query or change attributes of the object, the handle is used in conjunction with the new attribute, as in: GV_set_vectmode(id, mem, color, width)

24.2 Naming Conventions

The following naming conventions for function prefixes are used:

- **GS_** Functions which have to do with loading & manipulating surfaces. Also functions for library initialization, setting global variables, viewer positioning, and lighting.
- **GSU_** Utility functions for distance calculations, common 2D & 3D unit vector operations such as cross product or vector arithmetic.
- **GV_** Functions which have to do with loading & manipulating vector sets.
- **GP_** Functions which have to do with loading & manipulating point sets.
- **GVL_** Functions which have to do with loading & manipulating 3D volumes.
- **GK_** Functions which have to do with setting & manipulating keyframes for viewer position animation (fly-bys).

Programmers' documentation is currently incomplete, but see the following for more details of the library design and capabilities in the appendix:

- public function prototypes
- public include file gsurf.h
- public include file keyframe.h
- public color packing utility macros rgbpack.h
- private types and defines gstypes.h
- private utilities gsget.h

24.3 Public function prototypes

24.3.1 Function Prototypes for gsurf Library

void *GS_Get_ClientData (int id)

float GS_P2distance (float *from, float *to)

GS_Set_ClientData (int id, void *clientd)

GS_alldraw_cplane_fences ( )

GS_alldraw_surf( )

GS_alldraw_wire( )

GS_background_color( )

int GS_check_cancel( )

GS_clear (int col)

GS_default_draw_color( )

GS_delete_surface (int id)

GS_distance (float *from, *to)

GS_done_draw( )

int
GS_draw_X (int id, float *pt)

pt only has to have an X & Y value in true world coordinates

GS_draw_cplane (int num)

GS_draw_cplane_fence (int hs1, int hs2, int num)

GS_draw_lighting_model( )

GS_draw_line_onsurf (int id, float x1, float y1, float x2, float y2)

GS_draw_surf (int id)

GS_draw_wire (int id)

int
GS_dv3norm(double dv1[3])

Changes dv1 so that it is a unit vector

double

GS_geodistance (double *from, double *to, char *units)

distance between 2 coordinates

Returns distance between two geographic coordinates in current projection.

Units is one of: "meters", "miles", "kilometers", "feet", "yards", "nmiles" (nautical miles), "rods", "inches", "centimeters", "millimeters", "micron", "nanometers", "cubits", "hands", "furlongs", "chains".

Default is meters.

GS_get_SDscale (float *scale)

GS_get_SDsurf (int id)

double GS_get_aspect( )

GS_get_att (int id, int att, int *set, float *constant, char *mapname)

GS_get_cat_at_xy (int id, int att, char *catstr, float x, float y)

GS_get_dims (int id, int *rows, int *cols)

int
GS_get_distance_alongsurf (int hs, int use_exag, float x1, float y1, float x2, float y2, float *dist)

Returns distance following terrain.

GS_get_drawmode (int id, int mode)

GS_get_drawres (int id, int *xres, int *yres, int *xwire, int *ywire)

GS_get_exag_guess (int id, float *exag)

int GS_get_fencecolor( )

GS_get_focus (float *realto)

int
GS_get_fov( )

Returns field of view, in 10ths of degrees.

GS_get_from (float *fr)

GS_get_from_real (float *fr)

GS_get_longdim (float *dim)

GS_get_maskmode (int id, int mode)

int
GS_get_modelposition (float *siz, float pos[3])

Retrieves coordinates for lighting model position, at center of view, (nearclip * 2) from eye.

GS_get_nozero (int id, int att, int *mode)

GS_get_region (float *n, float *s, float *w, float *e)

GS_get_scale (float *sx, float *sy, float *sz, int doexag)

int
GS_get_selected_point_on_surface (int sx, int sy, int *id, float *x, float *y, float *z)

Given screen coordinates sx & sy, find closest intersection of view ray with surfaces and return coordinates of intersection in x, y, z, and identifier of surface in id.
Returns 0 if no intersections found, otherwise number of intersections.

GS_get_surf_list (int *numsurfs)

GS_get_to (float *to)

GS_get_trans (int id, float *xtrans, float *ytrans, float *ztrans)

int GS_get_twist( )

int
GS_get_val_at_xy (int id, char *att, char *valstr, float x, float y)

Prints "NULL" or the value (i.e., "921.5") to valstr.
Colors are translated to rgb and returned as Rxxx Gxxx Bxxx
Usually call after GS_get_selected_point_on_surface
Returns -1 if point outside of window or masked, otherwise 1

GS_get_viewdir (float dir[3])

GS_get_wire_color (int id, int *colr)

int
GS_get_zextents (int id, float *min, float *max, float *mid)

Returns Z extents for a single surface.

int
GS_get_zrange (float *min, float *max, int doexag)

Returns Z extents for all loaded surfaces.

int
GS_get_zrange_nz (float *min, float *max)

Returns Z extents for all loaded surfaces, treating zeros as "no data".

float GS_global_exag( )

int GS_has_transparency( )

GS_init_view( )

GS_is_masked (int id, float *pt)

GS_libinit( )

GS_lights_off( )

GS_lights_on( )

GS_load_3dview(char *vname, int surfid)

GS_load_att_map (int id, char *filename, int att)

int
GS_look_here (int sx, int sy)

Reset center of view to screen coordinates sx, sy.
Send screen coords sx & sy, lib traces through surfaces & sets new center to point of nearest intersection. If no intersection, uses line of sight with length of current view ray (eye to center) to set new center.

GS_moveto (float *pt)

GS_moveto_real (float *pt)

int GS_new_light( )

GS_new_surface( )

int GS_num_surfs( )

GS_ready_draw( )

GS_save_3dview(char *vname, int surfid)

GS_set_SDscale (float scale)

GS_set_SDsurf (int id)

GS_set_att_const (int id, int att, float constant)

GS_set_att_defaults (float defs[MAX_ATTS], null_defs[MAX_ATTS])

GS_set_cancel (int c)

GS_set_cplane (int num)

GS_set_cplane_rot (int num, float dx, float dy, float dz)

GS_set_cplane_trans (int num, float dx, float dy, float dz)

GS_set_cxl_func(void (*f)( ))

int
GS_set_draw (int where)

Sets which buffer to draw to.
where should be one of: GSD_BOTH, GSD_FRONT, GSD_BACK

GS_set_drawmode (int id, int mode)

GS_set_drawres (int id, int xres, int yres, int xwire, int ywire)

GS_set_exag (int id, float exag)

GS_set_fencecolor (int mode)

GS_set_focus (float *realto)

GS_set_focus_center_map (int id)

GS_set_fov (int fov)

GS_set_global_exag (float exag)

GS_set_maskmode (int id, int mode)

GS_set_nofocus( )

GS_set_nozero (int id, int att, int mode)

GS_set_swap_func(void (*f)( ))

GS_set_trans (int id, float xtrans, float ytrans, float ztrans)

int
GS_set_twist (int t)

t is tenths of degrees clockwise from 12:00.

GS_set_viewdir (float dir[3])

GS_set_viewport (int left, int right, int bottom, int top)

GS_set_wire_color (int id, int colr)

GS_setall_drawmode (int mode)

GS_setall_drawres (int xres, int yres, int xwire, int ywire)

int
GS_setlight_ambient (int num, float red, float green, float blue)

Red, green, blue from 0.0 to 1.0

int
GS_setlight_color (int num, float red, float green, float blue)

Red, green, blue from 0.0 to 1.0

GS_setlight_position (int num, float xpos, float ypos, float zpos, int local)

GS_surf_exists (int id)

GS_switchlight (int num, int on)

GS_transp_is_set( )

GS_unset_SDsurf( )

GS_unset_att (int id, int att)

GS_unset_cplane (int num)

GS_update_curmask (int id)

GS_update_normals (int id)

GS_v2dir (float v1[2], float v2[2], float v3[2])

GS_v3add (float v1[3], float v2[3])

int
GS_v3cross (float v1[3], float v2[3], float v3[3])

returns the cross product v3 = v1 cross v2

GS_v3dir (float v1[3], float v2[3], float v3[3)

GS_v3eq (float v1[3], float v2[3])

GS_v3mag (float v1[3], float *mag)

int
GS_v3mult (float v1[3], float k)

v1 *= k

int
GS_v3norm (float v1[3])

Changes v1 so that it is a unit vector

int
GS_v3normalize (float v1[3], float v2[3])

Changes v2 so that v1v2 is a unit vector

int
GS_v3sub (float v1[3], float v2[3])

v1 -= v2

void *GV_Get_ClientData (int id)

GV_Set_ClientData (int id, int clientd)

GV_alldraw_vect( )

GV_delete_vector (int id)

GV_draw_fastvect (int vid)

GV_draw_vect (int vid)

GV_get_trans (int id, float *xtrans, float *ytrans, float *ztrans)

int *
GV_get_vect_list (int *numvects)

USER must free when no longer needed!

GV_get_vectmode (int id, int *mem, int *color, int *width)

GV_get_vectname (int id, char *filename)

GV_load_vector (int id, char *filename)

int GV_new_vector( )

int GV_num_vects( )

int
GV_select_surf (int hv, int hs)

Select surface identified by hs to have vector identified by hv draped over it.

GV_set_trans (int id, float xtrans, float ytrans, float ztrans)

GV_set_vectmode (int id, int mem, int color, int width)

GV_surf_is_selected (int hv, int hs)

GV_unselect_surf (int hv, int hs)

GV_vect_exists (int id)

void *GP_Get_ClientData (int id)

GP_Set_ClientData (int id, void *clientd)

GP_alldraw_site( )

GP_attmode_color (int id, char *filename)

GP_attmode_none (int id)

GP_delete_site (int id)

GP_draw_site (int id)

int *
GP_get_site_list (int *numsites)

USER must free when no longer needed!

GP_get_sitemode (int id, int *atmod, int *color, int *width, float *size, int *marker)

GP_get_sitename (int id, char *filename)

GP_get_trans (int id, float *xtrans, float *ytrans, float *ztrans)

GP_get_zmode (int id, int *use_z)

GP_load_site (int id, char *filename)

int GP_new_site( )

int GP_num_sites( )

GP_select_surf (int hp, int hs)

GP_set_sitemode (int id, int atmod, int color, int width, float size, int marker)

GP_set_trans (int id, float *xtrans, float *ytrans, float *ztrans)

GP_set_zmode (int id, int use_z)

GP_site_exists (int id)

GP_surf_is_selected (int hp, int hs)

GP_unselect_surf (int hp, int hs)

int
GK_add_key (float pos, unsigned long fmask, int force_replace, float precis)

The pos value is the relative position in the animation for this particular keyframe - used to compare relative distance to neighboring keyframes, it can be any floating point value.
The fmask value can be any of the following or'd together: KF_FROMX_MASK KF_FROMY_MASK KF_FROMZ_MASK KF_FROM_MASK (KF_FROMX_MASK | KF_FROMY_MASK | KF_FROMZ_MASK)
KF_DIRX_MASK KF_DIRY_MASK KF_DIRZ_MASK KF_DIR_MASK (KF_DIRX_MASK | KF_DIRY_MASK | KF_DIRZ_MASK)
KF_FOV_MASK KF_TWIST_MASK
KF_ALL_MASK (KF_FROM_MASK | KF_DIR_MASK | KF_FOV_MASK | KF_TWIST_MASK)
Other fields will be added later.
The value precis and the boolean force_replace are used to determine if a keyframe should be considered to be at the same position as a pre-existing keyframe. e.g., if anykey.pos - newkey.pos <= precis, GK_add_key will fail unless force_replace is TRUE.
Returns 1 if key is added, otherwise -1.

void
GK_clear_keys( )

Deletes all keyframes, resets field masks. Doesn't change number of frames requested.

int
GK_delete_key (float pos, float precis, int justone)

The values pos & precis are used to determine which keyframes to delete. Any keyframes with their position within precis of pos will be deleted if justone is zero. If justone is non-zero, only the first (lowest pos) keyframe in the range will be deleted.
Returns number of keys deleted.

int
GK_do_framestep (int step, int render)

Moves the animation to frame number "step". Step should be a value between 1 and the number of frames. If render is non-zero, calls draw_all.

int
GK_move_key (float oldpos, float precis, float newpos)

Precis works as in other functions - to identify keyframe to move. Only the first keyframe in the precis range will be moved.
Returns number of keys moved (1 or 0).

int
GK_set_interpmode (int mode)

Sets interpolation mode to KF_LINEAR or KF_SPLINE

void
GK_set_numsteps (int newsteps)

Sets the number of frames to be interpolated from keyframes.

int
GK_set_tension (float tens)

Sets value for tension when interpmode is KF_SPLINE. The value tens should be between 0.0 & 1.0.

void
GK_show_path (int flag)

Draws the current path.

GK_show_site (int flag)

GK_show_vect (int flag)

void GK_showtension_start( )

void
GK_showtension_stop( )

Use GK_showtension_start/GK_update_tension/GK_showtension_stop to initialize & stop multi-view display of path when changing tension.

void
GK_update_frames( )

Recalculates path using the current number of frames requested. Call after changing number of frames or when Keyframes change.

void GK_update_tension( )

void *GVL_Get_ClientData (int id)

GVL_Set_ClientData (int id, void *clientd)

GVL_alldraw_vol( )

GVL_delete_volume (int id)

GVL_draw_fastvol (int vid)

GVL_draw_vol (int vid)

GVL_get_trans (int id, float *xtrans, float *ytrans, float *ztrans)

int *GVL_get_vol_list (int *numvols)

GVL_get_volmode (int id, int *viztype)

GVL_get_volname (int id, char *filename)

GVL_load_volume (int id, char *filename)

int GVL_new_vol( )

int GVL_num_vol( )

GVL_set_trans (int id, float xtrans, float ytrans, float ztrans)

GVL_set_volmode (int id, int viztype)

int GVL_vol_exists (int id)

24.3.2 Public include file gsurf.h

See src/libes/ogsf/gsurf.h

24.3.3 Public include file keyframe.h

See src/libes/ogsf/keyframe.h

24.3.4 Public color packing utility macros rgbpack.h

See src/libes/ogsf/rgbpack.h

24.3.5 Private types and defines gstypes.h

See src/libes/ogsf/gstypes.h

24.3.6 Private utilities gsget.h

See src/libes/ogsf/gsget.h

25 Numerical math interface to LAPACK/BLAS

Author:

David D. Gray

(under development)

This chapter provides an explanation of how numerical algebra routines from LAPACK/BLAS () can be accessed through the GRASS GIS library "‘gmath"’. Most of the functions are wrapper modules for linear algebra problems, a few are locally implemented.

Getting BLAS/LAPACK (one package):
http://www.netlib.org/lapack/
http://netlib.bell-labs.com/netlib/master/readme.html

Pre-compiled binaries of LAPACK/BLAS are provided on many Linux distributions.

25.1 Implementation

The function name convention is as follows:

1. G_matrix_*(): corresponding to Level3 BLAS (matrix-matrix),
2. G_matvect_*(): corresponding to Level2 BLAS (vector-matrix) and
3. G_vector_*(): corresponding to Level1 BLAS (vector-vector)

25.2 Matrix-Matrix functions

mat_struct *
G_matrix_init (int rows, int cols, int ldim) — *Initialise a matrix structure*

Initialise a matrix structure. Set the number of rows with the first and columns with the second parameter. The third parameter lead dimension (>= no. of rows) needs attention by the programmer as it is related to the Fortran workspace:
A 3x3 matrix would be stored as
[x x x _][x x x _][x x x _]

This work space corresponds to the sequence:
(1,1) (2,1) (3,1) unused (1,2) (2,2) ... and so on, ie. it is column major. So although the programmer uses the normal parameter sequence of (row, col) the entries traverse the data column by column instead of row by row. Each block in the workspace must be large enough (here 4) to hold a whole column (3), and trailing spaces are unused. The number of rows (ie. size of a column) is therefore not necessarily the same size as the block size allocated to hold them (called the "'lead dimension"'). Some operations may produce matrices a different size from the inputs, but still write to the samer workspace. This may seem awkward but it does make for efficient code. Unfortunately this creates a responsibility on the programmer to specify the lead dimension (>= no. of rows). In most cases the programmer can just use the rows however. So for 3 rows/2 cols it would be called:
G_matrix_init (3, 2, 3);

Set parameters for a matrix structure

mat_struct *
G_matrix_set(mat_struct *A, int rows, int cols, int ldim)

Set parameters for a matrix structure that is allocated but not yet initialised fully.
Note:

G_matrix_set() is an alternative to G_matrix_init(). G_matrix_init() initialises and returns a pointer to a dynamically allocated matrix structure (ie. in the process heap). G_matrix_set() sets the parameters for an already created matrix structure. In both cases the data workspace still has to be allocated dynamically.
Example 1:

```
mat_struct *A;
G_matrix_init (A, 4, 3, 4);
```

Example 2:

```
mat_struct A; /* Allocated on the local stack */
G_matrix_set (&A, 4, 3, 4);
```

Add two matrices

mat_struct *
G_matrix_add (mat_struct *mt1, mat_struct *mt2)

Add two matrices and return the result. The receiving structure should not be initialised, as the matrix is created by the routine.

mat_struct *
G_matrix_product (mat_struct *mt1, mat_struct *mt2)

Multiply two matrices

Multiply two matrices and return the result. The receiving structure should not be initialised, as the matrix is created by the routine.

mat_struct *
G_matrix_scale (mat_struct *mt1, const double c)

Scale matrix

Scale the matrix by a given scalar value and return the result. The receiving structure should not be initialised, as the matrix is created by the routine.

mat_struct *
G_matrix_subtract (mat_struct *mt1, mat_struct *mt2)

Subtract two matrices

Subtract two matrices and return the result. The receiving structure should not be initialised, as the matrix is created by the routine.

mat_struct *
G_matrix_copy (const mat_struct *A)

Copy a matrix

Copy a matrix by exactly duplicating its structure.

mat_struct *
G_matrix_transpose (mat_struct *mt)

Transpose a matrix

Transpose a matrix by creating a new one and populating with transposed elements.

void
G_matrix_print (mat_struct *mt)

Print out a matrix

Print out a representation of the matrix to standard output.

int
G_matrix_LU_solve (const mat_struct *mt1, mat_struct **xmat0, const mat_struct *bmat, mat_type mtype)

Solve a general system A.X=B

Solve a general system A.X=B, where A is a NxN matrix, X and B are NxC matrices, and we are to solve for C arrays in X given B. Uses LU decomposition.

Links to LAPACK function dgesv_() and similar to perform the core routine. (By default solves for a general non-symmetric matrix.)
mtype is a flag to indicate what kind of matrix (real/complex, Hermitian, symmetric, general etc.) is used (NONSYM, SYM, HERMITIAN).
Warning: NOT YET COMPLETE: only some solutions' options available. Now, only general real matrix is supported.

Matrix inversion using LU decomposition

mat_struct *
G_matrix_inverse (mat_struct *mt)

Calls G_matrix_LU_solve() to obtain matrix inverse using LU decomposition. Returns NULL on failure.

Free up allocated matrix

void
G_matrix_free (mat_struct *mt)

Free up allocated matrix.

Set the value of the (i,j)th element

int
G_matrix_set_element (mat_struct *mt, int rowval, int colval, double val)

Set the value of the (i,j)th element to a double value. Index values are C-like ie. zero-based. The row number is given first as is conventional. Returns -1 if the accessed cell is outside the bounds.

Retrieve value of the (i,j)th element

double
G_matrix_get_element (mat_struct *mt, int rowval, int colval)

Retrieve the value of the (i,j)th element to a double value. Index values are C-like ie. zero-based.
Note: Does currently not set an error flag for bounds checking.

25.3 Matrix-Vector functions

Retrieve a column of matrix

vec_struct *
G_matvect_get_column (mat_struct *mt, int col)

Retrieve a column of the matrix to a vector structure. The receiving structure should not be initialised, as the vector is created by the routine. Col 0 is the first column.

vec_struct *
G_matvect_get_row (mat_struct *mt, int row)

Retrieve a row of matrix

Retrieve a row of the matrix to a vector structure. The receiving structure should not be initialised, as the vector is created by the routine. Row 0 is the first number.

int
G_matvect_extract_vector (mat_struct *mt, vtype vt, int indx)

Convert matrix to vector

Convert the current matrix structure to a vector structure. The vtype is RVEC or CVEC which specifies a row vector or column vector. The indx indicates the row/column number (zero based).

int
G_matvect_retrieve_matrix (vec_struct *vc)

Revert a vector to matrix

Revert a vector structure to a matrix.

25.4 Vector-Vector functions

vec_struct *
G_vector_init (int cells, int ldim, vtype vt)

Initialise a vector structure

Initialise a vector structure. The vtype is RVEC or CVEC which specifies a row vector or column vector.

int
G_vector_set (vec_struct *A, int cells, int ldim, vtype vt, int vindx)

Set parameters for vector structure

Set parameters for a vector structure that is allocated but not yet initialised fully. The vtype is RVEC or CVEC which specifies a row vector or column vector.

Copy a vector

vec_struct *
G_vector_copy (const vec_struct *vc1, int comp_flag)

Copy a vector to a new vector structure. This preserves the underlying structure unless you specify DO_COMPACT comp_flag:
0 Eliminate unnecessary rows (cols) in matrix
1 ... or not

Calculates euclidean norm

double
G_vector_norm_euclid (vec_struct *vc)

Calculates the euclidean norm of a row or column vector, using BLAS routine dnrm2_()

Calculates maximum value

double
G_vector_norm_maxval (vec_struct *vc, int vflag)

Calculates the maximum value of a row or column vector. The vflag setting defines which value to be calculated:
vflag:
1 Indicates maximum value
-1 Indicates minimum value
0 Indicates absolute value [???]

Note: More functions and wrappers will be implemented soon (11/2000).

25.5 Notes

The numbers of rows and columns is stored in the matrix structure:

```
printf("    M1 rows: %d, M1 cols: %d\n", m1->rows, m1->cols);
```

Draft Notes:

* G_vector_free() can be done by G_matrix_free().

```
#define G_vector_free(x) G_matrix_free( (x) )
```

* Ax calculations can be done with G_matrix_multiply()

* Vector print can be done by G_matrix_print().

```
#define G_vector_print(x) G_matrix_print( (x) )
```

25.6 Example

The Gmakefile needs a reference to $(GMATHLIB) in LIBES line.

Example Gmakefile:

```
PGM=mytest
HOME=.

LIBES=$(GMATHLIB) $(GISLIB) $(BLASLIB) $(LAPACKLIB) -lblas -llapack -lg2c
LIST = main.o

$(HOME)/$(PGM): $(LIST) $(LIBES)
        $(CC) $(LDFLAGS) -o $@ $(LIST) $(LIBES) $(MATHLIB) $(XDRLIB)

$(LIBES): #
```

Example module:

```
#include "la.h"
#include "gis.h"

int main(int argc, char *argv[])
{
  mat_struct *matrix;
  double val;

  /* init a 3x2 matrix */
  matrix=G_matrix_init (3, 2, 3);

  /* set cell element 0,0 in matrix to 2.2: */
  G_matrix_set_element (matrix, 0, 0, 2.2);

  /* retrieve this value */
  val = G_matrix_get_element (matrix, 0, 0);

  /*print it */
  fprintf(stderr, "Value: %g\n", val);

  /* Free the memory */
  G_matrix_free (matrix);

}
```

26 GUI programming: Graphical user interfaces

26.1 TclTkGRASS

Author:
Jacques Bouchard, based on L.A.S. initial developments
various contributors

TclTkGRASS is an tcl/tk based GUI (graphical user interface) for GRASS. It relies on the modular concept of GRASS.

26.1.1 TclTkGRASS Programming

TclTkGRASS is highly customizable. Module windows can be defined, the menu structure completely modified. The programming of new modules within the TCLTKGRASS environment can be done as follows:

1. Adding a module window into TCLTKGRASS:

   To add a module into the TCLTKGRASS windows environment you have to edit tcltkgrass/main/menu.tcl

   The structure of the menu file is as follows:

   a) Menu with submenu:

   ```
   <Menuentry> {
     "<Submenu entry 1> {
          "source $env(TCLTKGRASSBASE)/module/<module.definitionfile>"
          }
     "<Submenu entry 2> {
          "source $env(TCLTKGRASSBASE)/module/<module.definitionfile>"
          }
   }
   ```

   b) Simple Submenu:

   ```
    "<Submenu entry 1> {
          "source $env(TCLTKGRASSBASE)/module/<module.definitionfile>"
          }
   ```

   c) Direct module call (if the module has to used interactively):

   ```
    "<Submenu entry 1> {
          "exec xterm -exec <grassmodule>"
          }
   ```

d) Example:

```
"Misc tools" {
       "Convert raster to lines from a thinned raster" {
           "source $env(TCLTKGRASSBASE)/module/r.line"
       }
       "Vector digitizer" {
           "exec xterm -exec v.digit"
       }
```

2. Programming the module window itself (the module.definitionfile)

a) Programming of non-interactive module windows

Here a listing of the entries follows (the braces are important, but do not insert "<" and ">" !). See an example below.

```
First entry:
  interface_build {
Second entry:
  {<grass module name>}
Third entry (interactive flag):
      1 <if data must be input interactively from terminal (xterm),>
<or>  0 <...else (for setting paramters through module menu)>
Fourth entry:
  {<Comment to be displayed in first module windows line>.}
Fifth to xxx line: query for variables:
  {entry <module variable> {<Comment for this variable>:} 0 <button>}
Fifth to xxx line: checkboxes for options:
  {checkbox <module option> {<Comment for this option>.} "" <module option>}
Last line:
  }
```

Module variables have to be specified for input and output files. Module options allow for example to run this module quietly, or output special information in a tcltkgrass window etc. The "interactive flag" indicates if the module will be directed through the window entries or through a xterm (see description and example below). You get all required information about a specific GRASS module from the GRASS man pages, if you want to define the module yourself.

The <button> may be:

```
raster:     query GRASS raster map
+raster:    query several GRASS raster maps (for multiple input separated
              with comma
vector:     query GRASS vector map
+vector:    query several GRASS vector maps (for multiple input separated
              with comma
sites:      query GRASS site file
+sites:     query several GRASS sites maps (for multiple input separated
              with comma
file:       choose file for reading from user's home directory
File:       choose file for writing from user's home directory
xy:         pick x,y coordinates on the active monitor window
xyz.<map>:  pick x,y coordinates on the active monitor window
               + z value for the raster map whose name is in variable <map>
               (compare d.3d)
"":         no query button
arc:        choose ARC/INFO file in <location>/<mapset>/arc
```

```
area:       choose area unit
color:      choose a color from a list (for display commands)
Color:      choose a color from a list including color "none" in the list
3Dcolor:    choose a color from a list including color "color" in the list
               (only used in d.3d)
distance:   choose distance unit (km, m, etc.)
dlg:        choose dlg file
dlg_ascii:  choose dlg ascii file
font:       choose font
group:      choose image group
subgroup:   choose image subgroup
signature:  choose signature file from subgroup
icon:       choose paint icon file
label:      choose paint label file
paint:      choose painter device
monitor:    choose monitor
region:     choose region definition file
spheroid:   choose spheroid
```

...some more feature: see script/gui.tcl for details.

To create fields in the module window, you have three options:

```
 - "entry": This is an empty line

    Generally:  {entry parameter {description:} 0 button}
                                  (you may specify "" instead of button)
    Example:    {entry input {Input site map:} 0 sites}

 - "checkbox": Use this clickable box for flags
    Generally:  {checkbox flag {description} "" flag}
    Example:    {checkbox -h {Display reference information.} "" -h}

 - "scale": This displays a numbered adjust bar
    Generally:  {scale parameter {description} min max interval}
    Example:    {scale size {Neighborhood size:} 1 25 2}
```

The easiest way is to develop new module windows from existing definitions.

Example:

```
interface_build {
    {s.surf.tps} 0
    {Interpolates and computes topographic analysis from site map using spline with tensio
    {entry input {Input site map:} 0 sites}
    {entry elev {Output elevation raster map:} 0 raster}
    {entry slope {Output slope raster map (optional):} 0 raster}
    {entry aspect {Output aspect raster map (optional):} 0 raster}
    {entry pcurv {Output profile curvature raster map (optional):} 0 raster}
    {entry tcurv {Output tangential curvature raster map (optional):} 0 raster}
    {entry mcurv {Output mean curvature raster map (optional):} 0 raster}
    {entry maskmap {Use this existing raster file name as a mask (optional):} 0 raster}
    {entry dmin {Minimum distance between points (default: 0.5 grid cell):} 0 ""}
    {entry zmult {Multiplier for z-value in site map (default: 1):} 0 ""}
    {entry tension {Tension parameter (appropriate for smooth surfaces) (default: 40):} 0
    {entry smooth {Smoothing parameter (default: 0 = no smoothing:} 0 ""}
    {entry segmax {Maximum number of points per segment (default: 40):} 0 ""}
    {entry npmin {Minimum number of points for interpolation (default: 150, see man page)
    {checkbox -h {Display reference information.} "" -h}
}
```

b) Speciality: The GRASS module shall be used interactively from xterm

This will be achieved through "1" in the second line of the module definition.

Example:

```
interface_build {
    {r.mapcalc} 1
    {Raster map layer data calculator.}
}
```

c) Speciality: Define standard options for GRASS modules

Some modules need options which the user should not be able to change. You can enter a command name with several words in the second line of the module.definitionfile.

```
Example: v.support
interface_build {
    {v.support option=build} 0
    {(Re)Builds topology of vector file.}
    {entry map {Name of the GRASS vector file to be (re)build:} 0 vector}
    {entry threshold {Snap threshold value (valid only with -s option):} 0 ""}
    {checkbox -s {Snap nodes.} "" -s}
    {checkbox -r {Set map region from data.} "" -r}
}
```

The easiest way is to copy existing module.definitionfile and change this copy to your purposes. Some more internal details are stored in the comments of script/gui.tcl.

26.2 XML/Python

Author:
Jan-Oliver Wagner

27 Digitizer/Mouse/Trackball Files (.dgt)

The following is derived from the manual for Line Trace Plus (LTPlus) by John Dabritz and the Forest Service. The code for the digitizer drivers was taken from LTPlus and modified. The 'additions' file describes what has been changed from the original LTPlus version. Note that LTPlus supports mice and trackballs as well as digitizers. These can be ignored for v.digit, and herein, "digitizer" will be used to correspond to digitizers, mice, and trackballs.

27.1 Rules for Digitizer Configuration Files

The following are rules and restrictions for creating .dgt files.

1. No line may exceed 95 characters in length.

2. In a line, all characters following (and including) a pound sign (#) are considered comments (ignored). To put a pound sign into a string not to be ignored, use a \035. Any ascii character can be specified in this way: a backslash followed by a 3-digit (ascii decimal) number specifying the ascii decimal value of the character.

3. All other non-blank characters must be within brackets { } OR be one of the following (which are followed by brackets):

setup

startrun

startpoint

startquery

stop

query

format

These represent the groups of information used to initiate, gather, and stop input from a graphics input device (digitizer, mouse, track-ball ect.). Only one (left or right) bracket may be on a single line, although text and brackets may share a line. See *27.2 Digitizer Configuration File Commands (p. 396)*

4. Limits:

a) The file can have no more than 100 non-blank, non-comment lines.

b) Other limits are listed with their data type, below.

5. The legal lines within brackets depend on the group to which the brackets belong. ALL DATA LINES ARE DEPENDENT ON THE PARTICULAR DEVICE. YOU MUST REFER TO THE TECHNICAL REFERENCE MANUAL FOR THE PARTICULAR DEVICE (mouse/digitizer/trackball) in order to determine which parameters and which values need to be used. The groups (setup, startrun, startpoint, startquery, stop, query , format) may be in any order. Within the groups: startrun, startpoint, startquery, query, and stop the order of command lines is important. These are the legal line formats for each grouping:

27.2 Digitizer Configuration File Commands

The following is an in-depth description of each command available in the .dgt digitizer files.

27.2.1 Setup

This data is used to setup the communication link with the digitizer and is used during interpretation of the digitizer data.

27.2.1.1 Serial Line Characteristics

baud = n This line is optional, default = 9600 if not specified. If specified, n must be one of: 300, 600, 1200, 1800, 2400, 4800 9600, or 19200.

parity = str str must be ”odd”, ”even”, or ”none”. This item is optional, and defaults to none if not specified.

data_bits = n The number of data bits used (does NOT include parity bits, if any). Choices = 5,6,7,8 (default = 8)

stop_bits = n The number of stop bits used on the serial line. Choices are 1, or 2. Optional, default = 1.

buttons = n Number of buttons on digitizer cursor. This entry lets v.digit know if digitizer keys are available for input. Default is 0, so an entry must be made if the digitizer cursor is to be used for input. If the value of buttons is less than 5, keyboard keys will also be used for input.

buttonstart = n Number of the first key on the digitizer cursor. Usually 0 or 1. Default is 0. This is strictly for comunicating with the user. If you have arrow keys on your puck, you can set buttonstart to whatever you want.

buttonoffset = n Difference between 1 and the value sent by the lowest digitizer button. In other words, if the digitizer keys sent the values 0, 1, ..., n, buttonoffset would equal one, if the button output already starts with one, buttonoffset would be zero (the default value). Although these are the two most common cases, it is legal for buttonoffset to be any integer value. For instance if your keys for some reason output the values 16-32, it would be legal to use the value -15 as the buttonoffset.

footswitch = 0 or 1 Does the digitizer have a footswitch? Zero for no, one for yes.

digname = string Name of the digitizer.

description = string One line description of digitizer, format, etc.

button_up_char = c Character that indicates that no button is pressed. Only appropriate if format is ascii and includes a button press byte.

27.2.1.2 Data Interpretation Characteristics

debounce = d [r] These values control the delay and repeat rate for a digitizer or mouse button that is held down (who says you can't hold a good button down!) The first value (delay) specifies the number of continuous reports with the same button press which may be received before it is taken as a second button press. The second value, separated by a space, is the repeat rate, which specifies the number of continuous reports between further reports received which will be taken as subsequent button presses. The second value (repeat rate) is optional (default is 1/3 of the first value). A O for the first value indicates an infinite delay. For this value indicates an indefinite delay. For this value, only 1 key press will be taken no matter how long a button is held down. If no debounce values are listed, the default of 0s will be used.

units_per_inch = n Helps to set sensitivity (on absolute type devices see next item below) & map-inch size. dflt=1000. Not used for relative type devices (mice), see below.

coordinates = str str must be 'absolute' or 'relative', dflt=absolute. In general, mouse/trackball devices are relative, and digitizers coordinates are absolute.

sign_type = aaa This indicates the sign type for binary formats: none (all +) (default for absolute crds). 0negative (o=neg, used for some abs coords). 1negative (1=neg, used for some abs coords). 2s-complement (default for relative coords).

Note: for binary formats the sign bit should be coded as highest bit number for a coordinate.

Note: for ascii formats, minus (-) sign is expected from the raw device to indicate a negative number.

x_positive = dd This indicates the direction of x-positive coordinates. dd is a sting which may have the value right or left. The default is right. All digitizers and mice have x-positive to the right as of this writing.

y_positive = dd This indicates the direction of y-positive coordinates. dd is a string which may have the value up or down. The default is up. The microsoft mouse is a digitizing device which has y-positive coordinates to indicate a downward movement.

digcursor = fname Specifies the cursor file to be used while this digitizer is in use with LTPlus program. The digcursor file defines which command each digitizer button generates. v.digit does not need a cursor file, and ignores this line.

Note: The order of items is unimportant within the setup group.

27.2.1.3 Example of a Setup

setup

{

digname = Calcomp

description = Calcomp digitizer, ascii format 12

buttons = 16 # number of buttons on digitizer

buttonstart = 0 # number buttons start with

buttonoffset = 1 # offset to get buttons 1-15

baud = 9600

units_per_inch = 1000

}

27.2.2 Startrun, Startpoint, Startquery, Stop, Query

All of these allow the same operations, but are used at different times when communicating with the digitizer/mouse. The START groupings are used to initialize the digitizer each time communication is switched to that mode. The QUERY grouping is used when (and if) the digitizer is queried/prompted to send data information. The STOP grouping is used to stop digitizer output. All of these groupings are optional but at least one start group must be included (to use the file with v.digit, the startquery group must be included). If the digitizer is configured by default or switch settings to output data in the desired form of a certain mode, it is desirable to include that start group anyway, with some innocuous action (such as sending a carriage return) as the only action. If a start group is not included for a given mode, the module assumes that the digitizer is unable to operate in that mode.

There may be no more than 40 operations within each start group or the stop group. There may be no more than 10 operations in the query group.

27.2.2.1 Operations

send = aaaa This allows the sending of any ascii string to the digitizer (at the current baud rate and parity).

read = n This tells the module to read n bytes from the digitizer before trying to read again (gives up trying to read after 1 second). This is for reading digitizer prompts during start & stop groups and is NOT used for querying the digitizer, unless a non-data string is to be read (like a prompt character).

wait = n wait n seconds (decimal seconds allowed) before next

communication with the digitizer. Many computers are quicker than digitizers and need to allow time for the digitizer to change baud rate before resuming communication. Maximum resolution for wait is 0.001 second.

baud = n This allows changing of baud rate which was set during setup and is normally not used otherwise. If only 1 baud rate is used, then it is put in the setup group only. This is the normal case for most digitizers.

27.2.2.2 Notes

Control, extention, space, and all other characters can be specified in sent strings by using the backslash followed by the ascii decimal value to be sent (up to 3 digits). Example: send=/027 (indicates the escape character).

The lines/commands communicating with the digitizer will be executed in the SAME ORDER as they are in the start/stop/query grouping. Order is very important. Wait commands may be necessary to give the digitizer time to execute the command sent. Wait commands may need to be added/changed when the main modules is run on a faster cpu (in order to give the digitizer enough time to keep up). A maximum of 40 non-comment lines can be in a start, stop, or query group. All characters to be sent must be specified, including carriage return (\013) and linefeed (\010).

Each time a QUERY group is executed, a 0.001 second wait is done automatically after all query group commands. This allows time for the graphics input device to send a packet of information before the serail line is read by the module.

v.digit requires that a STARTQUERY group exists.

27.2.2.3 Example of Start Groupings

startrun

```
{
send = \027%R
baud = 2400
wait = 0.6
read = 3
wait = 0.1
send = \027%S
}
startpoint
{
send = \027%Ĺ12\013 # set output format to format 12
send = \027%P\013 # set to run mode
}
startquery
{
send = \027%Ĺ12\013 # set output format to format 12
send = \027%R\013 # set to run mode
send = \027%Q!\013 # set prompt character to '!' and
# put in prompt mode
}
```

27.2.2.4 Example of a Query Grouping

```
query
{
send = !\013 # send prompt
}
```

27.2.2.5 Example of a Stop Grouping

stop

{send = \027%K

wait = 0.1

send = \027%*

}

27.2.3 Format

This data is used each time a packet of information from the digitizer is interpreted. This group must be one of 2 types; ascii or binary. The digitizer file MUST contain a format group (either ascii or binary).

Ascii format groups have only 1 line:

ascii = format_string

Binary format groups have one line for each byte in the form:

byteN = format_string Where N is the byte number, (1 or greater) or byte No = format_string (similar to above for OPTIONAL bytes). **Note.** The module assumes the optional bytes containing ONLY button press information (no x or y information).

The legal format strings depend on the type (ascii or byteN).

27.2.3.1 ASCII format strings

ASCII format strings have these characteristics:

1. There are no imbedded blanks.

2. Legal characters are:

x denotes 1 character of the x-coordinate value (sign included).

y denotes 1 character of the y-coordinate value (sign included).

b denotes 1 character of button information.

p denotes 1 character of button press information (up or down).

, denotes the comma character (used to sync data if present).

c denotes a carriage return (optionally specified)

l denotes a line-feed (optionally specified)

? denotes any other character of information (including blanks).

27.2.3.2 Notes

The sign (+ or -) should be coded as part of the x or y value. The specifications of the carriage-return and linefeed are totally optional. They will be ignored whether they are specified or not. Their only use is to separate one ascii grouping of incoming data from another. Any combination of carriage-returns and/or linefeeds will serve this purpose in any case os ascii format use.

27.2.3.3 Example of ASCII Format Grouping

format

{

ascii =?xxxxx,yyyyy,??bcl

}

27.2.3.4 Binary Format String

Binary format strings have these characteristics.

0. byteNo form is used only for bytes which are sometimes, but not always sent by the digitizing devices. These byte(s) must be at the end of the grouping/packet. For example, the Logitech Mouseman sends an optional 4^{th} bytes only when the middle button is pressed. Very few digitizing devices use optional bytes.

1. 8 bits are specified with at least 1 blank between bit groupings, even f fewer bits are used. Fill the left (high) bits with ? if necessary.

2. Legal characters are:

xN denotes bit N of the x-coordinate value (low-order bit is 0, maximum bit allowed is 30) (include sign bit as highest bit used)

yN denotes bit N of the y-coordinate value (low-order bit is 0, maximum bit allowed is 30) (include sign bit as highest bit used)

bN denotes bit N of button press value (low-order bit is 0, maximum bit allowed is 7).

p denotes button press bit (will be 1 if button is pressed, 0 otherwise).

0 denotes bit is always zero (used for sync bit).

1 denotes bit is always one (used for sync bit).

? denoted any other information (bit not used).

27.2.3.5 Notes

There cannot be more than 100 lines of byten = in the format group.

Sign bits (if any) should be coded as the highest bit number for a given coordinate. Parity bits (if in the lowest 8 bits), and fill bits (if fewer than 8 bits used) should be coded as ?. No bits above the lowest 8 should be specified ar all (sometimes there is a 9^{th} parity bit).

0s and 1s are used for syncing the input, and should all occur in the same bit column.

27.2.3.6 Examples of a Binary Format Grouping

Example with odd or even parity and 7 data bits.

format

{

byte1 = ? 1 ? ? ? ? ? ?

byte2 = ? 0 ? b4 b3 b2 b1 b0

byte3 = ? 0 x5 x4 x3 x2 x2 x0

byte4 = ? 0 x11 x10 x9 x8 x7 x6

byte5 = ? 0 x16 x17 x15 x14 x13 x12

byte6 = ? 0 y5 y4 y3 y2 y1 y0

byte7 = ? 0 y11 y10 y9 y8 y7 y6

byte8 = ? 0 y16 y17 y15 y14 y13 y12

}

or

Example with 8 data bits (with or without parity.)

format

{

byte1 = 1 p b3 b2 b1 b0 x15 x14

byte2 = 0 x13 x12 x11 x10 x9 x8 x7

byte3 = 0 x6 x5 x4 x3 x2 x1 x0

byte4 = 0 ? ? ? x16 y16 y15 y14

byte5 = 0 y13 y12 y11 y10 y9 y8 y7

byte6 = 0 y6 y5 y4 y3 y2 y1 y0

}

27.3 Examples of Complete Files

The following are complete examples of digitizer files.

27.3.1 Example 1

setup

{

digname = Calcomp

description = Calcomp digitizer, ascii format 5

buttonoffset = 1

buttons = 16

buttonstart = 0

baud = 9600

units_per_inch = 1000

}

```
startrun
{
send = \027%Ĺ5\013 # set to format 5
send = \027%R\013
}
startpoint
{
send = \027%Ĺ5\013 # set to format 5
send = \027%P\013
}
startquery
{
send = \027%Ĺ5\013 # set to format 5
send = \027%R\013
send = \027%Q!\013
}
query
{
send = !\013
}
stop
{
send = \027%H\013
}
format
{
ascii = xxxxx,yyyyy,??b
```

}

27.3.2 Example 2

setup

{

digname = Altek

description = altek digitizer, model AC30, binary output format 8

buttonoffset = 1 # button output starts at 0, we want 1

buttonstart = 0 # first button is numbered 0

buttons = 16 # number of buttons is 16

baud = 9600

parity = none

stop_bits = 1

sign_type = none

units_per_inch = 1000

coordinates = absolute

sign_type = none

}

startrun

{

send=S2\13 # set to run mode

send=F8\13 # set output format to 8

send=R6\13 # enter rate mode 6

}

startpoint

{

send = P\013 # set to point mode

```
send = F8\013 # set output format to 8
}
startquery
{
send = S2\013 # altek has no specific prompt mode, but may be
# queried at any time, so set to run mode
send = F8\013 # set output format to 8
}
query
{
send = V\013 # request data
}
stop
{
send = \027\013 # reset
}
format
{
byte1 = 1 p b3 b2 b1 b0 x15 x14
byte2 = 0 x13 x12 x11 x10 x9 x8 x7
byte3 = 0 x6 x5 x4 x3 x2 x1 x0
byte4 = 0 ? ? ? x16 y16 y15 y14
byte5 = 0 y13 y12 y11 y10 y9 y8 y7
byte6 = 0 y6 y5 y4 y3 y2 y1 y0
}
```

27.4 Digitizer File Naming Conventions

The naming conventions for digitizers driver files is:

manufacturer name or abbreviation + model number of digitizer + output format the digitizer is using + _ + number of keys on puck

For example, an Altek model 30 digitizer using format 8 with a 16 button puck would be:

al + 30 + f8 + _ + 16

Put it together and you have –> al30f8_16

You can optionally stick a .dgt extention on the end of the file name, e.g., al30f8_16.dgt This is by no means required, but its a clear indicator as to the use of the digitizer file which helps everyone in the long run. Test your files thoroughly. When it works, tell other users about your file. This helps everyone by reducing duplication of effort.

28 Writing a Graphics Driver

28.1 Introduction

GRASS application modules which use graphics are written with the *Raster Graphics Library* . At compilation time, no actual graphics device driver code is loaded. It is only at run-time that the graphics requests make their way to device-specific code. At run-time, an application module connects with a running graphics *device driver*, typically via system level first-in-first-out (fifo) files. Each GRASS site may have one or more of these modules to choose from. They are managed by the module *d.mon.*

Porting GRASS graphics modules from device to device simply requires the creation of a new graphics driver module. Once completed and working, all GRASS graphics modules will work exactly as they were designed without modification (or recompilation). This section is concerned with the creation of a new graphics driver.

28.2 Basics

The various drivers have source code contained under the directory $GISBASE/src/D/devices.[1] This directory contains a separate directory for each driver, e.g., XDRIVER, SUNVIEW and MASS. In addition, the directory *lib* contains files of code which are shared by the drivers. The directory GENERIC contains the beginnings of the required subroutines and sample *Gmakefile.*

A new driver must provide code for this basic set of routines. Once working, the programmer can choose to rewrite some of the generic code to increase the performance of the new driver. Presented first below are the required routines. Suggested options for driver enhancement are then described.

28.3 Basic Routines

Described here are the basic routines required for constructing a new GRASS graphics driver. These routines are all found in the GENERIC directory. It is suggested that the programmer create a new directory (e.g., MYDRIVER) into which all of the GENERIC files are copied (i.e., cp GENERIC/* MYDRIVER).

[1] $GISBASE is the directory where GRASS is installed. See *10.1 UNIX Environment (p. 65)* for details.

28.3.1 Open/Close Device

Graph_Set () *initialize graphics*

This routine is called at the start-up of a driver. Any code necessary to establish the desired graphics environment is included here. Often this means clearing the graphics screen, establishing connection with a mouse or pointer, setting drawing parameters, and establishing the dimensions of the drawing screen. In addition, the global integer variables SCREEN_LEFT, SCREEN_RIGHT, SCREEN_TOP, SCREEN_BOTTOM, and NCOLORS must be set. Note that the GRASS software presumes the origin to be in the upper left-hand corner of the screen, meaning:

SCREEN_LEFT < SCREEN_RIGHT

SCREEN_TOP < SCREEN_BOTTOM

You may need to flip the coordinate system in your device-specific code to support a device which uses the lower left corner as the origin. These values must map precisely to the screen rows and columns. For example, if the device provides graphics access to pixel columns 2 through 1023, then these values are assigned to SCREEN_LEFT and SCREEN_RIGHT, respectively.

NCOLORS is set to the total number of colors available on the device. This most certainly needs to be more than 100 (or so).

Graph_Close () *shut down device*

Close down the graphics processing. This gets called only at driver termination time.

28.3.2 Return Edge and Color Values

The four raster edge values set in the *Graph_Set( )* routine above are retrieved with the following routines.

Screen_left (index) *return left pixel column value*

Screen_rite (index) *return right pixel column value*

Screen_top (index) *return top pixel row value*

Screen_bot (index) *return bottom pixel row value*

int *index ;

The requested pixel value is returned in **index.**

These next two routines return the number of colors. There is no good reason for both routines to exist; chalk it up to the power of anachronism.

Get_num_colors (index) *return number of colors*

int *index ;

The number of colors is returned in **index.**

get_num_colors () *return number of colors*

The number of colors is returned directly.

28.3.3 Drawing Routines

The lowest level drawing routines are draw_line( ), which draws a line between two screen coordinates, and Polygon_abs( ) which fills a polygon.

draw_line (x1,y1,x2,y2) *draw a line*

int x1, y1, x2, y2 ;

This routine will draw a line in the current color from **x1,y1** to **x2,y2.**

Polygon_abs (x,y,n) **draw filled polygon**

int *x, *y ;

int n ;

Using the **n** screen coordinate pairs represented by the values in the **x** and **y** arrays, this routine draws a polygon filled with the currently selected color.

28.3.4 Colors

This first routine identifies whether the device allows the run-time setting of device color look-up tables. If it can (and it should), the next two routines set and select colors.

Can_do ()*signals run-time color look-up table access*

If color look-up table modification is allowed, then this routine must return 1; otherwise it returns 0. If your device has fixed colors, you must modify the routines in the *lib* directory which set and select colors. Most devices now allow the setting of the color look-up table.

reset_color (number, red, green, blue) *set a color*

it number

unsigned char red, green, blue ;

The system's color represented by **number** is set using the color component intensities found in the **red, green**, and **blue** variables. A value of 0 represents 0% intensity; a value of 255 represents 100% intensity. **color** (number) select a color int number ;

The current color is set to **number.** This number points to the color combination defined in the last call to *reset_color( )* that referenced this number.

28.3.5 Mouse Input

The user provides input through the three following routines.

Get_location_with_box (cx,cy,wx,wy,button) *get location with rubber box*

int cx, cy ;

int *wx, *wy ;

int *button ;

Using mouse device, get a new screen coordinate and button number. Button numbers must be the following values which correspond to the following software meanings:

1 - left button

2 - middle button

3 - right button

A rubber-band box is used. One corner is fixed at the **cx,cy** coordinate. The opposite coordinate starts out at **wx,wy** and then tracks the mouse. Upon button depression, the current coordinate is returned in **wx,wy** and the button pressed is returned in **button.**

Get_location_with_line (cx,cy,wx,wy,button) *get location with rubber line*

int cx, cy ;

int *wx, *wy ;

int *button ;

Using mouse device, get a new screen coordinate and button number. Button numbers must be the following values which correspond to the following software meanings:

1 - left button

2 - middle button

3 - right button

A rubber-band line is used. One end is fixed at the **cx,cy** coordinate. The opposite coordinate starts out at **wx,wy** and then tracks the mouse. Upon button depression, the current coordinate is returned in **wx,wy** and the button pressed is returned in **button.**

Get_location_with_pointer (wx,wy,button) *get location with pointer*

int *wx, *wy ;

int *button ;

Using mouse device, get a new screen coordinate and button number. Button numbers must be the following values which correspond to the following software meanings:

1 - left button

2 - middle button

3 - right button

A cursor is used which starts out at **wx,wy** and then tracks the mouse. Upon button depression, the current coordinate is returned in **wx,wy** and the button pressed is returned in **button.**

28.3.6 Panels

The following routines cooperate to save and restore sections of the display screen.

Panel_save (name, top, bottom, left, right) *save a panel*

char *name ;

int top, bottom, left, right ;

The bit display between the rows and cols represented by **top, bottom, left**, and **right** are saved. The string pointed to by **name** is a file name which may be used to save the image.

Panel_restore (name) *restore a panel*

char *name ;

Place a panel saved in **name** (which is often a file) back on the screen as it was when it was saved. The memory or file associated with **name** is removed.

28.4 Optional Routines

All of the above must be created for any new driver. The GRASS *Rasterlib*, which provides the application module routines which are passed to the driver via the fifo files, contains many more graphics options. There are actually about 44. Above, we have described 19 routines, some of which do not have a counterpart in the *Rasterlib.* For GRASS 3.0, the basic driver library was expanded to accommodate all of the graphics subroutines which could be accomplished at a device-dependent level using the 19 routines described above. This makes driver writing quite easy and straightforward. A price that is paid is that the resulting driver is probably slower and less efficient than it might be if more of the routines were written in a device-dependent way. This section presents a few of the primary target routines that you would most likely consider rewriting for a new driver.

It is suggested that the driver writer copy entire files from the lib area that contain code which shall be replaced. In the loading of libraries during the compilation process, the entire file containing an as yet undefined routine will be loaded. For example, say a file "*ab.c*" contains subroutines a( ) and b( ). Even if the programmer has provided subroutine a( ) elsewhere, at load time, the entire file "*ab.c*" will be loaded to get subroutine b( ). The compiler will likely complain about a multiply defined external. To avoid this situation, do not break routines out of their files for modification; modify the entire file.

Raster_int (n, nrows, array, withzeros, type) *raster display*

int n ;

int nrows ;

unsigned int *array ;

int withzeros ;

int type ;

This is the basic routine for rendering raster images on the screen. Application modules construct images row by row, sending the completed rasters to the device driver. The default *Raster_int( )* in lib draws the raster through repetitive calls to *color( )* and *draw_line( ).* Often a 20x increase in rendering speed is accomplished through low-level raster calls. The raster is found in the **array** pointer. It contains color information for **n** colors and should be repeated for **nrows** rows. Each successive row falls under the previous row. (Depending on the complexity of the raster and the number of rows, it is sometimes advantageous to render the raster through low-level box commands.) The **withzeros** flag indicates whether the zero values should be treated as color 0 (withzeros= =1) or as invisible (withzeros= =0). Finally, **type** indicates that the raster values are already indexed to the hardware color look-up table (type= =0), or that the raster values are indexed to GRASS colors (which must be translated through a look-up table) to hardware look-up table colors (type= =1).

Further details on this routine and related routines *Raster_chr( ),* and *Raster_def( )* are, of course, found in the definitive documentation: the source code.

29 Writing a Paint Driver

29.1 Introduction

The *paint* system, which produces hardcopy maps for GRASS, is able to support many different types of color printers. This is achieved by placing all device-dependent code in a separate module called a device driver. Application programs, written using a library of device-independent routines, communicate with the device driver using the UNIX pipe mechanism. The device driver translates the device-independent requests into graphics for the device.

A *paint* driver has two parts: a shell script and an executable module. The executable module is responsible for translating device-independent requests into graphics on the printer. The shell script is responsible for setting some UNIX environment variables that are required by the interface, and then running the executable module. The user first selects a printer using the *p.select* module. The selected printer is stored in the GRASS environment variable PAINTER.[1] Then the user runs one of the application programs. The principal *paint* applications that produce color output are *p.map* which generates scaled maps, and *p.chart* which produces a chart of printer colors. The application looks up the PAINTER and runs the related shell script as a child process. The shell script sets the required environment variables and runs the executable. The application then communicates with the driver via pipes.

29.2 Creating a Source Directory for the Driver Code

The source code for *paint* drivers lives in

$GISBASE/src/paint/Drivers [2]

Each driver has its own subdirectory containing the source code for the executable program, the shell script, and a *Gmakefile* with rules that tell the GRASS *gmake* command how to compile the driver.[3]

[1] See *10.2 GRASS Environment (p. 66)*.
[2] $GISBASE is the directory where GRASS is installed. See *10.1 UNIX Environment (p. 65)* for details.
[3] See *11 Compiling and Installing GRASS Modules (p. 69)* for details on the GRASS compilation process.

29.3 The Paint Driver Executable Program

A *paint* device driver module consists of a set of routines (defined below) that perform the device-dependent functions. These routines must be written for each device to be supported.

29.3.1 Printer I/O Routines

The following routines open the printer port and perform low-level i/o to the printer.

Popen (port) *open the printer port*

char *port;

Open the printer **port** for output. If the **port** is a *tty*, perform any necessary *tty* settings (baud rate, xon/xoff, etc.) required. No data should be written to the **port.**

The **port** will be the value of the UNIX environment variable MAPLP,[4] if set, and NULL otherwise. It is recommended that device drivers use the **port** that is passed to them so that *paint* has a consistent logic.

The baud rate should not be hardcoded into *Popen* ( ). It should be set in the driver shell as the UNIX environment variable BAUD. *Popen* ( ) should determine the baud rate from this environment variable.

Pout (buf, n) *write to printer unsigned*

char *buf;

int n;

Output the data in **buf.** The number of bytes to send is **n.** This is a low-level request. No processing of the data is to be done. Output is simply to be sent as is to the printer.

It is not required that data passed to this routine go immediately to the printer. This routine can buffer the output, if desired.

It is recommended that this routine be used to send all output to the printer.

Poutc (c) *write a character to printer*

unsigned char c;

Sends the character **c** to the printer. This routine can be implemented as follows:

Poutcl' unsigned char c;

[4]This, and other, environment variables are set in the driver shell script which is described in *29.4 The Device Driver Shell Script (p. 421).*

```
{

Pout(c, 1);

}
```

Pouts (s) *write a string to printer unsigned*

char *s;

Sends the character string **s** to the printer. This routine can be implemented as follows:

```
Pouts(s) unsigned char *s;

{

Pout(s, strlen(s));

}
```

Pflush () *flush pending output*

Flush any pending output to the printer. Does not close the port.

Pclose ()*close the printer port*

Flushes any pending output to the printer and closes the port.

Note. The above routines are usually not device dependent. In most cases the printer is connected either to a serial *tty* port or to a parallel port. The *paint* driver library[5] contains versions of these routines which can be used for output to either serial or parallel ports. Exceptions to this are the *preview* driver, which sends its output to the graphics monitor, and the *NULL* driver which sends debug output to stderr.

29.3.2 Initialization

The following routine will be called after *Popen* to initialize the printer:

Pinit ()*initialize the printer*

Initializes the printer. Sends whatever codes are necessary to get the printer ready for printing.

[5] See *29.6 Paint Driver Library (p. 424)*.

29.3.3 Alpha-Numeric Mode

The following two routines allow the printer to be used for normal text printing:

Palpha ()*place printer in text mode*

Places the printer in alpha-numeric mode. In this mode, the driver should only hono*r Ptext* calls.

Ptext (text) *print text*

char *text;

Prints the **text** string on the printer.

The **text** will not normally have nonprinting characters (i.e., control codes, tabs, linefeeds, returns, etc.) in it. Such characters in the **text** should be ignored or suppressed if they do occur. If the printer requires any linefeeds or carriage returns, this routine should supply them.

Note. If the printer does not have support for text in the hardware, it must be simulated. The *shinko635* printer does not have text, and the code from that driver can be used.

29.3.4 Graphics Mode

The following routines perform raster color graphics:

Praster ()*place printer in graphics mode*

Places the printer in raster graphics mode. This implies that subsequent requests will be related to generating color images on the printer.

Pnpixels(nrows, ncols) *report printer dimensions*

int *nrows;

int *ncols;

The variable **ncols** should be set to the number of pixels across the printer page. If the driver is combining physical pixels into larger groupings (e.g., 2*x*2 pixels) to create more colors, then **ncols** should be set to the number of these larger pixels.

The variable **nrows** should be set to 0. A non-zero value means that the output media does not support arbitrarily long output and *p.map* will scale the output to fit into a window **nrows** *x* **ncols.** The only driver which should set this to a non-zero value is the *preview* driver, which sends its output to the graphics screen.

Ppictsize (nrows, ncols) *defined picture size*

int nrows;

int ncols;

Prepare the printer for a picture with **nrows** and **ncols.** The number of columns **ncols** will not exceed the number of columns returned by *Pnpixels.*[6]

There is no limit on the number of rows **nrows** that will be requested. *p.map* assumes that the printer paper is essentially infinite in length. Some printers (e.g., thermal printers like the *shinko635*) only allow a limited number of rows, after which they leave a gap before the output can begin again. It is up to the driver to handle this. The output will simply have gaps in it. The user will cut out the gaps and tape the pieces back together.

Pdata (buf, n) *send raster data to printer unsigned*

char *buf;

int n;

Output the raster data in **buf.** The number of bytes to send is **n**, which will be the *ncols* as specified in the previous call to *Ppictsize.* The values in **buf** will be printer color numbers, one per pixel.

Note that the color numbers in **buf** have full color information encoded into them (i.e., red, green, and blue). Some printers (e.g., inkjet) can output all the colors on a row by row basis. Others (e.g., thermal) must lay down a full page of one color, then repeat with another color, etc. Drivers for these printers will have to capture the raster data into temporary files and then make three passes through the captured data, one for each color.

Prle(buf, n) *send rle raster data to printer*

unsigned char *buf;

int n;

Output the run-length encoded raster data in **buf.** The data is in pairs: *color, count* , where *color* is the raster color to be sent, and *count* is the number of times the *color* is to be repeated (with a *count* of 0 meaning 256). The number of pairs is **n.** Of course, all the counts should add up to *ncols* as specified in the previous call to *Ppictsize.* If the printer can handle run-length encoded data, then the data can be sent either directly or with minimal manipulation. Otherwise, it must be converted into standard raster form before sending it to the printer.

[6]The programmer should, of course, code defensively. If the number of columns is too large, the driver should exit with an error message.

29.3.5 Color Information

The *paint* system expects that the printer has a predefined color table. No attempt is made by *paint* to download a specific color table. Rather, the driver is queried about its available colors. The following routines return information about the colors available on the printer. These routines may be called even if *Popen* has not been called.

Pncolors ()*number of printer colors*

This routine returns the number of colors available. Currently, this routine must not return a number larger than 255. If the printer is able to generate more than 255 colors, the driver must find a way to select a subset of these colors. Also, the *paint* system works well with printers that have around 125 different colors. If the printer only has three colors (e.g., cyan, yellow, and magenta), then 125 colors can be created using a 2*x*2 pixel.[7]

Pcolorlevels (red, green, blue) *get color levels*

int *red, *green, *blue;

Returns the number of colors levels. This means, for example, if the printer has 125 colors, the color level would be 5 for each color; if the printer has 216 colors, the color levels would be 6 for each color, etc.

Pcolornum(red, green, blue) *get color number*

float red, green, blue;

This routine returns the color number for the printer which most closely approximates the color specified by the **red, green**, and **blue** intensities. These intensities will be in the range 0.0 to 1.0. [8]

The printer color numbers must be in the range 0 to *n* -1, where *n* is the number of colors returned by *Pncolors.*

For printers that have cyan, yellow, and magenta instead of red, green and blue, the conversion formulas are:

cyan = 1.0 - red

yellow = 1.0 - blue

magenta = 1.0 - green

Pcolorvalue (n, red, green, blue) *get color intensities*

[7] See *29.8 Creating 125 Colors From 3 Colors (p. 426)*.

[8] Just to be safe, those above 1.0 can be changed to 1.0, and those below 0.0 can be changed to 0.0.

int n;

float *red, *green, *blue;

This routine computes the **red, green**, and **blue** intensities for the printer color number **n.** These intensities must be in the range 0.0 to 1.0. If **n** is not a valid color number, set the intensities to 1.0 (white).

29.4 The Device Driver Shell Script

The driver shell is a small shell script which sets some environment variables, and then executes the driver. The following variables must be set :[9]

MAPLP

This variable should be set to the *tty* port that the printer is on. The *tty* named by this variable is passed to *Popen.* Only in very special cases can drivers justify either ignoring this value or allowing it not to be set.

The drivers distributed by GRASS Development Team have MAPLP set to `/dev/${PAINTER}`. Thus each driver must have a corresponding /dev port. These are normally created as links to real /dev/tty ports.

BAUD

This specifies the baud rate of the output *tty* port. This variable is only needed if the output port is a serial RS-232 tty port. The value of the variable should be an integer (e.g., 1200, 9600, etc.), and should be used by *Popen* to set the baud rate of the *tty* port.

HRES

This specifies the horizontal resolution of the printer in pixels per inch. This is a positive floating point number.

VRES

This specifies the vertical resolution of the printer in pixels per inch. This is a positive floating point number.

NCHARS

This specifies the maximum number of characters that can be printed on one line in alpha-numeric mode.

[9] The driver shell script may set any other variables that the programmer has determined the driver needs.

Note. The application modules do not try to deduce the width in pixels of text characters.

TEXTSCALE

This positive floating point number is used by *p.map* to set the size of the numbers placed on the grid when maps are drawn. The normal value is 1.0, but if the numbers should appear too large, a smaller value (0.75) will shrink these numbers. If they appear too small, a larger value (1.25) will enlarge them. This value must be determined by trial and error.

The next five variables are used to control the color boxes drawn in the map legend for *p.map* as well as the boxes for the printer color chart created by *p.chart.* They hav e to be determined by trial and error in order to get the numbering to appear under the correct box.[10]

NBLOCKS

This positive integer specifies the maximum number of blocks that are to be drawn per line.

BLOCKSIZE

This positive integer specifies the number of pixels across the top of an individual box.

BLOCKSPACE

This positive integer specifies the number of pixels between boxes.

TEXTSPACE

This positive integer specifies the number of space characters to output after each number (printed under the boxes).

TEXTFUDGE

This nonnegative integer provides a way of inserting extra pixels between every other box, or every third box, etc. On some printers, this will not be necessary, in which case TEXTFUDGE should be set to 0. If you find that the numbers under the boxes are drifting away from the intended box, the solution may be to move every other box, or every third box over 1 pixel. For example, to move every other box, set TEXTFUDGE to 2.

The following is a sample *paint* driver shell script:

```
: ${PAINTER?} ${PAINT_DRIVER?}

MAPLP=/dev/$PAINTER
```

[10] Apologies are offered for this admittedly awkward design.

```
BAUD=9600

HRES=85.8

VRES=87.0

NCHARS=132

TEXTSCALE=1.0

NBLOCKS=25

BLOCKSIZE=23

BLOCKSPACE=13

TEXTSPACE=1

TEXTFUDGE=3

export MAPLP BAUD HRES VRES NCHARS

export TEXTSCALE TEXTSPACE TEXTFUDGE

export NBLOCKS BLOCKSIZE BLOCKSPACE

exec $PAINT_DRIVER
```

29.5 Programming Considerations

The *paint* driver uses its standard input and standard output to communicate with the *paint* application module. It is very important that neither the driver shell nor the driver module write to stdout or read from stdin.

Diagnostics, error messages, etc., should be written to stderr. There is an error routine which driver modules can use for fatal error messages. It is defined as follows:

error (message, perror)

char *message;

int perror;

This routine prints the **message** on stderr. If **perror** is true (i.e., non-zero), the UNIX routine *perror* ( ) will be also called to print a system error message. Finally, *exit* ( ) is called to terminate the driver.

29.6 Paint Driver Library

The *paint* system comes with some code that has already been written. This code is in object files under the *paint* driver library directory.[11] These object files are:

main.o

This file contains the *main* ( ) routine **which must be loaded by every driver**, since it contains the code that interfaces with the application programs.

io.o

This file contains versions of Popen, Pout, Poutc, Pout, Pflush, and *Pclos* which can be used with printers that are connected to serial or parallel ports. These routines handle the tricky *tty* interfaces for both System V and Berkeley UNIX, allowing full 8-bit data output to the printer, with xon/xoff control enabled, as well as baud rate selection.

colors125.o

This file contains versions of *Pncolors*, *Pcolorlevels*, *Pcolornum, and Pcolorvalue for the 125 color logic described in 24.8 Creating 125 Colors From 3 Colors.*

29.7 Compiling the Driver

Paint drivers are compiled using the GRASS *gmake* utility which requires a *Gmakefile* containing compilation rules.[12] The following is a sample *Gmakefile*:

```
NAME = sample

DRIVERLIB = $(SRC)/paint/Interface/driverlib

INTERFACE = $(DRIVERLIB)/main.o \

$(DRIVERLIB)/io.o \

$(DRIVERLIB)/colors125.o

DRIVER_SHELL = $(ETC)/paint/driver.sh/$(NAME)

DRIVER_EXEC = $(ETC)/paint/driver/$(NAME)


OBJ = alpha.o text.o raster.o npixels.o \
```

[11] See *29.7 Compiling the Driver (p. 424)* for an example of how to load this library code.
[12] See *11 Compiling and Installing GRASS Modules (p. 69)* for details on the GRASS compilation process.

pictsize.o data.o rle.o

all: $(DRIVER_EXEC) $(DRIVER_SHELL)

$(DRIVER_EXEC): $(OBJ) $(LOCKLIB)

$(CC) $(LDFLAGS) $(INTERFACE) $(OBJ) $(LOCKLIB) -o $@

$(DRIVER_SHELL): DRIVER.sh

rm -f $@

cp $? $@

chmod +x $@

$(OBJ): P.h

$(LOCKLIB): # in case library changes

There are some features about this *Gmakefile* that should be noted:

printer name (NAME)

The printer name *sample* is assigned to the NAME variable, which is then used everywhere else.

paint driver library (DRIVERLIB)

This driver loads code from the common *paint* driver library.[13] It loads *main.o* containing the *main* ( ) routine for the driver. **All drivers must load** *main.o* .It *loads io.o which contains versions of Popen, Pout, Poute, Pouts*, *Pflush*, and *Pclose* for serial and parallel ports. It also *loads colors125.o which contains versions of Pncolors, Pcolorlevels, Pcolornum, and Pcolorvalue for 125 colors.*

lock library (LOCKLIB)

The driver loads the lock library. This is a GRASS library which must be loaded if the *Popen* from the driver library is used.

homes for driver shell and executable

The driver executable is compiled into the *driver* directory, and the driver shell is copied into the *driver.sh* directory. This means that the driver executable is placed in

$GISBASE/etc/paint/driver[14]

[13] See also *29.6 Paint Driver Library (p. 424)*.
[14] $GISBASE is the directory where GRASS is installed. See *10.1 UNIX Environment (p. 65)* for details.

and the driver shell in

$GISBASE/etc/paint/driver.sh.

29.8 Creating 125 Colors From 3 Colors

The *paint* system expects that the printer will have a reasonably large number of colors. Some printers support a large color table in the hardware. But others only support three primary colors: red, green, and blue (or cyan, yellow, and magenta). If the printer only has three colors, the driver must simulate more.

If the printer pixels are grouped into 2*x*2 combinations of pixels, then 125 colors can be simulated. For example, a color with 20% red, 100% green, and 0% blue would have one of the four pixels painted red, all four pixels painted green, and none of the pixels painted blue.

The following code converts a color intensity in the range 0.0 to 1.0 into a number from 0-4 (i.e., the number of pixels to ”turn on” for that color):

npixels = (intensity * 5);

if (npixels > 4)

npixels = 4 ;

This logic will agree with the 125 color logic used by the *paint* driver library[15] routines *Pncolors, Pcolorlevels, Pcolornum*, and *Pcolorvalue,* provided that the color *numbers* are assigned as follows:

color_number = red_pixels * 25 + green_pixels * 5 + blue_pixels ;

[15] See *29.6 Paint Driver Library (p. 424)*.

30 Writing GRASS Shell Scripts

This section describes some of the things a programmer should consider when writing a shell script that will become a GRASS command.

30.1 Use the Bourne Shell

The Bourne Shell (/bin/sh) is the original UNIX command interpreter. It is available on most (if not all) versions of UNIX. Other command interpreters, such as the C-Shell (/bin/csh), are not as widely available. Therefore, programmers are strongly encouraged to write Bourne Shell scripts for maximum portability.

The discussion that follows is for the Bourne Shell only. It is also assumed that the reader knows (or can learn) how to write Bourne Shell scripts. This chapter is intended to provide guidelines for making them work properly as GRASS commands.

30.2 How a Script Should Start

There are some things that should be done at the beginning of any GRASS shell script:

(1) Verify that the user is running GRASS, and

(2) Cast the GRASS environment variables into the UNIX environment,[1] and verify that the variables needed by the shell script are set.

```
#!/bin/sh

if test "$GISRC" = ""
then
   echo "Sorry, you are not running GRASS" >\&2
   exit 1
fi

eval `g.gisenv`

: ${GISBASE?} ${GISDBASE?} ${LOCATION_NAME?} ${MASPET?}
```

[1] See *10 Environment Variables (p. 65)*

Note the use of the **:** command. This command simply evaluates its arguments. The syntax ${GISBASE?} means that if GISBASE is not set, issue an error message to standard error and exit the shell script.

30.3 g.ask

The GRASS command *g.ask* emulates the prompting found in all other GRASS commands, and should be used in shell scripts to ask the user for files from the GRASS database. The user's response can be cast into shell variables. The following example asks the user to select an existing raster file:

```
g.ask type=old prompt="Select a raster file" element=cell desc=raster unixfile=/tmp/$$
. /tmp/$$

rm -f /tmp/$$

if test "$name" = ""
then
  exit 0
fi
```

The *g.ask* manual entry in the *GRASS User's Reference Manual* describes this command in detail. Here, the reader should note the following:

(1) The temporary file used to hold the user's response is */tmp/$$* . The Bourne Shell will substitute its process id for the *$$* thus creating a unique file name;

(2) The next line, which begins with a dot, sources the commands contained in the temporary file. These commands are:

```
name=something
mapset=something
file=something
```

Therefore, the variables $name, $mapset, and $file will contain the name, mapset and full UNIX file name of the raster file selected by the user;

(3) The temporary file is removed; and

(4) If $name is empty, this means that the user changed his or her mind and did not select any raster file.[2] In this case, something reasonable is done, like exiting.

[2] The other variables will be empty as well.

30.4 g.findfile

The *g.findfile* command can be used to locate GRASS files that were specified as arguments to the shell script (instead of prompted for with *g.ask*). Assuming that the variable $request contains the name of a raster file, the following checks to see if the file exists. If it does, the variables $name, $mapset and $file will be set to the name, mapset and full UNIX file name for the raster file:

```
eval `g.findfile element=cell file="$request"`

if test "$mapset" = ""
then
  echo ERROR: raster file "$request" not found >&2
  exit 1
fi
```

Note. The programmer should use quotes with $request, since it may contain spaces. (quotes will preserve the full request). If found, *g.findfile* outputs $name as the name part and $mapset as the mapset part. See the *g.findfile* manual entry in the *GRASS User's Reference Manual* for more details.

31 GRASS CVS repository

The Concurrent Versions System (CVS) provides network-transparent source control for groups of developers. They can work independently and "feed the system" (the source code repository) remotely through internet - day and night from all over the world.

Even if you do not contribute to the code development, the CVS allows to follow development progress easily. After downloading the full GRASS CVS tree, just the future changes will be sent to you by using the CVS software. This method reduces further downloads for updating your local GRASS sources extremely.

Further features of CVS:

- maintains a history of all changes and keeps copies of all changes (a sort of "change recorder")
- provides tools to support process control and change control
- provides reliable access to its directory trees from remote hosts using Internet protocols
- supports parallel development allowing more than one developer to work on the same sources at the same time

The CVS prevents us and you from further GRASS version confusion. As every change is under control of the CVS system the current state as well as old versions can be accessed. This is very important for bug-tracking (finding bugs introduced by changes).

If you download the GRASS sources from the CVS system, you get the latest version existing! Of course, you can also download a previous version, if you prefer. Once downloaded the entire GRASS CVS tree including the CVS-internal subdirectories you can follow the ongoing development easily. The "update" function downloads only the changes not yet included in your system. CVS compares the GRASS source code version in the CVS-server with your local version.

As GRASS is a rather big package this might be one reason for you to use CVS. If you are not yet convinced please read the texts below. CVS is distributed here and usually shipped within several Linux distributions. It is available for rather every operating system.

The GRASS-CVS service is hosted at Intevation GmbH, Germany.[1] There is additionally a "web-cvs" interface which allows to browse the source code. The status of each file (its change history) is written there, you can also download individual files.

[1] http://freegis.org/grass/

A Appendix

A.1 Appendix A: Annotated Gmakefile Predefined Variables

The predefined Gmakefile variables are defined in the files *head.in* and *make.mid.* These files can be found under $GISBASE/src/CMD.[1]
Note: Some of the variables shown here are described in more detail in *11 Compiling and Installing GRASS Modules (p. 69)*

head
The *head* file contains machine dependent and installation dependent information. It is created by system personnel when GRASS is installed on a system prior to compilation. This file varies from system to system. The name of this file may also vary, depending on the machine or architecture for which GRASS is compiled.
Here is a sample *head.i686-pc-linux-gnu* (Linux-PC architecture) file:

Variable	Value	Description
CC	= gcc	C-Compiler
FC	= g77	Fortran-Compiler
LEX	= flex	lexical analyser
YACC	= yacc	lexical parser
ARCH	= i686-pc-linux-gnu	Architecture to compile on
MAKE	= make	make command
GISBASE	= /usr/local/grass-5.0b	Location of GRASS program
UNIX_BIN	= /usr/local/bin	GRASS startup script location
DEFAULT_DATABASE	=	Location of default database
DEFAULT_LOCATION	=	Name of defaut database
COMPILE_FLAGS	= -O2	Compiler flags
LDFLAGS	= -s	Loader flags
DLLIB	= -ldl	
XCFLAGS	= -I/usr/X11R6/include	
XLDFLAGS	=	
XINCPATH	=	
XMINCPATH	=	

[1] $GISBASE is the directory where GRASS is installed. See *10.1 UNIX Environment (p. 65)* for details.

```
XLIBPATH        = -L/usr/X11R6/lib
XTLIBPATH       =
XMLIBPATH       =
XLIB            = -lSM -lICE -lX11
XTLIB           = -lXt
XMLIB           = -lXm
XEXTRALIBS      =

COMPATLIB       =
TERMLIB         =                                   Terminal emulation libraries
CURSES          = -lncurses $(TERMLIB) $(COMPATLIB) ncurses libraries
MATHLIB         = -lm                               Math libraries
XDRLIB          = -lnsl

#PostgreSQL:
PQINCPATH       = -I/usr/include/pgsql
PQLIBPATH       = /usr/include/lib
PQLIB           = -L/usr/include/lib -lpq -lcrypt

#ODBC:
ODBCINC         = -I/usr/local/include

#Image formats:
PNGINC          = -I/usr/include
PNGLIB          = -L/usr/X11R6/lib -L/usr/lib

JPEGINCPATH     = -I/usr/include
JPEGLIBPATH     = -L/usr/X11R6/lib -L/usr/lib

TIFFINCPATH     = -I/usr/local/include
TIFFLIBPATH     = -L/usr/local/lib

#openGL files for       NVIZ/r3.showdspf
OPENGLINC       = -I/usr/local/include/GL
OPENGLwINC      = -I/usr/X11R6/include/GL
OPENGLLIB       = -l/usr/lib/libGL.so
OPENGLULIB      = -l/usr/lib/libGLU.so
LGLWM           =

#tcl/tk stuff
TCLINCDIR       = -I/usr/include
TKINCDIR        = -I/usr/include
TCLTKLIBS       = -L/usr/lib -ltk8.3 -ltcl8.3

LIBRULE         = ar ruv $ $?; ranlib $             Library archiver
USE_TERMIO      =                                   Use TERMIO or not?

# Uncomment below to    input Korean(Asian?) characters     in v.digit or etc.:
```

```
#DASIAN_CHARS       = -DASIAN_CHARS

# It's hard to click    two buttons simultaneously    with two button mouse.
# Situation is worse,   if it requires continuous     clicking.
# In this case uncomment below.
#DANOTHER_BUTTON    = -DANOTHER_BUTTON
```

make.mid

The *make.mid* file uses the variables in *head* to construct other variables that are useful for compilation rules. The contents of this file are usually unchanged from system to system.
Here is a sample *make.mid* file:

Variable	Value	Description
SHELL	= /bin/sh	
BIN	= $(GISBASE)/bin	GRASS command links
ETC	= $(GISBASE)/etc	Main GRASS commands
SCRIPTS	= $(GISBASE)/scripts	GRASS scripts
BIN_INTER	= $(ETC)/bin/main/inter	Main interactive commands
BIN_CMD	= $(ETC)/bin/main/cmd	Main command-line commands
TXT	= $(GISBASE)/txt	Text directory
MAN1	= $(GISBASE)/man/1	Manual page directories
MAN2	= $(GISBASE)/man/2	
MAN3	= $(GISBASE)/man/3	
MAN4	= $(GISBASE)/man/4	
MAN5	= $(GISBASE)/man/5	
MAN6	= $(GISBASE)/man/6	
HELP	= $(GISBASE)/man/help	
CFLAGS	= $(COMPILE_FLAGS) $(EXTRA_CFLAGS) -I$(LIBDIR) $(USE_TERMIO)	
AR	= $(GMAKE) -makeparentdir $@;	All library archiver flags
$(LIBRULE)		
MANROFF	= tbl -TX	Manual formatter command and options

```
$(SRC)/man.help/man.version
$(SRC)/man.help/man.header $? |
nroff -Tlp | col -b > $@
```

MAKEALL	= $(GMAKE) -all	Command to make GRASS
LIBDIR	= $(SRC)/libes	GRASS li-
braries		
DIG_LIBDIR	= $(SRC)/mapdev/libes	
DIG_INCLUDE	= $(SRC)/mapdev/lib	
VECT_INCLUDE	=-I$(SRC)/mapdev/Vlib	
-I$(SRC)/mapdev/diglib		
VASKLIB	= $(LIBDIR)/libvask.a	Vask libraries
VASK	= $(VASKLIB) $(CURSES)	Vask and flags

Variable	Value	Description
GISLIB	= $(LIBDIR)/libgis.a	GIS libraries
ICONLIB	= $(LIBDIR)/libicon.a	
LOCKLIB	= $(LIBDIR)/liblock.a	
IMAGERYLIB	= $(LIBDIR)/libI.a	GIS Libraries
RO WIOLIB	= $(LIBDIR)/librowio.a	
COORCNVLIB	= $(LIBDIR)/libcoorcnv.a	
SEGMENTLIB	= $(LIBDIR)/libsegment.a	
BTREELIB	= $(LIBDIR)/libbtree.a	
DLGLIB	= $(LIBDIR)/libdlg.a	
RASTERLIB	= $(LIBDIR)/libraster.a	
DISPLAYLIB	= $(LIBDIR)/libdisplay.a	
D_LIB	= $(LIBDIR)/libD.a	
DRIVERLIB	= $(SRC)/display/devices/lib/driverlib.a	
LINKMLIB	= $(LIBDIR)/liblinkm.a	
DIGLIB	= $(LIBDIR)/libdig.a	
DIG2LIB	= $(LIBDIR)/libdig2.a	
VECTLIB_REAL	= $(LIBDIR)/libvect.a	
VECTLIB	= $(VECTLIB_REAL) $(DIG2LIB)	
DIG_ATTLIB	= $(LIBDIR)/libdig_atts.a	
XDISPLAYLIB	= $(LIBDIR)/libXdisplay.a	

A.2 Appendix B: The CELL Data Type

GRASS cell file data is defined to be of type CELL. This data type is defined in the "gis.h" header file. Programmers must declare all variables and buffers which will hold raster data or category codes as type CELL.
Under GRASS the CELL data type is declared to be *int,* but the programmer should not assume this. What should be assumed is that CELL is a signed integer type. It may be changed sometime to *short* or *long.* This implies that use of CELL data with routines which do not know about this data type (e.g., fprintf(stdout,), sscanf( ), etc.) must use an intermediate variable of type *long.*
To print a CELL value, it must be cast to *long.* For example:

```
CELL c;                                    /* raster value to be printed */

/* some code to get a value for c */

fprintf(stdout, "%ld\n", (long) c);                    /* cast c to long to print */
```

To read a CELL value, for example from user typed input, it is necessary to read into a *long* variable, and then assign it to the CELL variable. For example:[2]

```
char userbuf[128];
CELL c; long x;

fprintf (stdout, "Which category? ");                  /* prompt user */
gets(userbuf); /* get user response *          /

sscanf (userbuf,"%ld", &x);                            /* scan category into long variable */
c = (CELL) x;                                          /* assign long value to CELL
value */
```

Of course, with GRASS library routines that are designed to handle the CELL type, this problem does not arise. It is only when CELL data must be used in routines which do not know about the CELL type, that the values must be cast to or from *long*.

[2] This example does not check for valid inputs, EOF, etc., which good code must do.

A.3 Appendix C: Index to GIS Library

Here is an index of GIS Library routines, with calling sequences and short function descriptions.

GIS Library

routine	parameters	description
G_add_color_rule	(cat1, r1, g1, b1, cat2, r2, g2, b2, colors)	set colors
G_adjust_Cell_head	(cellhd, rflag, cflag)	adjust cell header
G_adjust_easting	(east, region)	returns east larger than west
G_adjust_east_longitude	(east, west)	adjust east longitude
G_align_window	(region, ref)	align two regions
G_allocate_cell_buf	( )	allocate a raster buffer
G_area_for_zone_on_ellipsoid	(north, south)	area between latitudes
G_area_for_zone_on_sphere	(north, south)	area between latitudes
G_area_of_cell_at_row	(row)	cell area in specified
G_area_of_polygon	(x, y, n)	area in square meters of polygon
G_ask_any	(prompt, name, element, label, warn)	prompt for any valid file name
G_ask_cell_in_mapset	(prompt, name)	prompt for existing raster file
G_ask_cell_new	(prompt, name)	prompt for new raster file
G_ask_cell_old	(prompt, name)	prompt for existing raster file
G_ask_in_mapset	(prompt, name, element, label)	prompt for existing database file
G_ask_new	(prompt, name, element, label)	prompt for new database file
G_ask_old	(prompt, name, element, label)	prompt for existing database file

G_ask_sites_in_mapset	(prompt, name)	prompt for existing site list file
G_ask_sites_new	(prompt, name)	prompt for new site list file
G_ask_sites_old	(prompt, name)	prompt for existing site list file
G_ask_vector_in_mapset	(prompt, name)	prompt for an existing vector file
G_ask_vector_new	(prompt, name)	prompt for a new vector file
G_ask_vector_old	(prompt, name)	prompt for an existing vector file
G_begin_cell_area_calculations	( )	begin cell area calculations
G_begin_distance_calculations	( )	begin distance calculations
G_begin_ellipsoid_polygon_area	(a, e2)	begin area calculations
G_begin_geodesic_distance	(a, e2)	begin geodesic distance
G_begin_polygon_area_calculations	( )	begin polygon area calculations
G_begin_zone_area_on_ellipsoid	(a, e2, s)	begin ellipsoid area calculations
G_begin_zone_area_on_sphere	(r, s)	initialize calculations for sphere
G_bresenham_line	(x1, y1, x2, y2, point)	Bresenham line algorithm
G_calloc	(n,size)	memory allocation
G_close_cell	(fd)	close a raster file
G_col_to_easting	(col, region)	column to easting
G_database_projection_name	(proj)	query cartographic projection
G_database_unit_name	(plural)	database units
G_database_units_to_meters_factor	( )	conversion to meters
G_date	( )	current date and time
G_define_flag	( )	return Flag structure
G_define_option	( )	returns Option structure

G_disable_interactive	()	turns off interactive capability
G_distance	(x1, y1, x2, y2)	distance in meters
G_easting_to_col	(east, region)	easting to column
G_ellipsoid_name	(n)	return ellopsoid name
G_ellipsoid_polygon_area	(lon, lat, n)	area of lat-long polygon
G_fatal_error	(message)	print error message and exit
G_find_cell	(name,mapset)	find a raster file
G_find_cell_stat	(cat, count, s)	random query of cell stats
G_find_file	(element, name, mapset)	find a database file
G_find_vector2	(name,mapset)	find a vector file
G_find_vector	(name,mapset)	find a vector file
G_fopen_append	(element, name)	open a database file for update
G_fopen_new	(element, name)	open a new database file
G_fopen_old	(element, name, mapset)	open a database file for reading
G_fopen_sites_new	(name)	open a new site list file
G_fopen_sites_old	(name, mapset)	open an existing site list file
G_fopen_vector_new	(name)	opena new vector file
G_fopen_vector_old	(name, mapset)	open an existing vector file
G_fork	( )	create a protected child process
G_format_easting	(east, buf, projection)	easting to ASCII
G_format_northing	(north, buf, projection)	northing to ASCII
G_format_resolution	(resolution, buf, projection)	resolution to ASCII
G_free_cats	(cats)	free category structure memory

G_free_cell_stats	(s)	free cell stats
G_free_colors	(colors)	free color structure memory
G_fully_qualified_name	(name, mapset)	fully qualified file name
G_geodesic_distance	(lon1, lat1, lon2, lat2)	geodesic distance
G_geodesic_distance_lon_to_lon	(lon1, lon2)	geodesic distance
G_get_ask_return_msg	( )	get Hit RETURN msg
G_get_cat	(n,cats)	get a category label
G_get_cats_title	(cats)	get title from category structure
G_get_cellhd	(name, mapset, cellhd)	read the raster header
G_get_cell_title	(name, mapset)	get raster map title
G_get_color	(cat, red, green, blue, colors)	get a category color
G_get_color_range	(min, max, colors)	get color range
G_get_default_window	(region)	read the default region
G_get_ellipsoid_by_name	(name, a, e2)	get ellipsoid by name
G_get_ellipsoid_parameters	(a, e2)	get ellipsoid parameters
G_ _getenv	(name)	query GRASS environment variable
G_getenv	(name)	query GRASS environment variable
G_get_map_row	(fd, cell, row)	read a raster file
G_get_map_row_nomask	(fd, cell, row)	read a raster file (without masking)
G_get_range_min_max	(range, min, max)	get range min and max
G_gets	(buf)	get a line of input (detect ctrl-z)
G_get_set_window	(region)	get the active region
G_get_site	(fd, east, north, desc)	read site list file
G_get_window	(region)	read the database region

G_gisbase	( )	top level module directory
G_gisdbase	( )	top level database directory
G_gisinit	(program_name)	initialize gis library
G_home	()	user's home directory
G_init_cats	(n, title, cats)	initialize category structure
G_init_cell_stats	(s)	initialize cell stats
G_init_colors	(colors)	initialize color structure
G_init_range	(range)	initialize range structure
G_intr_char	( )	return interrupt char
G_is_reclass	(name, mapset, r_name, r_mapset)	reclass file?
G_legal_filename	(name)	check for legal database file names
G_location	( )	current location name
G_location_path	( )	current location directory
G_lookup_colors	(raster, red, green, blue, set, n, colors)	lookup an array of colors
G_make_aspect_colors	(colors, min, max)	make aspect colors
G_make_grey_scale_colors	(colors, min, max)	make linear grey scale
G_make_gyr_colors	(colors, min, max)	make green,yellow,red colors
G_make_histogram_eq_colors	(colors, s)	make histogram-stretched grey colors
G_make_rainbow_colors	(colors, min, max)	make rainbow colors
G_make_ramp_colors	(colors, min, max)	make color ramp
G_make_random_colors	(colors, min, max)	make random colors
G_make_ryg_colors	(colors, min, max)	make red,yellow,green colors
G_make_wave_colors	(colors, min, max)	make color wave
G_malloc	(size)	memory allocation

G_mapset	( )	current mapset name
G_meridional_radius_of_curvature	(lon, a, e2)	meridional radius of curvature
G_myname	()	location title
G_next_cell_stat	(cat, count, s)	retrieve sorted cell stats
G_northing_to_row	(north, region)	northing to row
G_open_cell_new	(name)	open a new raster file (sequential)
G_open_cell_new_random	(name)	open a new raster file (random)
G_open_cell_new_uncompressed	(name)	open a new raster file (uncompressed)
G_open_cell_old	(name, mapset)	open an existing raster file
G_open_new	(element, name)	open a new database file
G_open_old	(element, name, mapset)	open a database file for reading
G_open_update	(element, name)	open a database file for update
G_parser	(argc, argv)	parse command line
G_percent	(n, total, incr)	print percent complete messages
G_planimetric_polygon_area	(x, y, n)	area in coordinate units
G_plot_fx	(f, east1, east2)	plot f(east1) to f(east2)
G_plot_line	(east1, north1, east2, north2)	plot line between latlon coordinates
G_plot_polygon	(east, north, n)	plot filled polygon with n vertices
G_plot_where_en	(x, y, east, north)	x,y to east,north
G_plot_where_xy	(east, north, x, y)	east,north to x,y
G_pole_in_polygon	(x, y, n)	pole in polygon

G_program_name	( )	return module name
G_projection	( )	query cartographic projection
G_put_cellhd	(name, cellhd)	write the raster header
G_put_cell_title	(name, title)	change raster map title
G_put_map_row	(fd, buf)	write a raster file (sequential)
G_put_map_row_random	(fd, buf, row, col, ncells)	write a raster file (random)
G_put_site	(fd, east, north, desc)	write site list file
G_put_window	(region)	write the database region
G_radius_of_conformal_tangent_sphere	(lon, a, e2)	radius of conformal tangent sphere
G_read_cats	(name, mapset, cats)	read raster category file
G_read_colors	(name, mapset, colors)	read map layer color table
G_read_history	(name, mapset, history)	read raster history file
G_read_range	(name, mapset, range)	read raster range
G_read_vector_cats	(name, mapset, cats)	read vector category file
G_realloc	(ptr, size)	memory allocation
G_remove	(element, name)	remove a database file
G_rename	(element, old, new)	rename a database file
G_rewind_cell_stats	(s)	reset/rewind cell stats
G_row_to_northing	(row, region)	row to northing
G_row_update_range	(cell, n, range)	update range structure
G_scan_easting	(buf, easting, projection)	ASCII easting to double
G_scan_northing	(buf, northing, projection)	ASCII northing to double

G_tolcase	(s)	convert string to lower case
G_toucase	(s)	convert string to upper case
G_transverse_radius_of_curvature	(lon, a, e2)	transverse radius of curvature
G_unctrl	(c)	printable version of control character
G_unopen_cell	(fd)	unopen a raster file
G_unset_error_routine	( )	reset normal error handling
G_update_cell_stats	(data, n, s)	add data to cell stats
G_update_range	(cat, range)	update range structure
G_usage	( )	command line help/usage message
G_warning	(message)	print warning message and continue
G_whoami	( )	user's name
G_window_cols	( )	number of columns in active region
G_window_rows	( )	number of rows in active region
G_write_cats	(name, cats)	write raster category file
G_write_colors	(name, mapset, colors)	write map layer color table
G_write_history	(name, history)	write raster history file
G_write_range	(name, range)	write raster range file
G_write_vector_cats	(name, cats)	write vector category file
G_yes	(question,default)	ask a yes/no question
G_zero_cell_buf	(buf)	zero a raster buffer
G_zone	( )	query cartographic zone

A.4 Appendix D: Index to Vector Library

Here is an index of vector Library routines, with calling sequences and short function descriptions.

vector Library

routine	parameters	description
dig_check_dist	(Map, n, x, y, d)	find distance of point to line
dig_point_in_area	(Map, x, y, pa)	is point in area?
dig_point_to_area	(Map, x, y)	find which area point is in
dig_point_to_line	(Map, x, y, type)	find which arc point is closest to
V1_read_line	(Map, Points, offset)	read vector arc by specifying offset
V2_area_att	(Map, area)	get attribute number of area
V2_get_area_bbox	(Map, area, n, s, e, w)	get bounding box of area
V2_get_area	(Map, n, pa)	get area info from id
V2_get_line_bbox	(Map, line, n, s, e, w)	get bounding box of arc
V2_line_att	(Map, line)	get attribute number of arc
V2_num_areas	(Map)	get number of areas in vector map
V2_num_lines	(Map)	get number of arcs in vector map
V2_read_line	(Map, Points, line)	read vector arc by specifying line id
Vect_close	(Map)	close a vector map
Vect_copy_head_data	(from, to)	copy vector header struct data
Vect_copy_pnts_to_xy	(Points, x, y, n)	convert line_pnts structure to xy arrays
Vect_copy_xy_to_pnts	(Points, x, y, n)	convert xy arrays to line_pnts structure

Vect_destroy_line_struct line points structure space	(Points)	deallocate
Vect_get_area_points points for area polygon	(Map, area, Points)	get defining
Vect_level vector map	(Map)	get open level of
Vect_new_line_struct initialized line points structure	( )	create new
Vect_open_new tor map	(Map, name)	open new vec-
Vect_open_old existing vector map	(Map, name, mapset)	open
Vect_print_header info to stdout	(Map)	print header
Vect_read_next_line vector line	(Map, Points)	read next
Vect_remove_constraints tor read constraints	(Map)	unset any vec-
Vect_rewind map for re-reading	(Map)	rewind vector
Vect_set_constraint_region stricted region to read vector arcs from	(Map, n, s, e, w)	set re-
Vect_set_constraint_type arcs to read	(Map, type)	specify types of
Vect_set_open_level level for opening map	(level)	specify
Vect_write_line arc to vector map	(Map, type, Points)	write out

A.5 Appendix E: Index to Imagery Library

Here is an index of Imagery Library routines, with calling sequences and short function descriptions.

Imagery Library

routine	parameters	description
I_add_file_to_group_ref name to Ref structure	(name, mapset, ref)	add file
I_ask_group_any any valid group name	(prompt, group)	prompt for

I_ask_group_new	(prompt, group)	prompt for new group
I_ask_group_old	(prompt, group)	prompt for an existing group
I_find_group	(group)	does group exist?
I_free_group_ref	(ref)	free Ref structure
I_get_control_points	(group, cp)	read group control points
I_get_group_ref	(group, ref)	read group REF file
I_get_subgroup_ref	(group, subgroup, ref)	read subgroup REF file
I_get_target	(group, location, mapset)	read target information
I_init_group_ref	(ref)	initialize Ref structure
I_new_control_point	(cp, e1, n1, e2, n2, status)	add new control point
I_put_control_points	(group, cp)	write group control points
I_put_group_ref	(group, ref)	write group REF file
I_put_subgroup_ref	(group, subgroup, ref)	write subgroup REF file
I_put_target	(group, location, mapset)	write target information
I_transfer_group_ref_file	(src, n, dst)	copy Ref lists

A.6 Appendix F: Index to Display Graphics Library

Here is an index of Display Graphics Library routines, with calling sequences and short function descriptions.

Display Graphics Library

routine	parameters	description
D_add_to_list	(string)	add command to frame display list
D_a_to_d_col	(column)	array to screen

(column)		
D_a_to_d_row (row)	(row)	array to screen
D_cell_draw_setup for raster graphics	(top, bottom, left, right)	prepare
D_check_colormap_size of colors	(min,max,ncolors)	verify a range
D_check_map_window current map region	(region)	assign/retrieve
D_clear_window play lists	( )	clear frame dis-
D_clear_window tion about current frame	( )	clears informa-
D_clip dinates to window	(s, n, w, e, x, y, c_x, c_y)	clip coor-
D_color color for line	(cat, colors)	select raster
D_cont_abs	(x,y)	line to x,y
D_cont_rel	(x,y)	line to x,y
D_do_conversions conversions	(region, top, bottom, left, right)	initialize
D_draw_cell raster row	(row, raster, colors)	render a
D_d_to_a_col (x)	(x)	screen to array
D_d_to_a_row (y)	(y)	screen to array
D_d_to_u_col	(x)	screen to earth (x)
D_d_to_u_row (y)	(y)	screen to earth
D_erase_window frame	( )	erase current
D_get_cell_name raster map name	(name)	retrieve
D_get_cur_wind rent graphics frame	(name)	identify cur-
D_get_screen_window frame coordinates	(top, bottom, left, right)	retrieve current
D_lookup_colors hardware color	(data, n, colors)	change to
D_move_abs	(x,y)	move to pixel
D_move_rel	(x,y)	move to pixel

D_new_window (name, top, bottom, left, right) create new graphics frame

D_popup (bcolor, tcolor, dcolor, top, left, size, options) pop-up menu

D_raster (raster, n, repeat, colors) low lev el raster plotting

D_remove_window ( ) remove a frame

D_reset_color (data, r, g, b) reset raster color value

D_reset_colors (colors) set colors in driver

D_reset_screen_window (top, bottom, left, right) resets current frame position

D_set_cell_name (name) add raster map name to display list

D_set_clip_window_to_map_window ( ) set clipping window to map window

D_set_clip_window (top, bottom, left, right) set clip-ping window

D_set_colors (colors) establish raster colors for graphics

D_set_cur_wind (name) set current graphics frame

D_set_overlay_mode (flag) configure raster overlay mode

D_setup (clear) graphics frame setup

D_setup (clear) initialize/create a frame

D_show_window (color) outlinescurrent frame

D_timestamp ( ) give current time to frame

D_translate_color (name) color name to number

D_u_to_a_col (east) earth to array (east)

D_u_to_a_row (north) earth to ar-ray (north)

D_u_to_d_col (east) earth to screen (east)

D_u_to_d_row (north) earth to screen (north)

A.7 Appendix G: Index to Raster Graphics Library

Here is an index of Raster Graphics Library routines, with calling sequences and short function descriptions.

Raster Graphics Library

routine	parameters	description
R_box_abs	(x1,y1,x2,y2)	fill a box
R_box_rel	(dx,dy)	fill a box
R_close_driver	( )	terminate graphics
R_color	(color)	select color
R_color_table_fixed	( )	select fixed color table
R_color_table_float	( )	select floating color table
R_cont_abs	(x,y)	draw line
R_cont_rel	(dx,dy)	draw line
R_erase	()	erase screen
R_flush	()	flush graphics
R_font	(font)	choose font
R_get_location_with_box	(x,y,nx,ny,button)	get mouse location using a box
R_get_location_with_line	(x,y,nx,ny,button)	get mouse location using a line
R_get_location_with_pointer	(nx,ny,button)	get mouse location using pointer
R_get_text_box	(text, top, bottom, left, right)	get text extents
R_move_abs	(x,y)	move current location
R_move_rel	(dx,dy)	move current location
R_open_driver	( )	initialize graphics
R_polydots_abs	(x,y,num)	draw a series of dots
R_polydots_rel	(x,y,num)	draw a series of dots
R_polygon_abs	(x,y,num)	draw a closed polygon

R_polygon_rel	(x,y,num)	draw a closed polygon
R_polyline_abs	(x,y,num)	draw an open polygon
R_polyline_rel	(x,y,num)	draw an open polygon
R_raster	(num,nrows,withzero,raster)	draw a raster
R_reset_color	(red, green, blu, num)	define single color
R_reset_colors	(min,max,red,green,blue)	define multiple colors
R_RGB_color	(red,green,blue)	select color
R_RGB_raster	(num,nrows,red,green,blue,withzero)	draw a raster
R_screen_bot	( )	bottom of screen
R_screen_left	( )	screen left edge
R_screen_rite	( )	screen right edge
R_screen_top	( )	top of screen
R_set_RGB_color	(red,green,blue)	initialize graphics
R_set_window	(top,bottom,left,right)	set text clipping frame
R_stabilize	()	synchronize graphics
R_standard_color	(color)	select standard color
R_text_size	(width, height)	set text size
R_text	(text)	write text

A.8 Appendix H: Index to Rowio Library

Here is an index of Rowio Library routines, with calling sequences and short function descriptions.

Rowio Library

routine	parameters	description
rowio_fileno	(r)	get file descriptor
rowio_flush	(r)	force pending updates to disk
rowio_forget	(r, n)	forget a row
rowio_get	(r, n)	read a row
rowio_put	(r, buf, n)	write a row
rowio_release	(r)	free allocated memory

rowio_setup	(r, fd, nrows, len, getrow, putrow)	configure rowio structure

A.9 Appendix I: Index to Segment Library

Here is an index of Segment Library routines, with calling sequences and short function descriptions.

Segment Library

routine	parameters	description
segment_flush	(seg)	flush pending updates to disk
segment_format	(fd, nrows, ncols, srows, scols, len)	format a segment file
segment_get_row	(seg, buf, row)	read row from segment file
segment_get	(seg, value, row, col)	get value from segment file
segment_init	(seg, fd, nsegs)	initialize segment structure
segment_put_row	(seg, buf, row)	write row to segment file
segment_put	(seg, value, row, col)	put value to segment file
segment_release	(seg)	free allocated memory

A.10 Appendix J: Index to Vask Library

Here is an index of Vask Library routines, with calling sequences and short function descriptions.

Vask Library

routine	parameters	description
V_call	( )	interact with the user
V_clear	( )	initialize screen description
V_const	(value, type, row, col, len)	define screen constant

V_float_accuracy	(num)	set number of decimal places
V_intrpt_msg	(text)	change ctrl-c message
V_intrpt_ok	( )	allow ctrl-c
V_line	(num, text)	add line of text to screen
V_ques	(value, type, row, col, len)	define screen question

A.11 Appendix K: Index to Grid3D Library Subroutines

G3D Function Index

G3d_getFileType
G3d_getFloat
G3d_getFloatRegion
G3d_getNearestNeighborFunPtr
G3d_getNofTilesMap
G3d_getRegionMap
G3d_getRegionStructMap
G3d_getRegionValue
G3d_getResamplingFun
G3d_getStandard3dInputParams
G3d_getTileDimension
G3d_getTileDimensionsMap
G3d_getTilePtr
G3d_getValue
G3d_getValueRegion
G3d_getVolumeA
G3d_getWindow

G3d_incorporate2dRegion
G3d_initDefaults
G3d_isMasked
G3d_isNullValueNum
G3d_isValidLocation

G3d_location2coord
G3d_lockTile

G3d_makeMapsetMapDirectory
G3d_malloc
G3d_maskDouble
G3d_maskFile
G3d_maskFileExists
G3d_maskFloat
G3d_maskIsOff
G3d_maskIsOn
G3d_maskMapExists
G3d_maskNum
G3d_maskOff
G3d_maskOn
G3d_maskReopen
G3d_maskTile
G3d_minUnlocked

G3d_nearestNeighbor

G3d_openCellNew
G3d_openCellNewParam
G3d_openCellOld

G3d_printError
G3d_printHeader
G3d_putDouble
G3d_putFloat
G3d_putValue

G3d_range_load
G3d_range_min_max
G3d_range_write
G3d_readCats

G3d_readColors
G3d_readRegionMap
G3d_readTile
G3d_readTileDouble
G3d_readTileFloat
G3d_readWindow
G3d_realloc
G3d_regionCopy
G3d_removeColor
G3d_removeTile
G3d_retile

G3d_setCacheLimit
G3d_setCacheSize
G3d_setCompressionMode
G3d_setErrorFun
G3d_setFileType
G3d_setNullTile
G3d_setNullTileType
G3d_setNullValue
G3d_setResamplingFun
G3d_setStandard3dInputParams
G3d_setTileDimension
G3d_setUnit
G3d_setWindow
G3d_setWindowMap
G3d_skipError

G3d_tile2tileIndex
G3d_tileCoordOrigin
G3d_tileIndex2tile
G3d_tileIndexInRange
G3d_tileIndexOrigin
G3d_tileInRange
G3d_tileLoad
G3d_tilePrecisionMap
G3d_tileTypeMap
G3d_tilePrecisionMap

G3d_unlockAll
G3d_unlockTile
G3d_useWindowParams

G3d_windowPtr
G3d_writeAscii
G3d_writeCats
G3d_writeColors
G3d_writeTile
G3d_writeTileDouble
G3d_writeTileFloat
G3d_writeWindow

A.12 Appendix L: Index to DateTime Library Subroutines

datetime_change_from_to
datetime_change_timezone

```
datetime_change_to_utc
datetime_check_day
datetime_check_fracsec
datetime_check_hour
datetime_check_increment
datetime_check_minute
datetime_check_month
datetime_check_second
datetime_check_timezone
datetime_check_type
datetime_check_year
datetime_clear_error
datetime_copy
datetime_days_in_month
datetime_days_in_year
datetime_decompose_timezone
datetime_difference
datetime_error
datetime_format
datetime_get_day
datetime_get_error_code
datetime_get_error_msg
datetime_get_fracsec
datetime_get_hour
datetime_get_increment_type
datetime_get_local_time
datetime_get_local_timezone
datetime_get_minute
datetime_get_month
datetime_get_second
datetime_get_timezone
datetime_get_type
datetime_get_year
datetime_increment
datetime_invert_sign
datetime_is_absolute
datetime_is_leap_year
datetime_is_positive
datetime_is_relative
datetime_is_same
datetime_is_valid_increment
datetime_is_valid_timezone
datetime_is_valid_type
datetime_scan
datetime_set_day
datetime_set_fracsec
datetime_set_hour
datetime_set_increment_type
datetime_set_minute
datetime_set_month
datetime_set_negative
datetime_set_positive
datetime_set_second
datetime_set_timezone
datetime_set_type
datetime_set_year
datetime_unset_timezone
```

A.13 Appendix M: Permuted Index for Library Subroutines

get theactive region	G_get_set_window()
set theactive region	G_set_window()
number of columns inactive region	G_window_cols()
number of rows inactive region	G_window_rows()
add command to frame display list	D_add_to_list()
add data to cell stats	G_update_cell_stats()
add file name to Ref structure	I_add_file_to_group_ref()
add line of text to screen	V_line()
add new control point	I_new_control_point()
add raster map name to display list	D_set_cell_name()
adjust cell header	G_adjust_Cell_head()
adjust east longitude	G_adjust_east_longitude()
Bresenham linealgorithm	G_bresenham_line()
align two regions	G_align_window()
allocate a raster buffer	G_allocate_cell_buf()
copy string toallocated memory	G_store()
freeallocated memory	rowio_release()
freeallocated memory	segment_release()
memoryallocation	G_calloc()
memoryallocation	G_malloc()
memoryallocation	G_realloc()
allow ctrlhyphenc	V_intrpt_ok()
get bounding box ofarc	Vtwo_get_line_bbox()
get attribute number ofarc	Vtwo_line_att()
read vectorarc by specifying line id	Vtwo_read_line()
read vectorarc by specifying offset	Vone_read_line()
find whicharc point is closest to	dig_point_to_line()
write outarc to vector map	Vect_write_line()
set restricted region to read vectorarcs from	Vect_set_constraint_region()
get number ofarcs in vector map	Vtwo_num_lines()
specify types ofarcs to read	Vect_set_constraint_type()
is point inareaquestion	dig_point_in_area()
get attribute number ofarea	Vtwo_area_att()
get bounding box ofarea	Vtwo_get_area_bbox()
area between latitudes	G_area_for_zone_on_ellipsoid()
area between latitudes	G_area_for_zone_on_sphere()
begin cellarea calculations	G_begin_cell_area_calculations()
beginarea calculations	G_begin_ellipsoid_polygon_area()
begin polygonarea calculations	G_begin_polygon_area_calculations()
beginarea calculations for ellipsoid	G_begin_zone_area_on_ellipsoid()
area in coordinate units	G_planimetric_polygon_area()
cellarea in specified row	G_area_of_cell_at_row()
area in square meters of polygon	G_area_of_polygon()
getarea info from id	Vtwo_get_area()
area of lathyphenlong polygon	G_ellipsoid_polygon_area()

find whicharea point is in	dig_point_to_area()
get defining points forarea polygon	Vect_get_area_points()
get number ofareas in vector map	Vtwo_num_areas()
earth toarray parenlefteastparenright	D_u_to_a_col()
earth toarray parenleftnorthparenright	D_u_to_a_row()
lookup anarray of colors	G_lookup_colors()
array to screen parenleftcolumnparenright	D_a_to_d_col()
array to screen parenleftrowparenright	D_a_to_d_row()
screen toarray parenleftxparenright	D_d_to_a_col()
screen toarray parenleftyparenright	D_d_to_a_row()
convert line_pnts structure to xyarrays	Vect_copy_pnts_to_xy()
convert xyarrays to line_pnts structure	Vect_copy_xy_to_pnts()
easting toASCII	G_format_easting()
northing toASCII	G_format_northing()
resolution toASCII	G_format_resolution()
ASCII easting to double	G_scan_easting()
ASCII northing to double	G_scan_northing()
ASCII resolution to double	G_scan_resolution()
ask a yesslashno question	G_yes()
makeaspect colors	G_make_aspect_colors()
assignslashretrieve current map region	D_check_map_window()
getattribute number of arc	Vtwo_line_att()
getattribute number of area	Vtwo_area_att()
begin area calculations	G_begin_ellipsoid_polygon_area()
begin area calculations for ellipsoid	G_begin_zone_area_on_ellipsoid()
begin cell area calculations	G_begin_cell_area_calculations()
begin distance calculations	G_begin_distance_calculations()
begin geodesic distance	G_begin_geodesic_distance()
begin polygon area calculations	G_begin_polygon_area_calculations()
bottom of screen	R_screen_bot()
getbounding box of arc	Vtwo_get_line_bbox()
getbounding box of area	Vtwo_get_area_bbox()
fill abox	R_box_abs()
fill abox	R_box_rel()
get mouse location using abox	R_get_location_with_box()
get boundingbox of arc	Vtwo_get_line_bbox()
get boundingbox of area	Vtwo_get_area_bbox()
Bresenham line algorithm	G_bresenham_line()
allocate a rasterbuffer	G_allocate_cell_buf()
zero a rasterbuffer	G_zero_cell_buf()
begin cell areacalculations	G_begin_cell_area_calculations()
begin distancecalculations	G_begin_distance_calculations()
begin areacalculations	G_begin_ellipsoid_polygon_area()
begin polygon areacalculations	G_begin_polygon_area_calculations()
begin areacalculations for ellipsoid	G_begin_zone_area_on_ellipsoid()
initializecalculations for sphere	G_begin_zone_area_on_sphere()
turns off interactivecapability	G_disable_interactive()

querycartographic projection	G_database_projection_name()
querycartographic projection	G_projection()
querycartographic zone	G_zone()
convert string to lowercase	G_tolcase()
convert string to uppercase	G_toucase()
get acategory color	G_get_color()
set acategory color	G_set_color()
read rastercategory file	G_read_cats()
read vectorcategory file	G_read_vector_cats()
write rastercategory file	G_write_cats()
write vectorcategory file	G_write_vector_cats()
get acategory label	G_get_cat()
set acategory label	G_set_cat()
get title fromcategory structure	G_get_cats_title()
initializecategory structure	G_init_cats()
set title incategory structure	G_set_cats_title()
freecategory structure memory	G_free_cats()
begincell area calculations	G_begin_cell_area_calculations()
cell area in specified row	G_area_of_cell_at_row()
adjustcell header	G_adjust_Cell_head()
random query ofcell stats	G_find_cell_stat()
freecell stats	G_free_cell_stats()
initializecell stats	G_init_cell_stats()
retrieve sortedcell stats	G_next_cell_stat()
resetslashrewindcell stats	G_rewind_cell_stats()
add data tocell stats	G_update_cell_stats()
change ctrlhyphenc message	V_intrpt_msg()
change error handling	G_set_error_routine()
change raster map title	G_put_cell_title()
change to hardware color	D_lookup_colors()
return interruptchar	G_intr_char()
printable version of controlcharacter	G_unctrl()
check for legal database file names	G_legal_filename()
create a protectedchild process	G_fork()
choose font	R_font()
clear frame display lists	D_clear_window()
clears information about current frame	D_clear_window()
clip coordinates to window	D_clip()
set textclipping frame	R_set_window()
setclipping window	D_set_clip_window()
setclipping window to map window	D_set_clip_window_to_map_window()
close a raster file	G_close_cell()
close a vector map	Vect_close()
draw aclosed polygon	R_polygon_abs()
draw aclosed polygon	R_polygon_rel()
find which arc point isclosest to	dig_point_to_line()
change to hardwarecolor	D_lookup_colors()

get a categorycolor	G_get_color()
set a categorycolor	G_set_color()
selectcolor	R_color()
define singlecolor	R_reset_color()
selectcolor	R_RGB_color()
select standardcolor	R_standard_color()
select rastercolor for line	D_color()
color name to number	D_translate_color()
makecolor ramp	G_make_ramp_colors()
getcolor range	G_get_color_range()
initializecolor structure	G_init_colors()
freecolor structure memory	G_free_colors()
read map layercolor table	G_read_colors()
write map layercolor table	G_write_colors()
select fixedcolor table	R_color_table_fixed()
select floatingcolor table	R_color_table_float()
reset rastercolor value	D_reset_color()
makecolor wave	G_make_wav e_colors()
verify a range ofcolors	D_check_colormap_size()
setcolors	G_add_color_rule()
lookup an array ofcolors	G_lookup_colors()
make aspectcolors	G_make_aspect_colors()
make greencommayellowcommaredcolors	G_make_gyr_colors()
make histogramhyphenstretched greycolors	G_make_histogram_eq_colors()
make rainbowcolors	G_make_rainbow_colors()
make randomcolors	G_make_random_colors()
make redcommayellowcommagreencolors	G_make_ryg_colors()
define multiplecolors	R_reset_colors()
establish rastercolors for graphics	D_set_colors()
setcolors in driver	D_reset_colors()
array to screenparenleftcolumnparenright	D_a_to_d_col()
easting tocolumn	G_easting_to_col()
column to easting	G_col_to_easting()
number ofcolumns in active region	G_window_cols()
run a shell levelcommand	G_system()
parsecommand line	G_parser()
command line helpslashusage message	G_usage()
addcommand to frame display list	D_add_to_list()
print percentcomplete messages	G_percent()
concatenate strings	G_strcat()
configure raster overlay mode	D_set_overlay_mode()
configure rowio structure	rowio_setup()
radius ofconformal tangent sphere	G_radius_of_conformal_tangent_sphere()
define screenconstant	V_const()
unset any vector readconstraints	Vect_remove_constraints()
print warning message andcontinue	G_warning()
printable version ofcontrol character	G_unctrl()

add newcontrol point	I_new_control_point()
read groupcontrol points	I_get_control_points()
write groupcontrol points	I_put_control_points()
conversion to meters	G_database_units_to_meters_factor()
initializeconversions	D_do_conversions()
convert line_pnts structure to xy arrays	Vect_copy_pnts_to_xy()
convert string to lower case	G_tolcase()
convert string to upper case	G_toucase()
convert xy arrays to line_pnts structure	Vect_copy_xy_to_pnts()
area incoordinate units	G_planimetric_polygon_area()
retrieve current framecoordinates	D_get_screen_window()
plot line between latloncoordinates	G_plot_line()
clipcoordinates to window	D_clip()
copy Ref lists	I_transfer_group_ref_file()
copy string to allocated memory	G_store()
copy strings	G_strcpy()
copy strings	G_strncpy()
copy vector header struct data	Vect_copy_head_data()
create a lock	lock_file()
create a protected child process	G_fork()
create new graphics frame	D_new_window()
create new initialized line points structure	Vect_new_line_struct()
allowctrlhyphenc	V_intrpt_ok()
changectrlhyphenc message	V_intrpt_msg()
get a line of input parenleftdetectctrlhyphenzparenright	G_gets()
current date and time	G_date()
clears information aboutcurrent frame	D_clear_window()
erasecurrent frame	D_erase_window()
outlinescurrent frame	D_show_window()
retrievecurrent frame coordinates	D_get_screen_window()
resetscurrent frame position	D_reset_screen_window()
identifycurrent graphics frame	D_get_cur_wind()
setcurrent graphics frame	D_set_cur_wind()
movecurrent location	R_move_abs()
movecurrent location	R_move_rel()
current location directory	G_location_path()
current location name	G_location()
assignslashretrievecurrent map region	D_check_map_window()
current mapset name	G_mapset()
givecurrent time to frame	D_timestamp()
meridional radius ofcurvature	G_meridional_radius_of_curvature()
transverse radius ofcurvature	G_transverse_radius_of_curvature()
copy vector header structdata	Vect_copy_head_data()
adddata to cell stats	G_update_cell_stats()
top leveldatabase directory	G_gisdbase()
prompt for existingdatabase file	G_ask_in_mapset()

prompt for newdatabase file	G_ask_new()
prompt for existingdatabase file	G_ask_old()
find adatabase file	G_find_file()
open a newdatabase file	G_fopen_new()
open a newdatabase file	G_open_new()
remove adatabase file	G_remove()
rename adatabase file	G_rename()
open adatabase file for reading	G_fopen_old()
open adatabase file for reading	G_open_old()
open adatabase file for update	G_fopen_append()
open adatabase file for update	G_open_update()
check for legaldatabase file names	G_legal_filename()
read thedatabase region	G_get_window()
write thedatabase region	G_put_window()
database units	G_database_unit_name()
currentdate and time	G_date()
deallocate line points structure	Vect_destroy_line_struct()
set number ofdecimal places	V_float_accuracy()
read thedefault region	G_get_default_window()
define multiple colors	R_reset_colors()
define screen constant	V_const()
define screen question	V_ques()
define single color	R_reset_color()
getdefining points for area polygon	Vect_get_area_points()
initialize screendescription	V_clear()
get filedescriptor	rowio_fileno()
get a line of inputparenleftdetect ctrlhyphenzparenright	G_gets()
top level programdirectory	G_gisbase()
top level databasedirectory	G_gisdbase()
userquoterights homedirectory	G_home()
current locationdirectory	G_location_path()
force pending updates todisk	rowio_flush()
flush pending updates todisk	segment_flush()
add command to framedisplay list	D_add_to_list()
add raster map name todisplay list	D_set_cell_name()
clear framedisplay lists	D_clear_window()
begin geodesicdistance	G_begin_geodesic_distance()
geodesicdistance	G_geodesic_distance()
geodesicdistance	G_geodesic_distance_lon_to_lon()
begindistance calculations	G_begin_distance_calculations()
distance in meters	G_distance()
set geodesicdistance latone	G_set_geodesic_distance_lat()
set geodesicdistance lattwo	G_set_geodesic_distance_lat()
finddistance of point to line	dig_check_dist()
does group existquestion	I_find_group()
draw a series ofdots	R_polydots_abs()

draw a series ofdots	R_polydots_rel()
ASCII easting todouble	G_scan_easting()
ASCII northing todouble	G_scan_northing()
ASCII resolution todouble	G_scan_resolution()
draw a closed polygon	R_polygon_abs()
draw a closed polygon	R_polygon_rel()
draw a raster	R_raster()
draw a raster	R_RGB_raster()
draw a series of dots	R_polydots_abs()
draw a series of dots	R_polydots_rel()
draw an open polygon	R_polyline_abs()
draw an open polygon	R_polyline_rel()
draw line	R_cont_abs()
draw line	R_cont_rel()
set colors indriver	D_reset_colors()
earth to array parenlefteastparenright	D_u_to_a_col()
earth to array parenleftnorthparenright	D_u_to_a_row()
earth to screen parenlefteastparenright	D_u_to_d_col()
earth to screen parenleftnorthparenright	D_u_to_d_row()
screen toearth parenleftxparenright	D_d_to_u_col()
screen toearth parenleftyparenright	D_d_to_u_row()
earth to arrayparenlefteastparenright	D_u_to_a_col()
earth to screenparenlefteastparenright	D_u_to_d_col()
returnseast larger than west	G_adjust_easting()
adjusteast longitude	G_adjust_east_longitude()
column toeasting	G_col_to_easting()
easting to ASCII	G_format_easting()
easting to column	G_easting_to_col()
ASCIIeasting to double	G_scan_easting()
shortest way betweeneastings	G_shortest_way()
xcommay toeastcommanorth	G_plot_where_en()
eastcommanorth to xcommay	G_plot_where_xy()
screen leftedge	R_screen_left()
screen rightedge	R_screen_rite()
begin area calculations forellipsoid	G_begin_zone_area_on_ellipsoid()
getellipsoid by name	G_get_ellipsoid_by_name()
getellipsoid parameters	G_get_ellipsoid_parameters()
returnellopsoid name	G_ellipsoid_name()
query GRASSenvironment variable	G_ _getenv()
query GRASSenvironment variable	G_getenv()
set GRASSenvironment variable	G_ _setenv()
set GRASSenvironment variable	G_setenv()
erase current frame	D_erase_window()
erase screen	R_erase()
sleep onerrorquestion	G_sleep_on_error()
changeerror handling	G_set_error_routine()
reset normalerror handling	G_unset_error_routine()

printerror message and exit	G_fatal_error()
establish raster colors for graphics	D_set_colors()
does groupexistquestion	I_find_group()
prompt forexisting database file	G_ask_in_mapset()
prompt forexisting database file	G_ask_old()
prompt for anexisting group	I_ask_group_old()
prompt forexisting raster file	G_ask_cell_in_mapset()
prompt forexisting raster file	G_ask_cell_old()
open anexisting raster file	G_open_cell_old()
prompt forexisting site list file	G_ask_sites_in_mapset()
prompt forexisting site list file	G_ask_sites_old()
open anexisting site list file	G_fopen_sites_old()
prompt for anexisting vector file	G_ask_vector_in_mapset()
prompt for anexisting vector file	G_ask_vector_old()
open anexisting vector file	G_fopen_vector_old()
openexisting vector map	Vect_open_old()
print error message andexit	G_fatal_error()
get textextents	R_get_text_box()
plotfparenlefteastoneparenright to fparenlefteasttwoparenright	G_plot_fx()
plot fparenlefteastoneparenright tofparenlefteasttwoparenright	G_plot_fx()
prompt for existing rasterfile	G_ask_cell_in_mapset()
prompt for new rasterfile	G_ask_cell_new()
prompt for existing rasterfile	G_ask_cell_old()
prompt for existing databasefile	G_ask_in_mapset()
prompt for new databasefile	G_ask_new()
prompt for existing databasefile	G_ask_old()
prompt for existing site listfile	G_ask_sites_in_mapset()
prompt for new site listfile	G_ask_sites_new()
prompt for existing site listfile	G_ask_sites_old()
prompt for an existing vectorfile	G_ask_vector_in_mapset()
prompt for a new vectorfile	G_ask_vector_new()
prompt for an existing vectorfile	G_ask_vector_old()
close a rasterfile	G_close_cell()
find a rasterfile	G_find_cell()
find a databasefile	G_find_file()
find a vectorfile	G_find_vector()
find a vectorfile	G_find_vector()
open a new databasefile	G_fopen_new()
open a new site listfile	G_fopen_sites_new()
open an existing site listfile	G_fopen_sites_old()
open a new vectorfile	G_fopen_vector_new()
open an existing vectorfile	G_fopen_vector_old()
read a rasterfile	G_get_map_row()
read site listfile	G_get_site()
reclassfilequestion	G_is_reclass()

open an existing rasterfile	G_open_cell_old()
open a new databasefile	G_open_new()
write site listfile	G_put_site()
read raster categoryfile	G_read_cats()
read raster historyfile	G_read_history()
read vector categoryfile	G_read_vector_cats()
remove a databasefile	G_remove()
rename a databasefile	G_rename()
unopen a rasterfile	G_unopen_cell()
write raster categoryfile	G_write_cats()
write raster historyfile	G_write_history()
write raster rangefile	G_write_range()
write vector categoryfile	G_write_vector_cats()
read group REFfile	I_get_group_ref()
read subgroup REFfile	I_get_subgroup_ref()
write group REFfile	I_put_group_ref()
write subgroup REFfile	I_put_subgroup_ref()
format a segmentfile	segment_format()
get value from segmentfile	segment_get()
read row from segmentfile	segment_get_row()
put value to segmentfile	segment_put()
write row to segmentfile	segment_put_row()
getfile descriptor	rowio_fileno()
open a databasefile for reading	G_fopen_old()
open a databasefile for reading	G_open_old()
open a databasefile for update	G_fopen_append()
open a databasefile for update	G_open_update()
prompt for any validfile name	G_ask_any()
fully qualifiedfile name	G_fully_qualified_name()
returns a temporaryfile name	G_tempfile()
addfile name to Ref structure	I_add_file_to_group_ref()
check for legal databasefile names	G_legal_filename()
open a new rasterfile parenleftrandomparenright	G_open_cell_new_random()
write a rasterfile parenleftrandomparenright	G_put_map_row_random()
open a new rasterfile parenleftsequentialparenright	G_open_cell_new()
write a rasterfile parenleftsequentialparenright	G_put_map_row()
open a new rasterfile parenleftuncompressedparenright	G_open_cell_new_uncompressed()
read a rasterfile parenleftwithout maskingparenright	G_get_map_row_nomask()
fill a box	R_box_abs()
fill a box	R_box_rel()
plotfilled polygon with n vertices	G_plot_polygon()
find a database file	G_find_file()
find a raster file	G_find_cell()
find a vector file	G_find_vector()

find a vector file	G_find_vector()
find distance of point to line	dig_check_dist()
find which arc point is closest to	dig_point_to_line()
find which area point is in	dig_point_to_area()
selectfixed color table	R_color_table_fixed()
returnFlag structure	G_define_flag()
selectfloating color table	R_color_table_float()
flush graphics	R_flush()
flush pending updates to disk	segment_flush()
choosefont	R_font()
force pending updates to disk	rowio_flush()
forget a row	rowio_forget()
format a segment file	segment_format()
clears information about currentframe	D_clear_window()
erase currentframe	D_erase_window()
identify current graphicsframe	D_get_cur_wind()
create new graphicsframe	D_new_window()
remove aframe	D_remove_window()
set current graphicsframe	D_set_cur_wind()
initializeslashcreate aframe	D_setup()
outlines currentframe	D_show_window()
give current time toframe	D_timestamp()
set text clippingframe	R_set_window()
retrieve currentframe coordinates	D_get_screen_window()
add command toframe display list	D_add_to_list()
clearframe display lists	D_clear_window()
resets currentframe position	D_reset_screen_window()
graphicsframe setup	D_setup()
free allocated memory	rowio_release()
free allocated memory	segment_release()
free category structure memory	G_free_cats()
free cell stats	G_free_cell_stats()
free color structure memory	G_free_colors()
free Ref structure	I_free_group_ref()
fully qualified file name	G_fully_qualified_name()
begingeodesic distance	G_begin_geodesic_distance()
geodesic distance	G_geodesic_distance()
geodesic distance	G_geodesic_distance_lon_to_lon()
setgeodesic distance latone	G_set_geodesic_distance_lat()
setgeodesic distance lattwo	G_set_geodesic_distance_lat()
initializegis library	G_gisinit()
give current time to frame	D_timestamp()
prepare for rastergraphics	D_cell_draw_setup()
establish raster colors forgraphics	D_set_colors()
terminategraphics	R_close_driver()
flushgraphics	R_flush()
initializegraphics	R_open_driver()

initializegraphics	R_set_RGB_color()
synchronizegraphics	R_stabilize()
identify currentgraphics frame	D_get_cur_wind()
create newgraphics frame	D_new_window()
set currentgraphics frame	D_set_cur_wind()
graphics frame setup	D_setup()
queryGRASS environment variable	G_ _getenv()
queryGRASS environment variable	G_getenv()
setGRASS environment variable	G_ _setenv()
setGRASS environment variable	G_setenv()
makegreencommayellowcommared colors	G_make_gyr_colors()
make histogramhyphenstretchedgrey colors	G_make_histogram_eq_colors()
make lineargrey scale	G_make_grey_scale_colors()
prompt for newgroup	I_ask_group_new()
prompt for an existinggroup	I_ask_group_old()
readgroup control points	I_get_control_points()
writegroup control points	I_put_control_points()
doesgroup existquestion	I_find_group()
prompt for any validgroup name	I_ask_group_any()
readgroup REF file	I_get_group_ref()
writegroup REF file	I_put_group_ref()
change errorhandling	G_set_error_routine()
reset normal errorhandling	G_unset_error_routine()
change tohardware color	D_lookup_colors()
adjust cellheader	G_adjust_Cell_head()
read the rasterheader	G_get_cellhd()
write the rasterheader	G_put_cellhd()
printheader info to stdout	Vect_print_header()
copy vectorheader struct data	Vect_copy_head_data()
command linehelpslashusage message	G_usage()
makehistogramhyphenstretched grey colors	G_make_histogram_eq_colors()
read rasterhistory file	G_read_history()
write rasterhistory file	G_write_history()
initializehistory structure	G_short_history()
getHit RETURN msg	G_get_ask_return_msg()
setHit RETURN msg	G_set_ask_return_msg()
userquoterightshome directory	G_home()
get area info fromid	Vtwo_get_area()
read vector arc by specifying lineid	Vtwo_read_line()
identify current graphics frame	D_get_cur_wind()
get areainfo from id	Vtwo_get_area()
print headerinfo to stdout	Vect_print_header()
read targetinformation	I_get_target()
write targetinformation	I_put_target()
clearsinformation about current frame	D_clear_window()
initialize calculations for sphere	G_begin_zone_area_on_sphere()
initialize category structure	G_init_cats()

initialize cell stats	G_init_cell_stats()
initialize color structure	G_init_colors()
initialize conversions	D_do_conversions()
initialize gis library	G_gisinit()
initialize graphics	R_open_driver()
initialize graphics	R_set_RGB_color()
initialize history structure	G_short_history()
initialize plotting routines	G_setup_plot()
initialize range structure	G_init_range()
initialize Ref structure	I_init_group_ref()
initialize screen description	V_clear()
initialize segment structure	segment_init()
initializeslashcreate a frame	D_setup()
create newinitialized line points structure	Vect_new_line_struct()
get a line ofinput parenleftdetect ctrlhyphenzparenright	G_gets()
interact with the user	V_call()
turns offinteractive capability	G_disable_interactive()
returninterrupt char	G_intr_char()
get a categorylabel	G_get_cat()
set a categorylabel	G_set_cat()
returns eastlarger than west	G_adjust_easting()
set geodesic distancelatone	G_set_geodesic_distance_lat()
set geodesic distancelattwo	G_set_geodesic_distance_lat()
area betweenlatitudes	G_area_for_zone_on_ellipsoid()
area betweenlatitudes	G_area_for_zone_on_sphere()
plot line betweenlatlon coordinates	G_plot_line()
area oflathyphenlong polygon	G_ellipsoid_polygon_area()
read maplayer color table	G_read_colors()
write maplayer color table	G_write_colors()
removeleadingslashtraining white	G_strip()
screenleft edge	R_screen_left()
check forlegal database file names	G_legal_filename()
run a shelllevel command	G_system()
toplevel database directory	G_gisdbase()
specifylevel for opening map	Vect_set_open_level()
get openlevel of vector map	Vect_level()
toplevel module directory	G_gisbase()
lowlevel raster plotting	D_raster()
initialize gislibrary	G_gisinit()
select raster color forline	D_color()
find distance of point toline	dig_check_dist()
parse commandline	G_parser()
drawline	R_cont_abs()
drawline	R_cont_rel()
get mouse location using aline	R_get_location_with_line()
read next vectorline	Vect_read_next_line()

Bresenhamline algorithm	G_bresenham_line()
plotline between latlon coordinates	G_plot_line()
commandline helpslashusage message	G_usage()
read vector arc by specifyingline id	Vtwo_read_line()
get aline of input parenleftdetect ctrlhyphenzparenright	G_gets()
addline of text to screen	V_line()
create new initializedline points structure	Vect_new_line_struct()
deallocateline points structure	Vect_destroy_line_struct()
line to xcommay	D_cont_abs()
line to xcommay	D_cont_rel()
makelinear grey scale	G_make_grey_scale_colors()
convert xy arrays toline_pnts structure	Vect_copy_xy_to_pnts()
convertline_pnts structure to xy arrays	Vect_copy_pnts_to_xy()
add command to frame displaylist	D_add_to_list()
add raster map name to displaylist	D_set_cell_name()
prompt for existing sitelist file	G_ask_sites_in_mapset()
prompt for new sitelist file	G_ask_sites_new()
prompt for existing sitelist file	G_ask_sites_old()
open a new sitelist file	G_fopen_sites_new()
open an existing sitelist file	G_fopen_sites_old()
read sitelist file	G_get_site()
write sitelist file	G_put_site()
clear frame displaylists	D_clear_window()
copy Reflists	I_transfer_group_ref_file()
move currentlocation	R_move_abs()
move currentlocation	R_move_rel()
currentlocation directory	G_location_path()
currentlocation name	G_location()
location title	G_myname()
get mouselocation using a box	R_get_location_with_box()
get mouselocation using a line	R_get_location_with_line()
get mouselocation using pointer	R_get_location_with_pointer()
create alock	lock_file()
remove alock	unlock_file()
adjust eastlongitude	G_adjust_east_longitude()
lookup an array of colors	G_lookup_colors()
low lev el raster plotting	D_raster()
convert string tolower case	G_tolcase()
get number of areas in vectormap	Vtwo_num_areas()
get number of arcs in vectormap	Vtwo_num_lines()
close a vectormap	Vect_close()
get open level of vectormap	Vect_level()
open new vectormap	Vect_open_new()
open existing vectormap	Vect_open_old()
specify level for openingmap	Vect_set_open_level()
write out arc to vectormap	Vect_write_line()

rewind vectormap for rehyphenreading	Vect_rewind()
readmap layer color table	G_read_colors()
writemap layer color table	G_write_colors()
retrieve rastermap name	D_get_cell_name()
add rastermap name to display list	D_set_cell_name()
assignslashretrieve currentmap region	D_check_map_window()
get rastermap title	G_get_cell_title()
change rastermap title	G_put_cell_title()
set clipping window tomap window	D_set_clip_window_to_map_window()
currentmapset name	G_mapset()
read a raster file parenleftwithoutmaskingparenright	G_get_map_row_nomask()
get range min andmax	G_get_range_min_max()
free category structurememory	G_free_cats()
free color structurememory	G_free_colors()
copy string to allocatedmemory	G_store()
free allocatedmemory	rowio_release()
free allocatedmemory	segment_release()
memory allocation	G_calloc()
memory allocation	G_malloc()
memory allocation	G_realloc()
pophyphenupmenu	D_popup()
meridional radius of curvature	G_meridional_radius_of_curvature()
command line helpslashusagemessage	G_usage()
change ctrlhyphencmessage	V_intrpt_msg()
print warningmessage and continue	G_warning()
print errormessage and exit	G_fatal_error()
print percent completemessages	G_percent()
conversion tometers	G_database_units_to_meters_factor()
distance inmeters	G_distance()
area in squaremeters of polygon	G_area_of_polygon()
get rangemin and max	G_get_range_min_max()
configure raster overlaymode	D_set_overlay_mode()
getmouse location using a box	R_get_location_with_box()
getmouse location using a line	R_get_location_with_line()
getmouse location using pointer	R_get_location_with_pointer()
move current location	R_move_abs()
move current location	R_move_rel()
move to pixel	D_move_abs()
move to pixel	D_move_rel()
get Hit RETURNmsg	G_get_ask_return_msg()
set Hit RETURNmsg	G_set_ask_return_msg()
definemultiple colors	R_reset_colors()
plot filled polygon withn vertices	G_plot_polygon()
retrieve raster mapname	D_get_cell_name()
prompt for any valid filename	G_ask_any()
return ellopsoidname	G_ellipsoid_name()

fully qualified filename	G_fully_qualified_name()
get ellipsoid byname	G_get_ellipsoid_by_name()
current locationname	G_location()
current mapsetname	G_mapset()
return programname	G_program_name()
returns a temporary filename	G_tempfile()
userquoterightsname	G_whoami()
prompt for any valid groupname	I_ask_group_any()
add raster mapname to display list	D_set_cell_name()
colorname to number	D_translate_color()
add filename to Ref structure	I_add_file_to_group_ref()
check for legal database filenames	G_legal_filename()
readnext vector line	Vect_read_next_line()
resetnormal error handling	G_unset_error_routine()
earth to arrayparenleftnorthparenright	D_u_to_a_row()
earth to screenparenleftnorthparenright	D_u_to_d_row()
row tonorthing	G_row_to_northing()
northing to ASCII	G_format_northing()
ASCIInorthing to double	G_scan_northing()
northing to row	G_northing_to_row()
color name tonumber	D_translate_color()
get attributenumber of arc	Vtwo_line_att()
getnumber of arcs in vector map	Vtwo_num_lines()
get attributenumber of area	Vtwo_area_att()
getnumber of areas in vector map	Vtwo_num_areas()
number of columns in active region	G_window_cols()
setnumber of decimal places	V_float_accuracy()
number of rows in active region	G_window_rows()
read vector arc by specifyingoffset	Vone_read_line()
open a database file for reading	G_fopen_old()
open a database file for reading	G_open_old()
open a database file for update	G_fopen_append()
open a database file for update	G_open_update()
open a new database file	G_fopen_new()
open a new database file	G_open_new()
open a new raster file parenleftrandomparenright	G_open_cell_new_random()
open a new raster file parenleftsequentialparenright	G_open_cell_new()
open a new raster file parenleftuncompressedparenright	G_open_cell_new_uncompressed()
open a new site list file	G_fopen_sites_new()
open a new vector file	G_fopen_vector_new()
open an existing raster file	G_open_cell_old()
open an existing site list file	G_fopen_sites_old()
open an existing vector file	G_fopen_vector_old()
open existing vector map	Vect_open_old()
getopen level of vector map	Vect_level()

open new vector map	Vect_open_new()
draw anopen polygon	R_polyline_abs()
draw anopen polygon	R_polyline_rel()
specify level foropening map	Vect_set_open_level()
returnsOption structure	G_define_option()
outlines current frame	D_show_window()
configure rasteroverlay mode	D_set_overlay_mode()
get ellipsoidparameters	G_get_ellipsoid_parameters()
parse command line	G_parser()
forcepending updates to disk	rowio_flush()
flushpending updates to disk	segment_flush()
printpercent complete messages	G_percent()
move topixel	D_move_abs()
move topixel	D_move_rel()
set number of decimalplaces	V_float_accuracy()
plot fparenlefteastoneparenright to fparenlefteasttwoparenright	G_plot_fx()
plot filled polygon with n vertices	G_plot_polygon()
plot line between latlon coordinates	G_plot_line()
low lev el rasterplotting	D_raster()
initializeplotting routines	G_setup_plot()
add new controlpoint	I_new_control_point()
ispoint in areaquestion	dig_point_in_area()
find which arcpoint is closest to	dig_point_to_line()
find which areapoint is in	dig_point_to_area()
find distance ofpoint to line	dig_check_dist()
get mouse location usingpointer	R_get_location_with_pointer()
read group controlpoints	I_get_control_points()
write group controlpoints	I_put_control_points()
get definingpoints for area polygon	Vect_get_area_points()
create new initialized linepoints structure	Vect_new_line_struct()
deallocate linepoints structure	Vect_destroy_line_struct()
pole in polygon	G_pole_in_polygon()
area in square meters ofpolygon	G_area_of_polygon()
area of lathyphenlongpolygon	G_ellipsoid_polygon_area()
pole inpolygon	G_pole_in_polygon()
draw a closedpolygon	R_polygon_abs()
draw a closedpolygon	R_polygon_rel()
draw an openpolygon	R_polyline_abs()
draw an openpolygon	R_polyline_rel()
get defining points for areapolygon	Vect_get_area_points()
beginpolygon area calculations	G_begin_polygon_area_calculations()
plot filledpolygon with n vertices	G_plot_polygon()
pophyphenup menu	D_popup()
resets current frameposition	D_reset_screen_window()
prepare for raster graphics	D_cell_draw_setup()
print error message and exit	G_fatal_error()

print header info to stdout	Vect_print_header()
print percent complete messages	G_percent()
print warning message and continue	G_warning()
printable version of control character	G_unctrl()
create a protected childprocess	G_fork()
top levelprogram directory	G_gisbase()
returnprogram name	G_program_name()
query cartographicprojection	G_database_projection_name()
query cartographicprojection	G_projection()
prompt for a new vector file	G_ask_vector_new()
prompt for an existing group	I_ask_group_old()
prompt for an existing vector file	G_ask_vector_in_mapset()
prompt for an existing vector file	G_ask_vector_old()
prompt for any valid file name	G_ask_any()
prompt for any valid group name	I_ask_group_any()
prompt for existing database file	G_ask_in_mapset()
prompt for existing database file	G_ask_old()
prompt for existing raster file	G_ask_cell_in_mapset()
prompt for existing raster file	G_ask_cell_old()
prompt for existing site list file	G_ask_sites_in_mapset()
prompt for existing site list file	G_ask_sites_old()
prompt for new database file	G_ask_new()
prompt for new group	I_ask_group_new()
prompt for new raster file	G_ask_cell_new()
prompt for new site list file	G_ask_sites_new()
create aprotected child process	G_fork()
put value to segment file	segment_put()
fullyqualified file name	G_fully_qualified_name()
query cartographic projection	G_database_projection_name()
query cartographic projection	G_projection()
query cartographic zone	G_zone()
query GRASS environment variable	G_ _getenv()
query GRASS environment variable	G_getenv()
randomquery of cell stats	G_find_cell_stat()
ask a yesslashnoquestion	G_yes()
define screenquestion	V_ques()
radius of conformal tangent sphere	G_radius_of_conformal_tangent_sphere()
meridionalradius of curvature	G_meridional_radius_of_curvature()
transverseradius of curvature	G_transverse_radius_of_curvature()
makerainbow colors	G_make_rainbow_colors()
make colorramp	G_make_ramp_colors()
open a new raster fileparenleftrandomparenright	G_open_cell_new_random()
write a raster fileparenleftrandomparenright	G_put_map_row_random()
makerandom colors	G_make_random_colors()
random query of cell stats	G_find_cell_stat()
get colorrange	G_get_color_range()
read rasterrange	G_read_range()

write rasterrange file	G_write_range()
getrange min and max	G_get_range_min_max()
verify arange of colors	D_check_colormap_size()
initializerange structure	G_init_range()
updaterange structure	G_row_update_range()
updaterange structure	G_update_range()
draw araster	R_raster()
draw araster	R_RGB_raster()
allocate araster buffer	G_allocate_cell_buf()
zero araster buffer	G_zero_cell_buf()
readraster category file	G_read_cats()
writeraster category file	G_write_cats()
selectraster color for line	D_color()
resetraster color value	D_reset_color()
establishraster colors for graphics	D_set_colors()
prompt for existingraster file	G_ask_cell_in_mapset()
prompt for newraster file	G_ask_cell_new()
prompt for existingraster file	G_ask_cell_old()
close araster file	G_close_cell()
find araster file	G_find_cell()
read araster file	G_get_map_row()
open an existingraster file	G_open_cell_old()
unopen araster file	G_unopen_cell()
open a newraster file parenleftrandomparenright	G_open_cell_new_random()
write araster file parenleftrandomparenright	G_put_map_row_random()
open a newraster file parenleftsequentialparenright	G_open_cell_new()
write araster file parenleftsequentialparenright	G_put_map_row()
open a newraster file parenleftuncompressedparenright	G_open_cell_new_uncompressed()
read araster file parenleftwithout maskingparenright	G_get_map_row_nomask()
prepare forraster graphics	D_cell_draw_setup()
read theraster header	G_get_cellhd()
write theraster header	G_put_cellhd()
readraster history file	G_read_history()
writeraster history file	G_write_history()
retrieveraster map name	D_get_cell_name()
addraster map name to display list	D_set_cell_name()
getraster map title	G_get_cell_title()
changeraster map title	G_put_cell_title()
configureraster overlay mode	D_set_overlay_mode()
low lev elraster plotting	D_raster()
readraster range	G_read_range()
writeraster range file	G_write_range()
render araster row	D_draw_cell()
specify types of arcs toread	Vect_set_constraint_type()

read a raster file	G_get_map_row()
read a raster file parenleftwithout maskingparenright	G_get_map_row_nomask()
read a row	rowio_get()
unset any vectorread constraints	Vect_remove_constraints()
read group control points	I_get_control_points()
read group REF file	I_get_group_ref()
read map layer color table	G_read_colors()
read next vector line	Vect_read_next_line()
read raster category file	G_read_cats()
read raster history file	G_read_history()
read raster range	G_read_range()
read row from segment file	segment_get_row()
read site list file	G_get_site()
read subgroup REF file	I_get_subgroup_ref()
read target information	I_get_target()
read the database region	G_get_window()
read the default region	G_get_default_window()
read the raster header	G_get_cellhd()
read vector arc by specifying line id	Vtwo_read_line()
read vector arc by specifying offset	Vone_read_line()
set restricted region toread vector arcs from	Vect_set_constraint_region()
read vector category file	G_read_vector_cats()
open a database file forreading	G_fopen_old()
open a database file forreading	G_open_old()
reclass filequestion	G_is_reclass()
makeredcommayellowcommagreen colors	G_make_ryg_colors()
read groupREF file	I_get_group_ref()
read subgroupREF file	I_get_subgroup_ref()
write groupREF file	I_put_group_ref()
write subgroupREF file	I_put_subgroup_ref()
copyRef lists	I_transfer_group_ref_file()
add file name toRef structure	I_add_file_to_group_ref()
freeRef structure	I_free_group_ref()
initializeRef structure	I_init_group_ref()
assignslashretrieve current mapregion	D_check_map_window()
read the defaultregion	G_get_default_window()
get the activeregion	G_get_set_window()
read the databaseregion	G_get_window()
write the databaseregion	G_put_window()
set the activeregion	G_set_window()
number of columns in activeregion	G_window_cols()
number of rows in activeregion	G_window_rows()
set restrictedregion to read vector arcs from	Vect_set_constraint_region()
align tworegions	G_align_window()
remove a database file	G_remove()
remove a frame	D_remove_window()

remove a lock	unlock_file()
remove leadingslashtraining white	G_strip()
remove unnecessary white	G_squeeze()
rename a database file	G_rename()
render a raster row	D_draw_cell()
rewind vector map forrehyphenreading	Vect_rewind()
reset normal error handling	G_unset_error_routine()
reset raster color value	D_reset_color()
resetslashrewind cell stats	G_rewind_cell_stats()
resets current frame position	D_reset_screen_window()
resolution to ASCII	G_format_resolution()
ASCIIresolution to double	G_scan_resolution()
setrestricted region to read vector arcs from	Vect_set_constraint_region()
retrieve current frame coordinates	D_get_screen_window()
retrieve raster map name	D_get_cell_name()
retrieve sorted cell stats	G_next_cell_stat()
return ellopsoid name	G_ellipsoid_name()
return Flag structure	G_define_flag()
return interrupt char	G_intr_char()
get HitRETURN msg	G_get_ask_return_msg()
set HitRETURN msg	G_set_ask_return_msg()
return module name	G_program_name()
returns a temporary file name	G_tempfile()
returns east larger than west	G_adjust_easting()
returns Option structure	G_define_option()
rewind vector map for rehyphenreading	Vect_rewind()
initialize plottingroutines	G_setup_plot()
array to screenparenleftrowparenright	D_a_to_d_row()
render a rasterrow	D_draw_cell()
cell area in specifiedrow	G_area_of_cell_at_row()
northing torow	G_northing_to_row()
forget arow	rowio_forget()
read arow	rowio_get()
write arow	rowio_put()
readrow from segment file	segment_get_row()
row to northing	G_row_to_northing()
writerow to segment file	segment_put_row()
configurerowio structure	rowio_setup()
number ofrows in active region	G_window_rows()
run a shell level command	G_system()
make linear greyscale	G_make_grey_scale_colors()
erasescreen	R_erase()
bottom ofscreen	R_screen_bot()
top ofscreen	R_screen_top()
add line of text toscreen	V_line()
array toscreen parenleftcolumnparenright	D_a_to_d_col()
definescreen constant	V_const()

initializescreen description	V_clear()
earth toscreen parenlefteastparenright	D_u_to_d_col()
screen left edge	R_screen_left()
earth toscreen parenleftnorthparenright	D_u_to_d_row()
definescreen question	V_ques()
screen right edge	graveR_screen_rite()
array toscreen parenleftrowparenright	D_a_to_d_row()
screen to array parenleftxparenright	D_d_to_a_col()
screen to array parenleftyparenright	D_d_to_a_row()
screen to earth parenleftxparenright	D_d_to_u_col()
screen to earth parenleftyparenright	D_d_to_u_row()
format asegment file	segment_format()
get value fromsegment file	segment_get()
read row fromsegment file	segment_get_row()
put value tosegment file	segment_put()
write row tosegment file	segment_put_row()
initializesegment structure	segment_init()
select color	R_color()
select color	R_RGB_color()
select fixed color table	R_color_table_fixed()
select floating color table	R_color_table_float()
select raster color for line	D_color()
select standard color	R_standard_color()
open a new raster fileparenleftsequentialparenright	G_open_cell_new()
write a raster fileparenleftsequentialparenright	G_put_map_row()
draw aseries of dots	R_polydots_abs()
draw aseries of dots	R_polydots_rel()
set a category color	G_set_color()
set a category label	G_set_cat()
set clipping window	D_set_clip_window()
set clipping window to map window	D_set_clip_window_to_map_window()
set colors	G_add_color_rule()
set colors in driver	D_reset_colors()
set current graphics frame	D_set_cur_wind()
set geodesic distance latone	G_set_geodesic_distance_lat()
set geodesic distance lattwo	G_set_geodesic_distance_lat()
set GRASS environment variable	G_ _setenv()
set GRASS environment variable	G_setenv()
set Hit RETURN msg	G_set_ask_return_msg()
set number of decimal places	V_float_accuracy()
set restricted region to read vector arcs from	Vect_set_constraint_region()
set text clipping frame	R_set_window()
set text size	R_text_size()
set the active region	G_set_window()
set title in category structure	G_set_cats_title()
graphics framesetup	D_setup()

run ashell level command	G_system()
shortest way between eastings	G_shortest_way()
definesingle color	R_reset_color()
prompt for existingsite list file	G_ask_sites_in_mapset()
prompt for newsite list file	G_ask_sites_new()
prompt for existingsite list file	G_ask_sites_old()
open a newsite list file	G_fopen_sites_new()
open an existingsite list file	G_fopen_sites_old()
readsite list file	G_get_site()
writesite list file	G_put_site()
set textsize	R_text_size()
sleep on errorquestion	G_sleep_on_error()
retrievesorted cell stats	G_next_cell_stat()
remove unnecessary white	G_squeeze()
remove leadingslashtraining white	G_strip()
deallocate line points structure	Vect_destroy_line_struct()
cell area inspecified row	G_area_of_cell_at_row()
specify level for opening map	Vect_set_open_level()
specify types of arcs to read	Vect_set_constraint_type()
read vector arc byspecifying line id	Vtwo_read_line()
read vector arc byspecifying offset	Vone_read_line()
initialize calculations forsphere	G_begin_zone_area_on_sphere()
radius of conformal tangentsphere	G_radius_of_conformal_tangent_sphere()
area insquare meters of polygon	G_area_of_polygon()
selectstandard color	R_standard_color()
random query of cellstats	G_find_cell_stat()
free cellstats	G_free_cell_stats()
initialize cellstats	G_init_cell_stats()
retrieve sorted cellstats	G_next_cell_stat()
resetslashrewind cellstats	G_rewind_cell_stats()
add data to cellstats	G_update_cell_stats()
print header info tostdout	Vect_print_header()
copystring to allocated memory	G_store()
convertstring to lower case	G_tolcase()
convertstring to upper case	G_toucase()
concatenatestrings	G_strcat()
copystrings	G_strcpy()
copystrings	G_strncpy()
copy vector headerstruct data	Vect_copy_head_data()
return Flagstructure	G_define_flag()
returns Optionstructure	G_define_option()
get title from categorystructure	G_get_cats_title()
initialize categorystructure	G_init_cats()
initialize colorstructure	G_init_colors()
initialize rangestructure	G_init_range()
update rangestructure	G_row_update_range()
set title in categorystructure	G_set_cats_title()

initialize historystructure	G_short_history()
update rangestructure	G_update_range()
add file name to Refstructure	I_add_file_to_group_ref()
free Refstructure	I_free_group_ref()
initialize Refstructure	I_init_group_ref()
configure rowiostructure	rowio_setup()
initialize segmentstructure	segment_init()
convert xy arrays to line_pntsstructure	Vect_copy_xy_to_pnts()
create new initialized line pointsstructure	Vect_new_line_struct()
free categorystructure memory	G_free_cats()
free colorstructure memory	G_free_colors()

B Newindex

(See next page.)

Index

C GNU Free Documentation License

Version 1.1, March 2000

http://www.gnu.org/copyleft/fdl.html

59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other written document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you".

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied Verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies of the Document numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as Verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly-accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- **A.** Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

- **B.** List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).

- **C.** State on the Title page the name of the publisher of the Modified Version, as the publisher.

- **D.** Preserve all the copyright notices of the Document.

- **E.** Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- **F.** Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- **G.** Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- **H.** Include an unaltered copy of this License.
- **I.** Preserve the section entitled "History", and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
- **J.** Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
- **K.** In any section entitled "Acknowledgements" or "Dedications", preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
- **L.** Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
- **M.** Delete any section entitled "Endorsements". Such a section may not be included in the Modified Version.
- **N.** Do not retitle any existing section as "Endorsements" or to conflict in title with any Invariant Section.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties–for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through

arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled "History" in the various original documents, forming one section entitled "History"; likewise combine any sections entitled "Acknowledgements", and any sections entitled "Dedications". You must delete all sections entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for Verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding Verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an "aggregate", and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document's Cover Texts

may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

FSF & GNU inquiries & questions to *gnu@gnu.org*.