GRASS 4.1 Programmer's Manual

Michael Shapiro

James Westervelt
Dave Gerdes
Marjorie Larson
Kenneth R. Brownfield

U.S. Army Construction Engineering Research Laboratory

ABSTRACT

This manual introduces the reader to the Geographic Resources Analysis Support System from the programming perspective. Design theory, system support libraries, system maintenance, and system enhancement are all presented.

March, 1993

Foreword

This work was performed by the Environmental Division (EN) of the U.S. Army Construction Engineering Research Laboratory (USACERL). Dr. William D. Goran is Acting Chief of the Environmental Sustainment Laboratory; Diane K. Mann is Acting Chief of the Environmental Compliance Division.

COL Daniel Waldo, Jr. is Commander and Director of USACERL, and Dr. L.R. Shaffer is Technical Director.

Chapter 1

Introduction

1.1. Background

The Geographic Resources Analysis Support System (GRASS) is a geographic information system (GIS) designed and developed by researchers at the U.S. Army Construction Engineering Research Laboratory (USACERL). 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, 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 USACERL still maintain and support GRASS, and still develop and organize new versions of GRASS for release, programmers at numerous sites now work directly with GRASS source code.

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. Software enhancements are developed at various sites, and submitted to USACERL to be shared with other sites and included in future releases of GRASS. Improvements to the code are periodically incorporated into new releases (which occur approximately once per year).

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.

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 and 4.1.

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.

1.4. Scope

Information in this manual is valid for GRASS version 4.1, released in May, 1991. 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 in the newsletter *GRASSClippings* and other GRASS information forums.

1.5. Mode of Technology Transfer

Army and Corps of Engineer organizations can acquire GRASS software from USACERL. Several other 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 Information Center.[1]

[1] See *§1.6 GRASS Information Center* [*p. 4*] for phone numbers and mail addresses.

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,

The GRASS Inter-Agency Coordinating Committee,

An annual GRASS/GIS User Group Meeting,

GRASSClippings , a periodic newsletter, and

GRASSNET, an electronic mail 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 the newsletter *GRASSClippings* , and (4) updated information on the status of GRASS user group meetings and software releases.

The **GRASS Inter-Agency Coordinating Committee** is an informal organization with members from government agencies and other organizations that use, support and enhance GRASS. This organization sponsors the annual User Group Meeting and the quarterly newsletter. It holds at least two meetings annually to share and coordinate GRASS plans among the participating agencies.

The annual **GRASS/GIS User Group Meeting** is hosted each year 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.

The **GRASSClippings** newsletter is published to provide information to anyone interested in GRASS software. The newsletter includes articles on software development, hardware options and applications of GRASS.

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 Arpanet, Internet and other networks. GRASSNET also includes a library of contributed software available for users to retrieve and review. Thus, new software is available before it is integrated into a formal release of GRASS code.

 - 4 -

1.6. GRASS Information Center

Sites wishing to contribute code to GRASS, or wanting to participate in any of these GRASS/GIS user community forums, should contact the GRASS Information Center by phone at: (800)-USA-CERL, extension 220 or (217)-373-7220; by U.S. mail at: GRASS Information Center, USACERL, P.O. Box 9005, Champaign, IL, 61826-9005; or by electronic mail at: grass@zorro.cecer.army.mil .

Chapter 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 programs 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. The programmer will find some routines in the libraries which are specifically designed to support this projection.

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.

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 computer. 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 microcomputers in the under-$10,000 range. Higher-cost equipment should be necessary only for providing faster analyses, and more disk and memory space.

PORTABLE -

This system is intended to be as portable as possible. At the November 1986 User Group meeting, groups interested in GRASS resoundingly stated that portability was the number one concern, ranking firmly above speed and user friendliness. GRASS code must run on a wide variety of hardware configurations.

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 accommodate both AT&T (System 5) and Berkeley (BSD) UNIX.

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 programs 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. Programs developed at this level may be specifically designed for one hardware arrangement.

Command Level -

This is the level most used. Using the user's login shell, GRASS commands are made available through modification of the **PATH** variable. Help and on-line manual commands are available.

In version 2.0, GRASS programs included both user interface and program function capabilities and were highly interactive. GRASS 3.0 introduced complementary command-line versions of these functions in which the information required by the program 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.

In GRASS 4.1 the interactive and command-line versions of a program have been "merged" into a single program (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 Programs* [*p. 57*]. 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 program 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, and support both AT&T and Berkeley UNIX.

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 at USACERL. 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.

2.3. Documentation Standards

GRASS is a public domain system. While such systems are usually 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 programs 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 program.

On-line manual -

Manual entries in the style of the UNIX manual entries will also be available to the user.

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] Final documents have been kept separate from the GRASS directories, though it is suggested that they appear with appropriate "makefiles" under $GISBASE/tutorials.[2]

[1] This package, invoked with the -ms option to nroff, is documented in section 7 of the UNIX manual.

[2] $GISBASE is the directory where GRASS is installed. See *§10.1 UNIX Environment* [*p. 53*] for details.

Chapter 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. 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 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 program to program. All GRASS programs 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 programs 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 programs 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 *§25 Writing GRASS Shell Scripts* [*p. 283*].

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 Programs* [*p. 57*].

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 program, the programmer, using existing library functions, can concentrate primarily on the analysis algorithms of the new tool. Such programs 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 programs use routines from this library. See *§12 GIS Library* [*p. 69*].

Vector Library. While GRASS is primarily a raster map analysis and display system, it also has some 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. 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

[1] Known as *Gmake* under GRASS 3.0 and *gmake4.1* under GRASS 4.1.

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. 157*].

Segment Library. For programs 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. 221*].

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) and ctrl-k keys, moves from prompt to prompt, filling an answer into each field. When the ESC (escape) key is struck, the answers are provided to the program for analysis. Users have found this interface pleasant and consistent. See *§20 Vask Library* [*p. 229*].

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, programs using this library communicate (through fifo files) with another program which translates the graphics commands into graphics on the desired device. Each time the program 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. 195*] and *§15 Raster Graphics Library* [*p. 183*].

3.3. Driver Programmer

GRASS programs are written to be portable. To this end, a tremendous amount of modularity is designed into the system. Throughout its development, GRASS programs have become increasingly specialized. The original monolithic approach continues to fragment into ever smaller pieces. Smaller pieces will allow future developers and users ever 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) program. 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 program, and each program understands the graphics calls of the application program. Porting of GRASS to a new system primarily means the development of one new graphics driver. See *§23 Writing a Graphics Driver* [*p. 263*].

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 programs. See *§22 Writing a Digitizer Driver* [*p. 251*].

Similarly, GRASS sites may wish to write code to support different hardcopy color printers (inkjet, thermal, etc.). See *§24 Writing a Paint Driver* [*p. 271*].

3.4. GRASS System Designer

To date, GRASS system design has been done at one location: USACERL. One, and only one site must be responsible for the design of the system at the database and fundamental library level. As the software is public domain, 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 staff at USACERL**.

 - 14 -

Chapter 4

Database Structure

This section 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. 69*]. 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 programs 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.[1] GRASS programs access this information via routines in the *GIS Library.* The variables that specify the database are described briefly below; see *§10 Environment Variables* [*p. 53*] 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.[2] 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*(*p. 73*), must be used instead.

[1] The file in the user's home directory is called

[2] using *g.gisenv*; see *§25 Writing GRASS Shell Scripts* [*p. 283*]

 - 16 -

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*(*p. 73*).

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*(*p. 72*).

```
                      GISDBASE
                         |
 ------------------------------------------------
    |             |             |           |
location.1    location.2    location.3     ...
```

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 programs 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*(*p. 72*).

```
                           LOCATION
                              |
 ----------------------------------------------------------------
    |            |            |          |           |
mapset.1     mapset.2     mapset.3      ...      PERMANENT
```

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. 20*] 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*(*p. 73*).

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. 19*].

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 /.

```
                              MAPSET
                                 |
 ------------------------------------------------------------------
        |               |        |       |        |          |      |
   SEARCH_PATH        WIND     cats/   cell/    paint/   windows/   ...
```

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.[3] 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*(*p. 83*). See *§9.1 Region* [*p. 49*] for a discussion of the GRASS region.

[3] 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, have not been renamed. Thus WIND now contains the current 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. 20*] 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.[4] 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 programs 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 diverse 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.

4 See *§12.5.7 Creating and Opening a New Database File* [*p. 80*].

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

element	function
cell	binary raster file
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
hist	history information for raster maps
dig	binary vector data
dig_ascii	ascii vector data
dig_att	vector attribute support
dig_cats	vector category label support
dig_plus	vector topology support
reg	digitizer point registration
bdlg	binary dlg files
dlg	ascii dlg files
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 programs
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 routines 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 two special files that are only found in this mapset. These files are MYNAME and DEFAULT_WIND and are never modified by GRASS software.

MYNAME contains a single line descriptive name for the location. This name is returned by the routine *G_myname*(*p. 72*).

DEFAULT_WIND contains the default region for the location. The contents of this file are returned by *G_get_default_window*(*p. 84*). 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.

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. 16*]).

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. 20*]) unless prohibited by normal UNIX file permissions (see *§4.7.2 UNIX File Permissions* [*p. 20*]).

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.[5] 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.

Warning. Since the PERMANENT mapset contains global database information, all

[5] 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.

 - 21 -

users must have read and search access to the PERMANENT mapset directory.[6] Do not remove the read and search permissions from PERMANENT.

[6] 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.

- 22 -

Chapter 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 4.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
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. 16*] 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. 98*].

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** 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 $\boxed{0|3|100}$

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 *4.1 format*[4] raster files will transfer between

---

[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 4.0.

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] The *pre-3.0 compressed* format is not stored in a machine-independent format, and cannot generally be used for intermachine transfer, unless the two machines have the same integer and long integer format.

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 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. 106*].

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 *proj*ection 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

Others may be added in the future. The *zone* field is the projection zone. In the

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

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

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.**

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.[7]

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.

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)
```

development.

[7] These fields were added for 4.0. Under 3.0 the number of rows and columns were calculated from the other values in the header. However, this sometimes resulted in wrong results since the resolution values could not be stored with sufficient accuracy in ascii format. 4.0 used the resolution fields only if the row and column fields are not present (i.e., 3.0 format header files).

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. 108*].

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[8] 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[9] this method of storing color information is clearly untenable.

GRASS 4.1 solves the above problem 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 4.1 color file:

4.1 color table file

% 1387 1801		
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 1525-1594*
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.[10] The color for category 1400 is calculated from the colors for categories 1387 and 1456:

---

[8] See the *GRASS 3.0 Programmer's Manual,* for details on the 3.0 color file.

[9] These values are for 32-bit architectures.

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

red	=	interpolate(255,170)	=	239
green	=	interpolate(85,170)	=	101
blu	=	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:

4.1 color table file

```
% 1387 1801
1387:0     1801:255
```

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

4.1 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. 111*].

5.6. Raster History File

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 program 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* program allows the user to view this information, and the *r.support* program 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 *§12.10.4 Raster History File* [*p. 116*].

5.7. Raster Range File

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 *§12.10.5 Raster Range File* [*p. 117*].

- 32 -

Chapter 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 nonintersecting 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.* Two 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. 23*] 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

[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.

paste vect.xfig diagram here

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
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.

Note. Vector files are also called *digit* files, since they are created and modified by the GRASS digitizing program *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. 157*].

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]

[6] Database directories are also called *elements.* See *§4.4 Mapsets* [*p. 16*] for a description of database elements.

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

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
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

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: US Army CERL
DIGIT DATE:   03/18/88
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:
```

* Currently, GRASS programs 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* program. If the data comes from outside GRASS, this field can be set to 0.0.

 - 36 -

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.

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* program 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.

---

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

[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).*

6.3. Vector Category Attribute File

As was mentioned in *§6.1 What is a Vector Map Layer?* [p. 33], 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*)*
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 programs 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 program *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 *§5.4 Raster Category File Format* [p. 28], and the reader is referred to that section for details.

Note. The program *v.support* allows the user to enter and modify the vector category labels. The program *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 *§12.11.6 Vector Category File* [p. 125].

6.5. Vector Index and Pointer File

The *dig_plus* file contains information that accelerates vector queries. It is created by the program *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* program 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:

 - 39 -

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 program in GRASS.

6.7. Vector Topology Rules

The following rules apply to the vector data:

1 Arcs should not cross each other (i.e., arcs which would cross must be split at their intersection to form distinct arcs).

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.

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. 34], is required.

dig_att

The *dig_att* file, described in *§6.3 Vector Category Attribute File* [p. 37], 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

 - 40 -

sophisticated vector query.

dig_cats

The *dig_cats* file, described in *§6.4 Vector Category Label File* [*p. 38*], 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. 38*], is created by the GRASS program *v.import* when converting the *dig_ascii* file to the binary *dig* file.

Chapter 7

Point Data: Site List Files

This section 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 *s.menu*[1] program, which was developed as an application within GRASS to aid in archeological site predictive modeling. The *site list* files were designed for use by this program, but have since become the principal data structure for point data.[2]

7.2. Site File Format

Site files are ascii files stored under the **site_lists** database element.[3] The format of a site file is best explained by example:

> **name**|sample
> **desc**|sample site list
> 728220|5182440|site 27
> 727060|5181710|site 28
> 725500|5184000|site 29
> 719800|5187200|site 30

name

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

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

[1] The *GRASS User's Reference Manual* contains a complete description of the *s.menu* capability.

[2] Other GRASS programs which read site lists include *s.out.ascii, d.sites* and *p.map*.

[3] See *§4.5.2 Elements* [*p. 18*] for an explanation of database elements.

desc

This line contains a description of the site list file, and is printed on all the reports generated by the *s.menu* program. 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:[4]

east|north|description

The *east* and *north* fields represent the geographic coordinates (easting and northing) of the site. The *description* field provides a one line text description (label) of the site, and is optional.

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 *§12.12 Site List Processing* [*p. 126*] and the programmer should refer to that section for details.

[4] The pipe character ("|") is sometimes used to separate the fields in the records, but sometimes blank spaces are used.

Chapter 8

Image Data: Groups

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 programs[1] which extract image data from magnetic tape extract the band data into 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

[1] See *§8.4 Imagery Programs* [p. 47] for a list of the major GRASS imagery programs.

(3) A place to store spectral signatures, image statistics, etc., which are needed by image classification procedures.

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 to produce a landcover map.

---

[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, if the image is being registered to another image.

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 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.

```
                              group/
                                |
 ------------------------------------------------------------------
    |             |              |              |           |
mss.may80/   nhap.jun88/   nhap.oct88/    tm.apr88/        ...
```

```
                     tm.apr88/
                         |
 ---------------------------------------------
  |          |            |             |
 REF      POINTS       TARGET       subgroup/
```

In this example, the groups are named *mss.may80*, *nhap.jun88*, 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:

```
tm.apr88.1    grass
tm.apr88.2    grass
tm.apr88.3    grass
tm.apr88.4    grass
tm.apr88.5    grass
tm.apr88.7    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.apr88.1*, *tm.apr88.2*, *tm.apr88.3*, *tm.apr88.4*, *tm.apr88.5* and *tm.apr88.7* in mapset *grass*. (Presumably these are bands 1-5 and 7 from an April 88 Landsat Thematic Mapper image.)

[4] See *§4.5.2 Elements* [*p. 18*] for an explanation of database elements.

[5] The group names are chosen by the user.

 - 46 -

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* program. 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.)

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:

---

[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.

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

```
                 subgroup/
                     |
 ------------------------------------------
   |             |             |        |
  123/          234/          1357/    ...
```

```
        1357/
          |
 ----------------------
   |                |
  REF              sig/
                    |
           ---------------------
             |               |
          cluster.1       cluster.2
```

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.apr88.1	grass
tm.apr88.3	grass
tm.apr88.5	grass
tm.apr88.7	grass

indicating that the subgroup is composed of bands 1, 3, 5, and 7 from the April 1988 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 program *i.cluster*.

8.4. Imagery Programs

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

---

[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 extraction

i.tape.mss	Landsat Multispectral Scanner data
i.tape.tm	Landsat Thematic Mapper data
i.tape.other	other formats, such as scanned aerial photography or SPOT satellite data

image rectification

i.points	image registration (assign control points)
i.rectify	image rectification
i.target	establish target database for the group

image classification

i.cluster	unsupervised clustering
i.maxlik	maximum likelihood classifier

other

i.group	group management

8.5. Programming Interface for Groups

The programming interface to the group data is described in *§14 Imagery Library* [*p. 173*] and the reader is referred to that chapter for details.

Chapter 9

Region and Mask

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 programs automatically retrieve only data that fall within the region. Furthermore, if there is a mask, only data that fall within the mask are retained. Programs 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.[1]

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,[2] 3 is Latitude Longitude.[3] 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. 26*] for details about raster header files.

---

[1] These fields were not present in GRASS 3.0

[2] State Plane is not yet fully supported in GRASS.

[3] Latitude Longitude is a nonplanimetric projection and is only partially supported in GRASS.

9.2. Mask

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.[4] 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:

$$\begin{array}{ccccc} \text{input} & & \text{MASK} & & \text{output} \\ \begin{array}{|c|c|c|} \hline 3 & 4 & 4 \\ \hline 3 & 3 & 4 \\ \hline 2 & 3 & 3 \\ \hline \end{array} & + & \begin{array}{|c|c|c|} \hline 0 & 1 & 1 \\ \hline 1 & 1 & 0 \\ \hline 1 & 0 & 0 \\ \hline \end{array} & = & \begin{array}{|c|c|c|} \hline 0 & 4 & 4 \\ \hline 3 & 3 & 0 \\ \hline 2 & 0 & 0 \\ \hline \end{array} \end{array}$$

9.3. Variations

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

[4] The *r.mask* program 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.

- 52 -

Chapter 10

Environment Variables

GRASS programs are written to be independent of which database the user is using, where the database resides on the disk, or where the programs themselves reside. When programs 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 *grass4.1* sets the following UNIX environment variables:[1]

GISBASE	top level directory for the GRASS programs
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 */grass,* then GISBASE would be set to */grass.* The command directory would be */grass/bin,* the command support directory would be */grass/etc,* the source code directory would be */grass/src,* the on-line manual would live in */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*(*p. 72*).

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).

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

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

variables are stored. This file is **.grassrc** in the user's home directory.

10.2. GRASS Environment

All GRASS users will have a file in their home directory named **.grassrc**[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	top level 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 programs access these variables using the *G_gisdbase*(*p. 73*), *G_location*(*p. 72*), and *G_mapset*(*p. 72*) routines in the *GIS Library.* See *§4.2 GISDBASE* [*p. 16*] for details about GISDBASE, *§4.3 Locations* [*p. 16*] for details about database locations, and *§4.4 Mapsets* [*p. 16*] 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*(*p. 73*) and *G_setenv*(*p. 73*). The GRASS program *g.gisenv* provides a command level interface to these variables.

10.3. Difference Between GRASS and UNIX Environments

The GRASS environment is similar to the UNIX environment in that programs 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 program for later use by other programs. For example, the GRASS start-up sets these variables for use by all other

[2] GRASS programs 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.

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.

- 56 -

Chapter 11

Compiling and Installing GRASS Programs

GRASS programs are compiled and installed using the GRASS *gmake4.1* front-end to the UNIX *make* command: *gmake4.1* reads a file named *Gmakefile* to construct a *make.rules* file (see *§11.4.1 Multiple-Architecture Conventions* [*p. 64*] 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 RFS or NFS mounted to various machines with differing architectures.) This chapter assumes that the programmer is familiar with *make* and its accompanying *makefiles.*

11.1. gmake4.1

The GRASS *gmake4.1* utility allows *make* compilation rules to be developed without having to specify machine and installation dependent information. *gmake4.1* 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. 58*].)

gmake4.1 is invoked as follows:[1]

```
gmake4.1 [source directory] [target]
```

If run without arguments, *gmake4.1* 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, *gmake4.1* 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.

[1] When GRASS is installed, *gmake4.1* is placed into a directory which is in your $PATH (e.g. /usr/local/bin). You should be able to run *gmake4.1* without having to specify its full path name.

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. 60*]. The full set of variables can be seen in *Appendix A. Annotated Gmakefile Predefined Variables* [*p. 287*]. Variables marked with (-) are not commonly used.

GRASS Directories: The following variables tell *gmake4.1* where source code and program directories are:

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

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

ETC This is the directory where support files and programs live. These support files and programs 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. *gmake4.1* 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. 69*] for details about this library, and *§12.21 Loading the GIS Library* [*p. 156*] for a sample *Gmakefile* which loads this library.

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

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. 229*] for details about this library, and *§20.4 Loading the Vask Library* [*p. 233*] for a sample *Gmakefile* which loads this library.

SEGMENTLIB

This names the *Segment Library,* which manages large matrix data. See *§19 Segment Library* [*p. 221*] for details about this library, and *§20.4 Loading the Vask Library* [*p. 233*] 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. 183*] for details about this library, and *§15.9 Loading the Raster Graphics Library* [*p. 193*] 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. 195*] for details about this library, and *§16.11 Loading the Display Graphics Library* [*p. 207*] 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 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 programs from source (.c) files* [*p. 61*] 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. 62*] for details.

CFLAGS (-) This variable specifies all the C compiler options. It should never be necessary to use this variable - *gmake4.1* 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 *gmake4.1* 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 programs since there is no way to automatically supply these flags to the loader.

MAKEALL This defines a command which runs *gmake4.1* 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 program) 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 programs from source (.c) files

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

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

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

$(BIN)/xyz: $(OBJ) $(GISLIB)
    $(CC) $(LDFLAGS) -o $@ $(OBJ) $(GISLIB)

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

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

$@ 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* defines $(CC) as cc, but using $(CC) will allow other C-like compilers to be used instead.

$(BIN) is a *gmake4.1* variable which names the UNIX directory where GRASS commands live. Specifying the target as $(BIN)/xyz will cause *gmake4.1* 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 program 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 give 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 program.

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 program *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)

$(GISLIB): # in case 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. *gmake4.1* requires that these libraries be built using the $(AR) rule as

- 63 -

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 given. However, it is the first one in the *Gmakefile* which is built by *gmake4.1*. If there are more targets to be built, the first target must explicitly or implicitly cause *gmake4.1* 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)

$(BIN)/xyz: $(XYZ) $(GISLIB)
	$(CC) $(LDFLAGS) -o $@ $(XYZ) $(GISLIB)

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

If it is desired to run the compilation in various subdirectories, a *Gmakefile* could be constructed which simply runs *gmake4.1* 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.386 would contain library archives for Sun/4 and SPARC architectures. Likewise, $(SRC)/OBJ.386 would contain the object files for 386 architectures, and $(SRC)/LIB.386 would contain library archives for 386 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.

gmake4.1 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 v4.1 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 program which uses the new parser capability (see *§12.15 Command Line Parsing* [p. 132].) The parser has both a command-line and a rudimentary prompt-based interactive interface.

2. The programmer writes 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 program and the interactive program be somehow merged into one program. This is accomplished by placing both programs in separate directories under $(GISBASE)/etc/bin and creating a link (as described below) in $(BIN).

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

etc/bin/main/inter $(BIN_MAIN_INTER)
Interactive versions of the primary GRASS commands.

etc/bin/main/cmd $(BIN_MAIN_CMD)
Command-line versions of the primary GRASS commands.

etc/bin/alpha/inter $(BIN_ALPHA_INTER)
Interactive versions of the alpha-version commands.

etc/bin/alpha/cmd $(BIN_ALPHA_CMD)
Command-line versions of the alpha-version commands.

etc/bin/contrib/inter $(BIN_CONTRIB_INTER)
Interactive versions of the contributed commands.

etc/bin/contrib/cmd $(BIN_CONTRIB_CMD)
Command-line versions of the contributed 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

The front.end program 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 program has only one .c source file, it is tempting to compile the program 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 program.

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 -- gmake4.1 cannot detect simultaneous compilations.

- 68 -

Chapter 12

GIS Library

12.1. Introduction

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. 15] for a general description of GRASS databases, *§5 Raster Maps* [p. 23] for details about raster map layers in GRASS, and *§9 Region and Mask* [p. 49] 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 programs 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 programs which use this prefix.

An alphabetic index is provided in *§25.4 Appendix C. Index to GIS Library* [p. 293].

[1] The GRASS compilation process, described in *§11 Compiling and Installing GRASS Programs* [p. 57], automatically tells the C compiler how to find this and other GRASS header files.

12.2. Library Initialization

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

G_gisinit (program_name) *initialize gis library*

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 program exits. The **program_name** is stored for later recall by *G_program_name*(*p. 152*). It is recommended that argv[0] be used for the **program_name**:

```
main(argc, argv) 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.

G_fatal_error (message) *print error message and exit*

G_warning (message) *print warning message and continue*

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 programs may want the

[2] $GISBASE is the directory where GRASS is installed. See *§10.1 UNIX Environment* [*p. 53*] for details.

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:

G_set_error_routine (handler) *change error handling*

int (*handler)( );

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

```
handler (message, fatal)
        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 program will exit after the handler returns.

G_unset_error_routine ( ) *reset normal error handling*

This routine resets the error handling for *G_fatal_error*(*p. 70*) and *G_warning*(*p. 70*) back to the default action.

G_sleep_on_error (flag) *sleep on error?*

int flag;

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

G_suppress_warnings (flag) *suppress warnings?*

int flag;

If **flag** is 0, then *G_warning*(*p. 70*) 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*(*p. 70*).

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. 53*] for a discussion of the GRASS environment.

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

char *
G_location ( ) *current location name*

Returns the name of the current database location. This routine should be used by programs that need to display the current location to the user. See *§4.3 Locations* [*p. 16*] for an explanation of locations.

char *
G_mapset ( ) *current mapset name*

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. 16*] for an explanation of mapsets.

char *
G_myname ( ) *location title*

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. 19*] for a discussion of the PERMANENT mapset.

char *
G_gisbase ( ) *top level program directory*

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

bin	commands run by the user
etc	programs 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 programs to be written independently of where the GRASS system is actually installed on the machine. For example, to run the program *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 ( ) *top level database directory*

Returns the full UNIX path name of the directory which holds the database locations. See *§4.2 GISDBASE* [*p. 16*] for a full explanation of this directory.

char *
G_location_path ( ) *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/grass3/data* database directory, this routine will return a string which looks like */usr/grass3/data/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.**

char *
G_getenv (name) *query GRASS environment variable*

char *
G__getenv (name) *query GRASS environment variable*

char *name;

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.

G_setenv (name, value) *set GRASS environment variable*

G__setenv (name, value) *set GRASS environment variable*

char *name;
char *value;

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 program 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. 98*]), vector files (*§12.11 Vector File Processing* [*p. 121*]), or site files (*§12.12 Site List Processing* [*p. 126*]), 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" element. See *§4.5.2 Elements* [*p. 18*] 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. 18*] 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 */
```

The user will see the following:

[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].*

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 (prompt, name, element, label) *prompt for existing database file*

char *prompt;
char *name;
char *element;
char *label;

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. 20*] for some more details about the search path.

char *
G_ask_new (prompt, name, element, label) *prompt for new database file*

char *prompt;
char *name;
char *element;
char *label;

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 (prompt, name, element, label) *prompt for existing database file*

char *prompt;
char *name;
char *element;
char *label;

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

[4] This line of the prompt can be modified using *G_set_ask_return_msg*(*p. 76*).

used to modify a file. GRASS only permits users to modify files in the current mapset.

char *
G_ask_any (prompt, name, element, label, warn) *prompt for any valid file name*

char *prompt;
char *name;
char *element;
char *label;
int warn;

The user is asked to enter any legal 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 legal name is accepted and no warning is issued to the user if the file exists.

G_set_ask_return_msg (msg) *set Hit 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**."

char *
G_get_ask_return_msg ( ) *get Hit RETURN msg*

The current *msg* (as set by *G_set_ask_return_msg*(*p. 76*)) 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:

char *
G_fully_qualified_name (name, mapset) *fully qualified file 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"
main(argc,argv) 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 programs cannot make use of the interactive prompting routines described above. For example, a command line driven program 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 (element, name, mapset) *find a database file*

char *element;
char *name;
char *mapset;

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

[5] See *§4.7.1 Mapset Search Path* [*p. 20*] for more details about the search path.

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. 76*]) 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 legal name. The routines which prompt the user for file names (e.g., *G_ask_new*(*p. 75*)) 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:

G_legal_filename (name) *check for legal database file names*

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*(*p. 75*), and noninteractively using *G_find_file*(*p. 77*)

G_open_old (element, name, mapset) *open a database file for reading*

```
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.

FILE *
G_fopen_old (element, name, mapset) *open a database file for reading*

```
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*(*p. 75*), and noninteractively using *G_find_file*(*p. 77*)

G_open_update (element, name) *open a database file for 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.

G_fopen_append (element, name) *open a database file for update*

char *element;
char *name;

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*(*p. 75*). If obtained noninteractively (e.g., from the command line), *G_legal_filename*(*p. 78*) 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*(*p. 75*) guarantees that **name** will not exist, but if **name** is obtained from the command line, **name** may exist. In this case *G_find_file*(*p. 77*) could be used to see if **name** exists.

[6] GRASS does not allow files to be created outside the current mapset; see *§4.7 Database Access Rules* [*p. 20*].

G_open_new (element, name) *open a new database file*

char *element;
char *name;

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 (element, name) *open a new database file*

char *element;
char *name;

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.

12.5.8. Database File Management

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

G_rename (element, old, new) *rename a database file*

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.

[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. 20*].

G_remove (element, name) *remove a database file*

```
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 UNIX free( ) routine to release memory allocated by these routines.

char *
G_malloc (size) *memory allocation*

```
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.

char *
G_realloc (ptr, size) *memory allocation*

```
char *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 is different than malloc( ), which does not handle a NULL **ptr.**

char *
G_calloc (n, size) *memory allocation*

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. 103*]

12.7. The Region

The region concept is explained in *§9.1 Region* [*p. 49*]. 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 program. This active program region is what controls reading and writing of raster file data.

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 *§12.20 GIS Library Data Structures* [*p. 153*].

12.7.1. The Database Region

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

G_get_window (region) *read the database region*

struct Cell_head *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,

[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.)

since its use implies that the active program region will not be used. Programs that read or write raster file data (or vector data) can query the active program region using *G_window_rows*(*p. 85*) and *G_window_cols*(*p. 85*).

G_put_window (region) *write the database region*

struct Cell_head *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 programs 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* programs 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 programs *g.region* allow the user to set their database region from the default region. (See *§4.6 Permanent Mapset* [*p. 19*] for a discussion of the default region.) The following routine reads this region:

G_get_default_window (region) *read the default region*

struct Cell_head *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.

12.7.2. The Active Program Region

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

G_window_rows ( ) *number of rows in active region*

G_window_cols ( ) *number of columns in active region*

These routines return the number of rows and columns (respectively) in the active program 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 ...
    }
}
```

G_set_window (region) *set the active region*

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 program.[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.

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 program 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.

[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 program to be executed using the system( ) or popen( ) routines.

G_get_set_window (region) *get the active region*

struct Cell_head *region;

Gets the values of the currently active region into **region.** If *G_set_window*(*p. 85*) has been called, then the values set by that call are retrieved. Otherwise the user's database region is retrieved.

Note. For programs that read or write raster data, and really need the full region information, this routine is preferred over *G_get_window*(*p. 83*). However, since *G_window_rows*(*p. 85*) and *G_window_cols*(*p. 85*) 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 *
G_align_window (region, ref) *align two regions*

struct Cell_head *region, *ref;

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
G_col_to_easting (col, region) *column to easting*

double col;
struct Cell_head *region;

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
G_row_to_northing (row, region) *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.

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

    double row;
    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. 49*] for more information about these values.

G_projection ( ) *query cartographic projection*

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[11]

Others may be added in the future.

char *
G_database_projection_name (proj) *query cartographic projection*

int proj;

Returns a pointer to a string which is a printable name for projection code **proj** (as returned by *G_projection*(*p. 88*)). Returns NULL if **proj** is not a valid projection.

char *
G_database_unit_name (plural) *database units*

int plural

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 ( ) *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.

G_zone ( ) *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

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

the Equidistant Cylindrical Projection.[12] ECP has the property that *where am I* and *row-column* calculations are identical to those in planimetric grids (like UTM[13]). 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 programs (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 ±360 degrees are equivalent.

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,[14] then latitude-longitude ASCII format is invoked. Otherwise, a standard floating-point to ASCII conversion is made.

---

[12] Also known as Plate Carree.

[13] Universal Transverse Mercator Projection.

[14] Defined in "gis.h".

G_format_easting (east, buf, projection) *easting to ASCII*

double east ;
char *buf ;
int projection ;

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

G_format_northing (north, buf, projection) *northing to ASCII*

double north ;
char *buf ;
int projection ;

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

G_format_resolution (resolution, buf, projection) *resolution to ASCII*

double resolution ;
char *buf ;
int projection ;

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

G_scan_easting (buf, easting, projection) *ASCII easting to double*

char *buf ;
double *easting ;
int projection ;

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

G_scan_northing (buf, northing, projection) *ASCII northing to double*

char *buf ;
double *northing ;
int projection ;

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

G_scan_resolution (buf, resolution, projection) *ASCII resolution to double*

```
char *buf ;
double *resolution ;
int projection ;
```

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.

G_begin_cell_area_calculations ( ) *begin cell area calculations*

This routine must be called once before any call to *G_area_of_cell_at_row*(*p. 91*). 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 have 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 (row) *cell area in specified row*

```
int 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.

G_begin_zone_area_on_ellipsoid (a, e2, s) *begin area calculations for ellipsoid*

double a, e2, 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.

double
G_area_for_zone_on_ellipsoid (north, south) *area between latitudes*

double north, south ;

Returns the area between latitudes **north** and **south** scaled by the factor **s** passed to *G_begin_zone_area_on_ellipsoid*(*p. 92*).

G_begin_zone_area_on_sphere (r, s) *initialize calculations for sphere*

double north, south ;

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*(*p. 92*).

double
G_area_for_zone_on_sphere (north, south) *area between latitudes*

double north, south ;

Returns the area between latitudes **north** and **south** scaled by the factor **s** passed to *G_begin_zone_area_on_sphere*(*p. 92*).

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.

G_begin_polygon_area_calculations ( ) *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 have a metric (e.g. imagery.)

double
G_area_of_polygon (x, y, n) *area in square meters of polygon*

```
double *x, *y ;
int n ;
```

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.

G_begin_ellipsoid_polygon_area (a, e2) *begin area calculations*

```
double a, e2 ;
```

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 (lon, lat, n) *area of lat-long polygon*

```
double *lon, *lat ;
int n ;
```

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.

G_begin_distance_calculations ( ) *begin distance calculations*

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 have a metric (e.g. imagery.)

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

```
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.

G_begin_geodesic_distance (a, e2) *begin geodesic distance*

```
double a, 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.

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

double lon1, lat1, lon2, 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.

G_set_geodesic_distance_lat1 (lat1) *set geodesic distance lat1*

double lat1 ;

Set the first latitude.

G_set_geodesic_distance_lat2 (lat2) *set geodesic distance lat2*

double lat2 ;

Set the second latitude.

double
G_geodesic_distance_lon_to_lon (lon1, lon2) *geodesic distance*

double lon1, lon2 ;

Calculates the geodesic distance from **lon1,lat1** to **lon2,lat2** in meters, where **lat1** was the latitude passed to *G_set_geodesic_distance_lat1*(*p. 95*) and **lat2** was the latitude passed to *G_set_geodesic_distance_lat2*(*p. 95*).

12.8.5. Global Wraparound

These next routines provide a mechanism for determining the relative position of a pair of longitudes. Since longitudes of ±360 are equivalent, but GRASS requires the east to be bigger than the west, some adjustment of coordinates is necessary.

double
G_adjust_easting (east, region) *returns east larger than west*

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.

double
G_adjust_east_longitude (east, west) *adjust east longitude*

double east, 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.

G_shortest_way (east1, east2) *shortest way between eastings*

double *east1, *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

char *
G_ellipsoid_name (n) *return ellopsoid 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++ )
        printf("%s\n", name);
```

G_get_ellipsoid_by_name (name, a, e2) *get ellipsoid by name*

char *name
double *a, *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.

G_get_ellipsoid_parameters (a, e2) *get ellipsoid parameters*

double *a, *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.

double
G_meridional_radius_of_curvature (lon, a, e2) *meridional radius of curvature*

double lon, a, 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 (lon, a, e2) *transverse radius of curvature*

double lon, a, e2 ;

Returns the transverse radius of curvature at a given longitude:

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

double
G_radius_of_conformal_tangent_sphere (lon, a, e2) *radius of conformal tangent sphere*

double lon, a, e2 ;

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

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

G_pole_in_polygon (x, y, n) *pole in polygon*

```
double *x, *y ;
int n ;
```

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. All analyses are performed with raster file data. 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 program 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[15], 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 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.[16] 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.

---

[15] See *Appendix B. The CELL Data Type* [p. 291] for a discussion of the CELL type and how to use it (and avoid misusing it).

[16] 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].*

char *
G_ask_cell_old (prompt, name) *prompt for existing raster file*

char *prompt;
char *name;

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

char *
G_ask_cell_in_mapset (prompt, name) *prompt for existing raster file*

char *prompt;
char *name;

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

char *
G_ask_cell_new (prompt, name) *prompt for new raster file*

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)
    exit(0);
```

12.9.2. Finding Raster Files in the Database

Noninteractive programs cannot make use of the interactive prompting routines described above. For example, a command line driven program 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 *
G_find_cell (name, mapset) *find a raster file*

char *name;
char *mapset;

Look 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,[17] 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*(*p. 99*) or *G_ask_cell_in_mapset*(*p. 99*), and noninteractively using *G_find_cell*(*p. 100*)

[17] See *§4.7.1 Mapset Search Path* [*p. 20*] for more details about the search path.

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

char *name;
char *mapset;

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 program region.[18] Preparation required for reading the various raster file formats[19] 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[20] and open it for writing. The raster file **name** should be obtained interactively using *G_ask_cell_new*(*p. 99*). If obtained noninteractively (e.g., from the command line), *G_legal_filename*(*p. 78*) 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*(*p. 99*) guarantees that **name** will not exist, but if **name** is obtained from the command line, **name** may exist. In this case *G_find_cell*(*p. 100*) could be used to see if **name** exists.

Warning. However, there is a subtle trap. The temporary file, which is created using *G_tempfile*(*p. 131*), 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( ),[21] 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

---

[18] See also *§12.7 The Region* [*p. 83*].

[19] See *§5.2 Raster File Format* [*p. 24*] for an explanation of the various raster file formats.

[20] GRASS does not allow files to be created outside the current mapset. See *§4.7 Database Access Rules* [*p. 20*].

[21] See also *G_fork*(*p. 150*).

under *G_tempfile*(*p. 131*) for more details.)

G_open_cell_new (name) *open a new raster file (sequential)*

char *name;

Creates and opens the raster file **name** for writing by *G_put_map_row*(*p. 105*) 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.

G_open_cell_new_random (name) *open a new raster file (random)*

char *name;

Creates and opens the raster file **name** for writing by *G_put_map_row_random*(*p. 105*) which allows writing the raster file in a random fashion. The file will be created uncompressed.[22]

A nonnegative file descriptor is returned if the open is successful. Otherwise a diagnostic message is printed and a negative value is returned.

G_open_cell_new_uncompressed (name) *open a new raster file (uncompressed)*

char *name;

Creates and opens the raster file **name** for writing by *G_put_map_row*(*p. 105*) 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.[23] This routine is provided so the *r.compress* program can create uncompressed raster files.

---

[22] 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*").

[23] 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.

12.9.5. Allocating Raster I/O Buffers

Since there is no predefined limit for the number of columns in the region,[24] buffers which are used for reading and writing raster data must be dynamically allocated.

CELL *
G_allocate_cell_buf ( ) *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( );
```

If larger buffers are required, the routine *G_malloc*(*p. 82*) can be used.

If sufficient memory is not available, an error message is printed and exit( ) is called.

G_zero_cell_buf (buf) *zero a raster buffer*

CELL *buf;

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*(*p. 103*).

12.9.6. Reading Raster Files

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.[25] 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.

---

[24] See *G_window_cols*(*p. 85*) to find the number of columns in the region.

[25] The GRASS region is discussed from a user perspective in *§9.1 Region* [*p. 49*] and from a programmer perspective in *§12.7 The Region* [*p. 83*]. The routines which are commonly used to determine the number of rows and columns in the region are *G_window_rows*(*p. 85*) and *G_window_cols*(*p. 85*).

G_get_map_row (fd, cell, row) *read a raster file*

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*(*p. 101*)) 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*(*p. 103*).

This routine prints a diagnostic message and returns -1 if there is an error reading the raster file. Otherwise a nonnegative value is returned.

G_get_map_row_nomask (fd, cell, row) *read a raster file (without masking)*

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.

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 programs *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 programs which do this should be minimal. See *§9.2 Mask* [*p. 51*] for more information about the mask.

12.9.7. Writing Raster Files

The routines described here write raster file data.

G_put_map_row (fd, buf) *write a raster file (sequential)*

int fd;
CELL *buf;

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*(*p. 102*). The cell **buf** must have been allocated large enough for the region, perhaps using *G_allocate_cell_buf*(*p. 103*).

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);
}
```

G_put_map_row_random (fd, buf, row, col, ncells) *write a raster file (random)*

int fd;
CELL *buf;
int row, col, ncells;

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*(*p. 102*). 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.

G_close_cell (fd) *close a raster file*

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 program 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. 106*] for routines which write raster support files.

G_unopen_cell (fd) *unopen a raster file*

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. 101*]).

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 program exits without closing the file, the temporary file will not be removed at program 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. 23*]. 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. 26*]. The routines described below use the *Cell_head* structure which is shown in detail in *§12.20 GIS Library Data Structures* [*p. 153*].

G_get_cellhd (name, mapset, cellhd) *read the raster header*

```
char *name;
char *mapset;
struct Cell_Head *cellhd;
```

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. 28*] for information about reclass files, and *G_is_reclass*(*p. 108*) 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.[26] If it is necessary to read the raster file directly without resampling into the active region,[27] then the raster header can be used to set the active region using *G_set_window*(*p. 85*).

char *
G_adjust_Cell_head (cellhd, rflag, cflag) *adjust cell header*

```
struct Cell_head *cellhd;
int rflag, cflag;
```

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.

---

[26] See *§12.7 The Region* [*p. 83*].

[27] But see *§9 Region and Mask* [*p. 49*] for a discussion of when this should and should not be done.

G_put_cellhd (name, cellhd) *write the raster header*

char *name;
struct Cell_head *cellhd;

This routine writes the information from the **cellhd** structure to the raster header file for the map layer **name** in the current mapset.

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*(*p. 106*) to give new raster files correct header files, and by the *r.support* program to give users a means of creating or modifying raster headers.

G_is_reclass (name, mapset, r_name, r_mapset) *reclass file?*

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.**

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. 28*].

The routines described below manage the category file. Some of them use the *Categories* structure which is described in *§12.20 GIS Library Data Structures* [*p. 153*].

12.10.2.1. Reading and Writing the Raster Category File

The following routines read or write the category file itself:

G_read_cats (name, mapset, cats) *read raster category file*

```
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.

G_write_cats (name, cats) *write raster category file*

```
char *name;
struct Categories *cats;
```

Writes the category file for the raster file **name** in the current mapset from the **cats** structure.

Returns 0 if successful. Otherwise, -1 is returned (no diagnostic is printed).

char *
G_get_cell_title (name, mapset) *get raster map title*

```
char *name;
char *mapset;
```

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 (name, title) *change raster map title*

```
char *name;
char *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 (n, cats) *get a category label*

CELL n;
struct Categories *cats;

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.

char *
G_get_cats_title (cats) *get title from category structure*

struct 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.

G_init_cats (n, title, cats) *initialize category structure*

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);
```

G_set_cat (n, label, cats) *set a category label*

CELL n;
char *label;
struct Categories *cats;

The **label** is copied into the **cats** structure for category **n.**

G_set_cats_title (title, cats) *set title in category structure*

char *title;
struct Categories *cats;

The **title** is copied into the **cats** structure.

G_free_cats (cats) *free category structure memory*

struct Categories *cats;

Frees memory allocated by *G_read_cats*(*p. 109*), *G_init_cats*(*p. 110*) and *G_set_cat*(*p. 111*).

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. 29*].

The routines that manipulate the raster color file use the *Colors* structure which is described in detail in *§12.20 GIS Library Data Structures* [*p. 153*].

12.10.3.1. Reading and Writing the Raster Color File

The following routines read, create, modify, and write color tables.

G_read_colors (name, mapset, colors) *read map layer color table*

char *name;
char *mapset;
struct Colors *colors;

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.

G_write_colors (name, mapset, colors) *write map layer color table*

```
char *name;
char *mapset;
struct Colors *colors;
```

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*(*p. 113*) to initialize the structure and *G_add_color_rule*(*p. 113*) to set the category colors.[28]

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.

G_lookup_colors (raster, red, green, blue, set, n, colors) *lookup an array of colors*

```
CELL *raster;
unsigned char *red;
unsigned char *green;
unsigned char *blue;
unsigned char *set;
int n;
struct Colors *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**.

[28] These routines are called by higher level routines which read or create entire color tables, such as *G_read_colors*(*p. 111*) or *G_make_ramp_colors*(*p. 115*).

Note. The **red, green,** and **blue** intensities will be in the range 0–255.

G_get_color (cat, red, green, blue, colors) *get a category 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.

G_init_colors (colors) *initialize color structure*

```
struct Colors *colors;
```

The **colors** structure is initialized for subsequent calls to *G_add_color_rule*(*p. 113*) and *G_set_color*(*p. 114*).

G_add_color_rule (cat1, r1, g1, b1, cat2, r2, g2, b2, colors) *set colors*

```
CELL cat1, cat2;
int r1,g1,b1;
int r2,g2,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*(*p. 112*) and *G_get_color*(*p. 113*). 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. 29*] how this routine fits into the 4.1 raster color logic.

Note. The **colors** structure must have been initialized by *G_init_colors*(*p. 113*).

See *§12.10.3.4 Predefined Color Tables* [*p. 114*] for routines to build some predefined color tables.

G_set_color (cat, red, green, blue, colors) *set a category 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 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*(*p. 113*) should be used whenever possible.

Note. The **colors** structure must have been initialized by *G_init_colors*(*p. 113*).

G_get_color_range (min, max, colors) *get color range*

```
CELL *min, *max;
struct Colors *colors;
```

Gets the minimum and maximum raster values that have colors associated with them.

G_free_colors (colors) *free color structure memory*

```
struct Colors *colors;
```

The dynamically allocated memory associated with the **colors** structure is freed.

Note. This routine may be used after *G_read_colors*(*p. 111*) as well as after *G_init_colors*(*p. 113*).

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*(*p. 118*).

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.

G_make_aspect_colors (colors, min, max) *make aspect colors*

struct Colors *colors;
CELL min, max;

Generates a color table for aspect data.

G_make_ramp_colors (colors, min, max) *make color ramp*

struct Colors *colors;
CELL min, 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.

G_make_wave_colors (colors, min, max) *make color wave*

struct Colors *colors;
CELL min, 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.

G_make_grey_scale_colors (colors, min, max) *make linear grey scale*

struct Colors *colors;
CELL min, max;

Generates a grey scale color table. Each color is a level of grey, increasing from black to white.

G_make_rainbow_colors (colors, min, max) *make rainbow colors*

struct Colors *colors;
CELL min, 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.

G_make_random_colors (colors, min, max) *make random colors*

struct Colors *colors;
CELL min, max;

Generates random colors. Good as a first pass at a color table for nominal data.

G_make_ryg_colors (colors, min, max) *make red,yellow,green colors*

struct Colors *colors;
CELL min, max;

Generates a color table that goes from red to yellow to green.

G_make_gyr_colors (colors, min, max) *make green,yellow,red colors*

struct Colors *colors;
CELL min, max;

Generates a color table that goes from green to yellow to red.

G_make_histogram_eq_colors (colors, s) *make histogram-stretched grey 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 *§12.10.6 Raster Histograms* [*p. 119*].)

12.10.4. 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* [*p. 30*].

The following routines manage this file. They use the *History* structure which is described in *§12.20 GIS Library Data Structures* [*p. 153*].

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.

G_read_history (name, mapset, history) *read raster history file*

```
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.

G_write_history (name, history) *write raster history file*

```
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*(*p. 117*).

G_short_history (name, type, history) *initialize history structure*

```
char *name;
char *type;
struct History *history;
```

This routine initializes the **history** structure, recording the date, user, program 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.

12.10.5. 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* [*p. 31*].

The routines below use the *Range* data structure which is described in *§12.20 GIS Library Data Structures* [*p. 153*].

G_read_range (name, mapset, range) *read raster 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.

G_write_range (name, range) *write raster range file*

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:

G_init_range (range) *initialize range structure*

struct Range *range;

Initializes the **range** structure for updates by *G_update_range*(*p. 118*) and *G_row_update_range*(*p. 119*).

G_update_range (cat, range) *update range structure*

CELL cat;
struct Range *range;

Compares the **cat** value with the minimum and maximum values in the **range** structure, modifying the range if **cat** extends the range.

G_row_update_range (cell, n, range) *update range structure*

```
CELL *cell;
int n;
struct Range *range;
```

This routine updates the **range** data just like *G_update_range*(*p. 118*), but for **n** values from the **cell** array.

The range structure is queried using the following routine:

G_get_range_min_max (range, min, max) *get range min and max*

```
struct Range *range;
CELL *min, *max;
```

The **min**inum and **max**imum CELL values are extracted from the **range** structure.

12.10.6. 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:

G_init_cell_stats (s) *initialize cell stats*

```
struct Cell_stats *s;
```

This routine, which must be called first, initializes the Cell_stats structure **s**.

G_free_cell_stats (s) *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*(*p. 119*).

This next routine stores values in the histogram:

G_update_cell_stats (data, n, s) *add data to 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):

G_find_cell_stat (cat, count, s) *random query of cell stats*

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:

G_rewind_cell_stats (s) *reset/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.

G_next_cell_stat (cat, count, s) *retrieve sorted cell stats*

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)
        printf("%ld %ld\n", (long) cat, count);
```

12.11. Vector File Processing

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. 157*].

12.11.1. 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.[29] 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.

[29] 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].*

char *
G_ask_vector_old (prompt, name) *prompt for an existing vector file*

```
char *name;
char *mapset;
```

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

char *
G_ask_vector_in_mapset (prompt, name) *prompt for an existing vector file*

```
char *name;
char *mapset;
```

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

char *
G_ask_vector_new (prompt, name) *prompt for a new vector file*

```
char *name;
char *mapset;
```

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);
```

12.11.2. Finding Vector Files in the Database

Noninteractive programs cannot make use of the interactive prompting routines described above. For example, a command line driven program 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:

G_find_vector (name, mapset) *find a vector file*

G_find_vector2 (name, mapset) *find a vector file*

```
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,[30] 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.[31] 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 */
```

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

```
char name[50];


if (G_find_vector(name,G_mapset( )) = = NULL)
    /* not found */
```

[30] See *§4.7.1 Mapset Search Path* [*p. 20*] for more details about the search path.

[31] 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( ).

12.11.3. 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*(*p. 122*) or *G_ask_vector_in_mapset*(*p. 122*), and noninteractively using *G_find_vector*(*p. 123*) or *G_find_vector2*(*p. 123*).

FILE *
G_fopen_vector_old (name, mapset) *open an existing vector file*

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. 157*].)

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.

12.11.4. Creating and Opening New Vector Files

The following routine creates the new vector file **name** in the current mapset[32] and opens it for writing. The vector file **name** should be obtained interactively using *G_ask_vector_new*(*p. 122*). If obtained noninteractively (e.g., from the command line), *G_legal_filename*(*p. 78*) 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*(*p. 122*) guarantees that **name** will not exist, but if **name** is obtained from the command line, **name** may exist. In this case *G_find_vector*(*p. 123*) could be used to see if **name** exists.

[32] GRASS does not allow files to be created outside the current mapset. See *§4.7 Database Access Rules* [*p. 20*].

FILE *
G_fopen_vector_new (name) *open a new vector file*

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).

The file descriptor can then be used with routines in the *Dig Library* to write the vector file. (See *§13 Vector Library* [*p. 157*].)

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.

12.11.5. Reading and Writing Vector Files

Reading and writing vector files is handled by routines in the *Dig Library.* See *§13 Vector Library* [*p. 157*] for details.

12.11.6. 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 *§12.20 GIS Library Data Structures* [*p. 153*].

Note. The vector category file has exactly the same structure as the raster category file. In fact, it exists so that the program *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. 110*] which modify the *Categories* structure can therefore be used to set and change vector categories as well.

G_read_vector_cats (name, mapset, cats) *read vector category file*

```
char *name;
char *mapset;
struct Categories *cats;
```

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.

G_write_vector_cats (name, cats) *write vector category file*

```
char *name;
struct Categories *cats;
```

Writes the category file for the vector file **name** in the current mapset from the **cats** structure.

Returns 0 if successful. Otherwise, -1 is returned (no diagnostic is printed).

12.12. Site List Processing

GRASS has a point database capability called *s.menu,* which manages a database of point or site information. The *s.menu* program provides the majority of the analytical capabilities within GRASS for site data. The routines described here provide programmers with mechanisms for reading existing site list files and for creating new ones. The reader should also see *§7 Point Data: Site List Files* [*p. 41*] for more details about the site list files.

12.12.1. 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.[33] 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

[33] 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].*

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 (prompt, name) *prompt for existing site list file*

    char *prompt;
    char *name;

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

char *
G_ask_sites_in_mapset (prompt, name) *prompt for existing site list file*

    char *prompt;
    char *name;

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

char *
G_ask_sites_new (prompt, name) *prompt for new site list file*

    char *prompt;
    char *name;

Asks the user to enter a 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);
```

12.12.2. Opening Site List Files

The following routines open site list files:

FILE *
G_fopen_sites_new (name) *open a new site list file*

char *name;

Creates an empty site list file **name** in the current mapset and opens it for writing.

Returns an open file descriptor if successful. Otherwise, returns NULL.

FILE *
G_fopen_sites_old (name, mapset) *open an existing site list file*

char *name;
char *mapset;

Opens the site list file **name** in **mapset** for reading.

Returns an open file descriptor if successful. Otherwise, returns NULL.

12.12.3. Reading and Writing Site List Files

G_get_site (fd, east, north, desc) *read site list file*

FILE *fd;
double *east, *north;
char **desc;

This routine sets **east** and **north** for the next "point" from the site list file open on file descriptor **fd** (as returned by *G_fopen_sites_old*(*p. 128*)), and **desc** is set to point to the description of the site.

Returns: 1 if a site was found; -1 if there were no more sites.

For example:

```
double east, north;
char *desc;
FILE *fd;

fd = G_fopen_site_old (name, mapset);
while (G_get_site (fd, &east, &north, &desc) > 0)
        printf ("%lf %lf %s\n", east, north, desc);
```

Note: desc points to static memory, so each call overrides the description from the previous call.

G_put_site (fd, east, north, desc) *write site list file*

FILE *fd;
double east, north;
char *desc;

Writes the **east** and **north** coordinates and site description **desc** to the site file opened on file descriptor **fd** (as returned by *G_fopen_sites_new*(*p. 128*)).

12.13. General Plotting Routines

The following routines form the foundation of a general purpose line and polygon plotting capability.

G_bresenham_line (x1, y1, x2, y2, point) *Bresenham line algorithm*

int x1, y1 ;
int x2, y2 ;
int (*point)( ) ;

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*(*p. 129*) to function correctly, and is independent of all following routines.

G_setup_plot (t, b, l, r, Move, Cont) *initialize plotting routines*

double t, b, l, r ;
int (*Move)( );
int (*Cont)( );

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

G_plot_line (east1, north1, east2, north2) *plot line between latlon coordinates*

double east1, north1, east2, 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*(*p. 129*) for the required coordinate initialization procedure.

G_plot_polygon (east, north, n) *plot filled polygon with n vertices*

double *east, *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*(*p. 129*) for the required coordinate initialization procedure.

G_plot_where_en (x, y, east, north) *x,y to east,north*

int x, y ;
double *east, *north ;

The pixel coordinates **x,y** are converted to map coordinates **east,north.**

See *G_setup_plot*(*p. 129*) for the required coordinate initialization procedure.

G_plot_where_xy (east, north, x, y) *east,north to x,y*

double east, north ;
int *x, *y ;

The map coordinates **east,north** are converted to pixel coordinates **x,y.**

See *G_setup_plot*(*p. 129*) for the required coordinate initialization procedure.

G_plot_fx (f, east1, east2) *plot f(east1) to f(east2)*

```
double (*f)( ) ;
double east, 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*(*p. 129*) for the required coordinate initialization procedure.

12.14. Temporary Files

Often it is necessary for programs to use temporary files to store information that is only useful during the program run. After the program 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 program. It would not be good for a fixed name like "/tmp/mytempfile" to be used. If the program 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 program 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 program. 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 program 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 program exits. However, GRASS database management will eventually remove all temporary files created by G_tempfile( ) that have been left behind by the programs 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 program. GRASS database management will remove these files only if the program which created them is no longer running. However, this feature has a subtle trap. Programs which create child processes (using the UNIX fork( )[34] 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.

12.15. 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 programs 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.

12.15.1. 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 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

[34] See also *G_fork*(*p. 150*).

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.

12.15.2. 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 *§12.15.5 Full Structure Members Description* [*p. 140*].

12.15.2.1. 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)

12.15.2.2. 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

12.15.3. 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 Programs* [*p. 57*].

struct Option *
G_define_option ( ) *returns Option structure*

Allocates memory for the Option structure and returns a pointer to this memory (of type *struct Option **).

struct Flag *
G_define_flag ( ) *return Flag structure*

Allocates memory for the Flag structure and returns a pointer to this memory (of type *struct Flag **).

G_parser (argc, argv) *parse command line*

```
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.

G_usage ( ) *command line help/usage message*

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*(*p. 134*) 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.

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 promprting.

12.15.4. 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.

12.15.4.1. 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*(*p. 134*) and *G_define_flag*(*p. 134*) as defined in *§12.15.3 Parser Routines* [*p. 134*].

```
#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 *§12.15.5.1 Complete Structure Members Table* [*p. 140*].

(3) Call the parser :

```
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:

```
printf("For the option \"%s\" you chose: <%s>\n", opt->description, opt->answer );
printf("The flag \"-%s\" is %s set.\n", flag->key, flag->answer ? "" : "not" );
```

(5) Running the example program

Once such a program 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 program. The command line options (in this case, "help") are sent to the parser via *G_parser*(*p. 134*). 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:
  -t  Flag 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 program 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
```

12.15.4.2. Full Program 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 Programs* [*p. 57*]).

```
sample: sample.o
$(CC) $(LDFLAGS) -o $@ sample.o $(GISLIB)
```

 - 138 -

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 *§12.15.5.1 Complete Structure Members Table* [*p. 140*].

```
#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 );

	printf("For the option \"%s\" you chose: <%s>\n", opt->description, opt->answer );
	printf("The flag \"-%s\" is: %s set\n", flag->key, flag->answer ? "" : "not");

	printf("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( ) );
		printf("%.31f,%.21f\n", X , Y );
	}
}
```

 - 140 -

12.15.5. 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 *§12.15.5.2 Description of Complex Structure Members* [*p. 142*].

12.15.5.1. 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" ;

 - 141 -

struct Option				
structure member	C type	required	default	description and example
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" ;
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 opt->checker = my_routine( ) ;

12.15.5.2. 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.

12.15.5.2.1. 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*(*p. 134*) in his program. After the *G_parser*(*p. 134*) 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*(*p. 134*), 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*(*p. 134*).

As an example, please review the use of answer members in the structures implemented in *§12.15.4.2 Full Program Example* [*p. 137*].

12.15.5.2.2. 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*(*p. 134*) to capture multiple answers. Multiple answers are separated by commas on the command line after an option.

Note. *G_parser*(*p. 134*) 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
```

12.15.5.2.3. 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"
```

 - 144 -

```
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*(*p. 134*):

```
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*(*p. 134*) 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*(*p. 134*):

```
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*(*p. 134*) 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*(*p. 134*) in case of an error, or by *G_usage*(*p. 134*) 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*(*p. 134*) standard. If the programmer defines the key_desc structure member before a call to *G_parser*(*p. 134*), 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.

12.15.5.2.4. 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*(*p. 75*), "new" to *G_ask_new*(*p. 75*), "any" to *G_ask_any*(*p. 76*), and "mapset" to *G_ask_in_mapset*(*p. 75*).

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")

12.15.6. Common Questions

"How is automatic prompting turned off?"

GRASS 4.0 introduced a new method for driving GRASS interactive and non-interactive programs as described in *§11 Compiling and Installing GRASS Programs* [*p. 57*]. Here is a short overview.

For most programs a user runs a front-end program out of the GRASS bin directory which in turn looks for the existence of standard, alpha, and contributed interactive and non-interactive versions of the program. 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*(*p. 135*) before calling *G_parser*(*p. 134*) (see *§12.15.3 Parser Routines* [*p. 134*] 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*(*p. 134*) and *G_define_flag*(*p. 134*).

"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.

12.16. 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 *
G_strcpy (dst, src) *copy strings*

char *dst, *src;

Copies the **src** string to **dst** up to and including the NULL which terminates the **src** string. Returns **dst.**

char *
G_strncpy (dst, src, n) *copy strings*

char *dst, *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 *
G_strcat (dst, src) *concatenate strings*

char *dst, *src;

Appends the **src** string to the end of the **dst** string, which is then NULL terminated. Returns **dst.**

These next 2 routines remove unwanted white space from a single string.

char *
G_squeeze (s) *remove unnecessary white space*

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.**

G_strip (s) *remove leading/training white space*

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.

This next routine copies a string to allocated memory.

char *
G_store (s) *copy string to allocated memory*

This routine allocates enough memory to hold the string **s,** copies **s** to the allocated memory, and returns a pointer to the allocated memory.

These 2 routines convert between upper and lower case.

char *
G_tolcase (s) *convert string to lower case*

char *s;

Upper case letters in the string **s** are converted to their lower case equivalent. Returns **s.**

char *
G_toucase (s) *convert string to upper case*

char *s;

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 (c) *printable version of control character*

unsigned char c;

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*(*p. 152*) for printing the user's interrupt character:

```
char G_intr_char( );
char *G_unctrl( );

printf("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.

12.17. 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.

12.17.1. 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 program into the background. For example, a program 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.[35] Thus a program which puts itself in the background may never finish if the user interrupts another program 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.

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.

12.17.2. Partially Interruptible System Call

The UNIX system( ) call allows one program, the parent, to execute another UNIX command or program 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 program itself. The G_system( ) call allows this.

[35] 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) *run a shell level command*

The shell level **command** is executed. Interrupt signals for the parent program 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. Miscellaneous

A number of general purpose routines have been provided.

char *
G_date ( ) *current date and time*

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.

G_gets (buf) *get a line of input (detect ctrl-z)*

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 program to reprompt when a program is restarted after being stopped with a ctrl-z. If this routine returns 0, then the calling program should reprint a prompt and call *G_gets* ( ) again. For example:

```
char buf[1024];

do {
    printf("Enter some input :  ") ;
} while ( ! G_gets(buf) )  ;
```

char *
G_home ( ) *user's home directory*

Returns a pointer to a string which is the full path name of the user's home directory.

char
G_intr_char ( ) *return interrupt 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_unctrl*(*p. 149*) for converting this character to a printable format.

G_percent (n, total, incr) *print percent complete messages*

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.

char *
G_program_name ( ) *return program name*

This routine returns the name of the program as set by the call to *G_gisinit*(*p. 70*).

char *
G_whoami ( ) *user's name*

Returns a pointer to a string which is the user's login name.

G_yes (question, default) *ask a yes/no question*

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.

12.19. Deleted Routines

The following routines have been deleted from the GIS Library:

G_parse_command( )
G_parse_command_usage( );
G_set_parse_command_usage( );

Replaced by *G_parser*(*p. 134*) and *G_usage*(*p. 134*).

G_make_histo_grey_scale( )

Replaced by *G_make_histogram_eq_colors*(*p. 116*).

12.20. 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.

12.20.1. 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. 26*] for more information about raster headers, and *§9.1 Region* [*p. 49*] for more information about regions.

The routines described in *§12.10.1 Raster Header File* [*p. 106*] use this structure.

12.20.2. 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. 108*].

12.20.3. 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. 111*] must be used to store and retrieve color information using this structure.

12.20.4. 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  25
# 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 *§12.10.4 Raster History File* [*p. 116*] 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.

12.20.5. 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 *§12.10.5 Raster Range File* [*p. 117*] should be used to access this structure.

12.21. 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 Programs* [*p. 57*] for a complete discussion of Gmakefiles.

Chapter 13

Vector Library

13.1. Introduction

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. 15*] for a general description of GRASS databases, and *§6 Vector Maps* [*p. 33*] 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 programs 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 programs which use this prefix.[3]

An alphabetic index is provided in *§25.4 Appendix D. Index to Vector Library* [*p. 299*].

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]

---

[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_Line_Ptr,* and *Mem_curr_position.*

[4] The GRASS compilation process, described in *§11 Compiling and Installing GRASS Programs* [*p. 57*], automatically tells the C compiler how to find this and other GRASS header files.

```
#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. 33*] 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 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

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 program using the new **Vect Library.**

But also see *§13.9 Loading the Vector Library* [*p. 170*].

[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.

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 programs 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.

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 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 programs 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 programs 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 program change.

13.2.4. Implementation

There are two macros set up for use in the Gmakefile to support the Vector library:

EXTRA_CFLAGS = $(VECT_INCLUDE)
: must exist in the Gmakefile for any program which uses the Vector library. NOTE: GRASS 3.1 required the line **-I$(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 program 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 */
```

[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.

```
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 level 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

Vect_open_old (Map, name, mapset) *open existing vector map*

struct Map_info *Map;
char *name, *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.

Vect_open_new (Map, name) *open new vector map*

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.

Vect_set_open_level (level) *specify level for opening map*

int level;

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.

Vect_close (Map) *close a vector map*

struct Map_info *Map;

This routine closes an open vector map and cleans up the structures associated with it. It MUST be called before exiting the program. When used in conjunction with *Vect_open_new*(*p. 161*), 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

Vect_read_next_line (Map, Points) *read next vector line*

struct Map_info *Map;
struct line_pnts *Points;

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(*p. 163*)
Vect_set_constraint_region(*p. 163*)
Vect_set_constraint_type(*p. 163*)

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)*.

Vect_rewind (Map) *rewind vector map for re-reading*

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*(*p. 162*).

Vect_set_constraint_region (Map, n, s, e, w) *set restricted region to read vector arcs from*

struct Map_info *Map;
double n, s, e, 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*(*p. 162*) 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.

Vect_set_constraint_type (Map, type) *specify types of arcs to read*

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(*p. 162*) 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.

Vect_remove_constraints (Map) *unset any vector read constraints*

struct Map_info *Map;

Removes all constraints currently affecting **Map**.

long
Vect_write_line (Map, type, Points) *write out arc to vector map*

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*(*p. 161*). 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.

V1_read_line (Map, Points, offset) *read vector arc by specifying offset*

struct Map_info *Map;
struct line_pnts *Points;
long 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*(*p. 162*).

V2_read_line (Map, Points, line) *read vector arc by specifying line id*

struct Map_info *Map;
struct line_pnts *Points;
int line;

This routine will read a line from the vector map at the specified line index in the map. Refer to *V2_num_lines*(*p. 167*) 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*(*p. 162*).

13.5. Data Structures

struct line_pnts *
Vect_new_line_struct ( ) *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.

Vect_destroy_line_struct (Points) *deallocate line points structure space*

struct line_pnts *Points;

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*(*p. 165*).

13.6. Data Conversion

Vect_copy_xy_to_pnts (Points, x, y, n) *convert xy arrays to line_pnts structure*

struct line_pnts *Points;
double *x, *y;
int n;

Since all Vector library routines require the use of the line_pnts structure, and many programs 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.

- 166 -

Vect_copy_pnts_to_xy (Points, x, y, n) *convert line_pnts structure to xy arrays*

struct line_pnts *Points;
double *x, *y;
int *n;

Since all Vector library routines require the use of the line_pnts structure, and many programs 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.

Vect_copy_head_data (from, to) *copy vector header struct data*

struct dig_head *from, *to;

This routine should be used to copy data **from** one dig_head struct **to** another. For example, if a 3.1 program 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*(*p. 162*) was called. This routine must used because there are now other fields in the head structure which applications programmers should not touch, and this program copies only those fields which are available to the programmer.

13.7. Miscellaneous

Vect_get_area_points (Map, area, Points) *get defining points for area polygon*

struct Map_info *Map;
int area;
struct line_pnts *Points;

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.
Note. This function, works only for level 2 or higher.

It returns the number of points or -1 on error.

V2_num_lines (Map) *get number of arcs in vector map*

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.

V2_num_areas (Map) *get number of areas in vector map*

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.

V2_line_att (Map, line) *get attribute number of arc*

struct Map_info *Map;
int line;

Given arc index n, return its attribute number.

Returns 0 if not labeled or on error.

V2_area_att (Map, area) *get attribute number of area*

struct Map_info *Map;
int area;

Given area index n, return its attribute number.

Returns 0 if not labeled or on error.

V2_get_area (Map, n, pa) *get area info from id*

```
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.

V2_get_area_bbox (Map, area, n, s, e, w) *get bounding box of area*

```
struct Map_info *Map;
int area;
double *n, *s, *e, *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.

V2_get_line_bbox (Map, line, n, s, e, w) *get bounding box of arc*

```
struct Map_info *Map;
int line;
double *n, *s, *e, *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.

Vect_print_header (Map) *print header info to stdout*

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.

Vect_level (Map) *get open level of vector map*

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]

13.8. Routines that remain from GRASS 3.1

dig_point_to_area (Map, x, y) *find which area point is in*

struct Map_info *Map;
double x, y;

Returns the index of the area containing the point **x,y**, or 0 if none found.

double
dig_point_in_area (Map, x, y, pa) *is point in area?*

struct Map_info *Map;
double x, 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*(*p. 168*).

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.

[8] The levels correspond to the 3.1 level 1 and level 2 accesses.

dig_point_to_line (Map, x, y, type) *find which arc point is closest to*

struct Map_info *Map;
double x, 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.

dig_check_dist (Map, n, x, y, d) *find distance of point to line*

struct Map_info *Map;
int n;
double x, 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*(*p. 164*) 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 Programs* [*p. 57*] for a complete discussion of Gmakefiles.

 - 172 -

Chapter 14

Imagery Library

14.1. Introduction

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. 15*] for a general description of GRASS databases, *§8 Image Data: Groups* [*p. 43*] for a description of imagery groups, and *§5 Raster Maps* [*p. 23*] for details about map layers in GRASS.

The routines in the *Imagery 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 programs 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 programs which use this prefix.

---

[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 Programs* [*p. 47*] for a list of some imagery programs.

[3] The GRASS compilation process, described in *§11 Compiling and Installing GRASS Programs* [*p. 57*], automatically tells the C compiler how to find this and other GRASS header files.

An alphabetic index is provided in *§25.4 Appendix E. Index to Imagery Library* [*p. 301*].

14.2. Group Processing

The group is the key database structure which permits integration of image processing in GRASS.

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.

I_ask_group_old (prompt, group) *prompt for an existing group*

char *prompt;
char *group;

Asks the user to enter the name of an existing **group** in the current mapset.

I_ask_group_new (prompt, group) *prompt for new group*

char *prompt;
char *group;

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

[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[50].*

I_ask_group_any (prompt, group) *prompt for any valid group name*

```
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*(*p. 175*) 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);
```

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:

I_find_group (group) *does group exist?*

```
char *group;
```

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. 180*].

The contents of the REF file are read or updated by the following routines:

I_get_group_ref (group, ref) *read group REF file*

```
char *group;
struct Ref *ref;
```

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).

I_put_group_ref (group, ref) *write group REF file*

```
char *group;
struct Ref *ref;
```

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.

I_get_subgroup_ref (group, subgroup, ref) *read subgroup REF file*

```
char *group;
char *subgroup;
struct Ref *ref;
```

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).

I_put_subgroup_ref (group, subgroup, ref) *write subgroup REF file*

```
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:

I_init_group_ref (ref) *initialize Ref structure*

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*(*p. 176*) and *I_get_subgroup_ref*(*p. 176*) call this routine automatically.

I_add_file_to_group_ref (name, mapset, ref) *add file name to Ref structure*

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. 180*].

I_transfer_group_ref_file (src, n, dst) *copy Ref lists*

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.

I_free_group_ref (ref) *free Ref structure*

struct Ref *ref;

This routine frees memory allocated to the **ref** structure.

14.2.4. TARGET File

The following two routines read and write the TARGET file.

I_get_target (group, location, mapset) *read target information*

char *group;
char *location;
char *mapset;

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.

I_put_target (group, location, mapset) *write target information*

char *group;
char *location;
char *mapset;

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 program *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. 180*].

Note. The interface to the *Control_Points* structure provided by the routines below is incomplete. A routine to initialize the structure is needed.

I_get_control_points (group, cp) *read group control points*

```
char *group;
struct Control_Points *cp;
```

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.

I_new_control_point (cp, e1, n1, e2, n2, status) *add new control point*

```
struct Control_Points *cp;
double e1, n1;
double e2, 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]

I_put_control_points (group, cp) *write group 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.

[6] Use of this routine implies that the point is probably good, so **status** should be set to 1.

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. 69*] for details on that library.

See *§11 Compiling and Installing GRASS Programs* [*p. 57*] for a complete discussion of Gmakefiles.

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]).

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++)
   printf ("%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:

---

[7] The *red,grn,blu* elements are expected to change as the imagery code develops. Do **not** reference them. Pretend they do not exist.

[8] The *nfiles* and *file* elements are not expected to change in the future.

 - 182 -

```
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*. Control point *i* is *e1* [i], *n1* [i], *e2* [i], *n2* [i], and its status is *status* [i].

Chapter 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 program 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 programs port to new machines or devices. Because no device-dependent code exists in application programs, virtually all GRASS graphics programs port without modification. Each graphics device must be provided a driver (or translator program). At run-time, GRASS graphics programs rendezvous with a user-selected driver program. 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*(*p. 186*), *R_screen_rite*(*p. 186*), *R_screen_bot*(*p. 186*), and *R_screen_top*(*p. 186*).

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

 - 184 -

variables or routines in their own programs which use this prefix.

An alphabetic index is provided in *§25.4 Appendix G. Index to Raster Graphics Library* [*p. 305*].

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.

R_open_driver ( ) *initialize graphics*

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 program sends a message to the user stating that a driver has not been selected or could not be opened. Note that only one application program can be connected to a graphics driver at once.

After all graphics have been completed, the driver should be closed.

R_close_driver ( ) *terminate graphics*

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 programs 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.

R_color_table_fixed ( ) *select fixed color table*

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.

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 program so that the colors involved point to the closest available color on the device.

R_reset_color (red, green, blu, num) *define single color*

```
unsigned char red, green, blue ;
int num ;
```

Sets color number **num** to the intensities represented by **red, green,** and **blue.**

R_reset_colors (min,max,red,green,blue) *define multiple colors*

```
int min, max ;
unsigned char *red, *green, *blue ;
```

Sets color numbers **min** through **max** to the intensities represented in the arrays **red, green,** and **blue.**

R_color (color) *select color*

```
int color ;
```

Selects the **color** to be used in subsequent draw commands.

 - 186 -

R_standard_color (color) *select standard color*

int color ;

Selects the standard **color** to be used in subsequent draw commands. The **color** value is best retrieved using *D_translate_color*(*p. 207*). See *§16 Display Graphics Library* [*p. 195*].

R_RGB_color (red,green,blue) *select color*

int red, green, blue ;

When in float mode (see *R_color_table_float*(*p. 185*)), 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.

R_screen_bot ( ) *bottom of screen*

Returns the pixel row number of the bottom of the screen.

R_screen_top ( ) *top of screen*

Returns the pixel row number of the top of the screen.

R_screen_left ( ) *screen left edge*

Returns the pixel column number of the left edge of the screen.

R_screen_rite ( ) *screen right edge*

Returns the pixel column number of the right edge of the screen.

R_move_abs (x,y) *move current location*

int x, y;

Move the current location to the absolute screen coordinate **x,y.** Nothing is drawn on the screen.

R_move_rel (dx,dy) *move current location*

int dx, 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.

R_cont_abs (x,y) *draw line*

int x, y;

Draw a line using the current color, selected via *R_color*(*p. 185*), from the current location to the location specified by **x,y.** The current location is updated to **x,y.**

R_cont_rel (dx,dy) *draw line*

int dx, dy;

Draw a line using the current color, selected via *R_color*(*p. 185*), from the current location to the relative location specified by **dx** and **dy.** The current location is updated:

```
Newx = Oldx + dx;
Newy = Oldy + dy;
```

R_box_abs (x1,y1,x2,y2) *fill a box*

int x1,y1;
int x2,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.**

R_box_rel (dx,dy) *fill a box*

int dx, 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;
```

R_erase ( ) *erase screen*

Erases the entire screen to black.

R_flush ( ) *flush graphics*

Send all pending graphics commands to the graphics driver. This is done automatically when graphics input requests are made.

R_stabilize ( ) *synchronize graphics*

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*(*p. 188*) 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.

R_polydots_abs (x,y,num) *draw a series of dots*

```
int *x, *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.

R_polydots_rel (x,y,num) *draw a series of dots*

```
int *x, *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.

R_polygon_abs (x,y,num) *draw a closed polygon*

```
int *x, *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.

R_polygon_rel (x,y,num) *draw a closed polygon*

```
int *x, *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.

R_polyline_abs (x,y,num) *draw an open polygon*

```
int *x, *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.

R_polyline_rel (x,y,num) *draw an open polygon*

```
int *x, *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, being principally a raster-based data system, requires efficient drawing of raster information to the display device. These calls provide that capability.

R_raster (num,nrows,withzero,raster) *draw a raster*

int num, nrows, 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.

R_set_RGB_color (red,green,blue) *initialize graphics*

unsigned char red[256], green[256], 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*(*p. 190*) 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.

R_RGB_raster (num,nrows,red,green,blue,withzero) *draw a raster*

int num, nrows, withzero ;
unsigned char *red, *green, *blue ;

This is useful only in fixed color mode (see *R_color_table_fixed*(*p. 185*)). 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*(*p. 190*). 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.

R_set_window (top,bottom,left,right) *set text clipping frame*

int top, bottom, left, right ;

Subsequent calls to *R_text*(*p. 192*) will have text strings clipped to the screen frame defined by **top, bottom, left, right.**

R_font (font) *choose 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

R_text_size (width, height) *set text size*

int width, height ;

Sets text pixel width and height to **width** and **height.**

R_text (text) *write text*

char *text ;

Writes **text** in the current color and font, at the current text width and height, starting at the current screen location.

R_get_text_box (text, top, bottom, left, right) *get text extents*

char *text ;
int *top, *bottom, *left, *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. 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.

R_get_location_with_pointer (nx,ny,button) *get mouse location using pointer*

int *nx, *ny, *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.

R_get_location_with_line (x,y,nx,ny,button) *get mouse location using a line*

int x, y;
int *nx, *ny, *button ;

Similar to *R_get_location_with_pointer*(*p. 192*) 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.

R_get_location_with_box (x,y,nx,ny,button) *get mouse location using a box*

```
int x, y;
int *nx, *ny, *button ;
```

Identical to *R_get_location_with_line*(*p. 192*) except a rubber-band box is used instead of a rubber-band line.

15.9. 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. 69*] for details on that library.

This library is usually loaded with the $(DISPLAYLIB). See *§16 Display Graphics Library* [*p. 195*] for details on that library.

See *§11 Compiling and Installing GRASS Programs* [*p. 57*] for a complete discussion of Gmakefiles.

 - 194 -

Chapter 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 programs 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 programs which use this prefix.

An alphabetic index is provided in *§25.4 Appendix F. Index to Display Graphics Library* [*p. 303*].

16.2. Library Initialization

The following routine performs a required setup procedure. Its use is encouraged and simplifies the use of this library.

[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.

D_setup (clear) *initialize/create a frame*

int clear

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;
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.

D_new_window (name, top, bottom, left, right) *create new graphics frame*

char *name ;
int top, bottom, left, 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.**

D_set_cur_wind (name) *set current graphics frame*

char *name ;

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.

D_get_cur_wind (name) *identify current graphics frame*

char *name ;

Captures the name of the current frame in string **name.**

D_show_window (color) *outlines current frame*

int color ;

Outlines current frame in **color.** Appropriate colors are found in $GISBASE/src/D/libes/colors.h[2] and are spelled with lowercase letters.

D_get_screen_window (top, bottom, left, right) *retrieve current frame coordinates*

int *top, *bottom, *left, *right ;

Returns current frame's coordinates in the pointers **top, bottom, left,** and **right.**

---

[2] $GISBASE is the directory where GRASS is installed. See *§10.1 UNIX Environment* [*p. 53*] for details.

D_check_map_window (region) *assign/retrieve current map region*

struct Cell_head *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 program. Otherwise **region** is associated with the current graphics frame.

Note this routine is called by *D_setup*(*p. 196*).

D_reset_screen_window (top, bottom, left, right) *resets current frame position*

int top, bottom, left, right ;

Re-establishes the screen position of a frame at the location specified by **top, bottom, left,** and **right.**

D_timestamp ( ) *give current time to frame*

Timestamp the current frame. This is used primarily to identify which frames are on top of other, specified frames.

D_erase_window ( ) *erase current frame*

Erases the frame on the screen using the currently selected color.

D_remove_window ( ) *remove a frame*

Removes any trace of the current frame.

D_clear_window ( ) *clears information about current frame*

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.

D_add_to_list (string) *add command to frame display 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*(*p. 199*) routine is used for this special case.

D_set_cell_name (name) *add raster map name to display list*

char *name ;

Stores the raster[3] map **name** in the information associated with the current frame.

D_get_cell_name (name) *retrieve raster map name*

char *name ;

Returns the **name** of the raster map associated with the current frame.

D_clear_window ( ) *clear frame display lists*

Removes all display information lists associated with the current frame.

16.5. Coordinate Transformation Routines

These routines provide coordinate transformation information. GRASS graphics programs typically work with the following three coordinate systems:

Coordinate system	Origin	
Array	upper left	(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*(*p. 200*) 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.

[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.

D_do_conversions (region, top, bottom, left, right) *initialize conversions*

struct Cell_head *region ;
int top, bottom, right, left ;

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*(*p. 196*).

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).

double
D_u_to_a_row (north) *earth to array (north)*

double north ;

Returns a *row* value in the array coordinate system when provided the corresponding **north** value in the earth coordinate system.

double
D_u_to_a_col (east) *earth to array (east)*

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 (row) *array to screen (row)*

double row ;

Returns a *y* value in the screen coordinate system when provided the corresponding **row** value in the array coordinate system.

 - 201 -

double
D_a_to_d_col (column) *array to screen (column)*

double 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 (north) *earth to screen (north)*

double 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 (east) *earth to screen (east)*

double 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 (y) *screen to earth (y)*

double 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 (x) *screen to earth (x)*

double x ;

Returns an *east* value in the earth coordinate system when provided the corresponding **x** value in the screen coordinate system.

double
D_d_to_a_row (y) *screen to array (y)*

double y ;

Returns a *row* value in the array coordinate system when provided the corresponding **y** value in the screen coordinate system.

double
D_d_to_a_col (x) *screen to array (x)*

double x ;

Returns a *column* value in the array coordinate system when provided the corresponding **x** value in the screen coordinate system.

int
D_reset_color (data, r, g, b) *reset raster color value*

CELL data ;
int r, g, 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*(*p. 203*). *D_reset_color*(*p. 202*) is for use by programs 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*(*p. 203*).

int
D_check_colormap_size (min,max,ncolors) *verify a range of colors*

CELL min, 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.

void
D_lookup_colors (data, n, colors) *change to hardware color*

    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*(*p. 203*).

void
D_color (cat, colors) *select raster color for line*

    CELL cat ;
    struct Colors #colors ;

D_color specifies a raster color to use for line drawing. See *R_color*(*p. 185*) 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.

D_set_colors (colors) *establish raster colors for graphics*

    struct Colors *colors;

This routine sets the colors to be used for raster graphics. The **colors** structure must be either be read using *G_read_colors*(*p. 111*) or otherwise prepared using the routines described in *§12.10.3 Raster Color Table* [*p. 111*].

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*(*p. 203*) or *D_color*(*p. 203*) must be used for that purpose.

D_cell_draw_setup (top, bottom, left, right) *prepare for raster graphics*

int top, bottom, left, right ;

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*(*p. 197*) for the current frame.

D_draw_cell (row, raster, colors) *render a raster row*

int row ;
CELL *raster ;
struct Colors *colors;

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*(*p. 203*).

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.

D_set_overlay_mode (flag) *configure raster overlay mode*

int flag ;

This routine determines if *D_draw_cell*(*p. 204*) 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.

D_raster (raster, n, repeat, colors) *low level raster plotting*

CELL *raster;
int n, 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*(*p. 203*).

Note. This routine does not perform resampling or placement. *D_draw_cell*(*p. 204*) 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 "gis.h"
plot_raster_map(name,mapset,overlay)
        char *name, *mapset;
```

```
        int overlay;
{
        struct Colors colors;
        CELL *raster;
        int row, fd, top, bottom, left, right;

/* perform plotting setup */
        D_setup(!overlay);
        D_get_screen_window(&top, &bottom, &left, &right);
        if (D_cell_draw_setup(&top, &bottom, &left, &right)) {ERROR}
        raster = G_allocate_cell_buf( );

/* open raster map, read and set the colors */
        if((fd = G_open_cell (name, mapset)) < 0) {ERROR}
        if(G_read_colors (name, mapset), &colors) < 0) {ERROR}
        D_set_colors(&colors);

/* plot */
        D_set_overlay_mode(overlay);
        for(row=0; row >= 0; ) {
                if (G_get_map_row(fd, raster, row) < 0) {ERROR}
                row = D_draw_cell(row, raster, &colors);
        }
        G_close_cell(fd);
        G_free_colors(&colors);
        free(raster);
}
```

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.

D_clip (s, n, w, e, x, y, c_x, c_y) *clip coordinates to window*

double s, n, w, e;
double *x1, *y1, *x2, *y2 ;

A line represented by the coordinates **x1,y1** and **x2,y2** 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 **x1** and **x2** are values to be compared to **w** and **e.** The **y1** and **y2** are values to be compared to **s** and **n.**

The **x1** and **x2** values returned lie between **w** and **e.** The **y1** and **y2** values returned lie between **s** and **n.**

16.8. Pop-up Menus

D_popup (bcolor, tcolor, dcolor, top, left, size, options) *pop-up menu*

```
int bcolor ;
int tcolor ;
int dcolor ;
int left, top ;
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 program.

16.9. Colors

D_reset_colors (colors) *set colors in driver*

```
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. 111*] for GIS Library routines which use this structure.

D_translate_color (name) *color name to number*

char *name ;

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. Deleted Routines

The following routines have been deleted from the DISPLAY Library:

D_parse_command( )
D_usage( );

Replaced by *G_parser*(*p. 134*) and *G_usage*(*p. 134*).

D_reset_colors( )

Replaced by *D_reset_color*(*p. 202*) and *D_set_colors*(*p. 203*).

D_draw_cell_row( )
D_overlay_cell_row( )

Replaced by *D_draw_cell*(*p. 204*) and *D_set_overlay_mode*(*p. 204*).

16.11. 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. 183*] for details on that library. Also $(RASTERLIB) uses routines in $(GISLIB). See *§12 GIS Library* [*p. 69*] for details on that library.

See *§11 Compiling and Installing GRASS Programs* [*p. 57*] for a complete discussion of Gmakefiles.

16.12. 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.

16.12.1. DISPLAYLIB routines

D_setup (clear) *graphics frame 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 program region agree (may change active program 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.

D_set_clip_window (top, bottom, left, right) *set clipping window*

int top, bottom, left, right ;

Sets the clipping window to the pixel window that corresponds to the current database region. This is the default.

D_set_clip_window_to_map_window ( ) *set clipping window to map window*

Sets the clipping window to the pixel window that corresponds to the current database region. This is the default.

D_cont_abs (x, y) *line to x,y*

int x, 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.

D_cont_rel (x, y) *line to x,y*

int x, y ;

Equivalent to *D_cont_abs*(*p. 209*)(curx+x, cury+y) where **curx,cury** is the current pixel location.

D_move_abs (x, y) *move to pixel*

int x, y ;

Move without drawing to pixel location **x,y,** even if it falls outside the clipping window.

D_move_rel (x, y) *move to pixel*

int x, y ;

Equivalent to *D_move_abs*(*p. 209*)(curx+x, cury+y) where **curx,cury** is the current pixel location.

- 210 -

 - 211 -

Chapter 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 Synopses

lock_file (file, pid) *create a lock*

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

unlock_file (file) *remove a lock*

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.

17.3. Use 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" program 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 program 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.

17.4. 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:

[1] This file is .gislock.

 - 213 -

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 Programs* [*p. 57*] for a complete discussion of Gmakefiles.

- 214 -

 - 215 -

Chapter 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 program 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 program 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 **rowio_**. To avoid name conflicts, programmers should not create variables or routines in their own programs which use this prefix.

An alphabetic index is provided in *§25.4 Appendix H. Index to Rowio Library* [*p. 307*].

18.2. Rowio Routine Synopses

The routines in the *Rowio Library* are described below. They use a data structure called ROWIO which is defined in the header file "rowio.h" that must be included in any code using these routines:[1]

```
#include "rowio.h"
```

rowio_setup (r, fd, nrows, len, getrow, putrow) *configure rowio structure*

```
ROWIO *r;
int fd, nrows, 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.

[1] The GRASS compilation process, described in *§11 Compiling and Installing GRASS Programs* [*p. 57*], automatically tells the C compiler how to find this and other GRASS header files.

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.

char *
rowio_get (r, n) *read a row*

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*(*p. 216*) 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*(*p. 218*), the **putrow( )** routine specified in *rowio_setup*(*p. 216*) 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**.

rowio_forget (r, n) *forget a row*

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*(*p. 217*) is later modified directly without also writing it to the file. See *§18.3 Rowio Programming Considerations* [*p. 218*].

rowio_fileno (r) *get file descriptor*

ROWIO *r;

Rowio_fileno( ) returns the file descriptor associated with the ROWIO structure.

rowio_release (r) *free allocated memory*

ROWIO *r;

Rowio_release( ) frees all the memory allocated for ROWIO structure **r.** It does not close the file descriptor associated with the structure.

rowio_put (r, buf, n) *write a row*

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*(*p. 216*) 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*(*p. 218*) 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 program to check.

18.4. Loading the Rowio Library

The library is loaded by specifying $(ROWIOLIB)[2] 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 Programs* [*p. 57*] for a complete discussion of Gmakefiles.

[2] This variable was NOT defined in releases 3.0 and 3.0A. Edit the file *$GISBASE/src/CMD/make.mid* and add the line **ROWIOLIB= $(LIBDIR)/ rowio.a** at the bottom of the file.

- 220 -

Chapter 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 program 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. However 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 programs which use this prefix.

An alphabetic index is provided in *§25.4 Appendix I. Index to Segment Library* [*p. 309*].

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 program. 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:

segment_format (fd, nrows, ncols, srows, scols, len) *format a segment file*

int fd, nrows, ncols, srows, scols, 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 if 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*(*p. 222*).

[1] The GRASS compilation process, described in *§11 Compiling and Installing GRASS Programs* [*p. 57*], automatically tells the C compiler how to find this and other GRASS header files.

segment_init (seg, fd, nsegs) *initialize segment structure*

SEGMENT *seg;
int fd, nsegs;

Initializes the **seg** structure. The file on **fd** is a segment file created by *segment_format*(*p. 222*) 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 *segment_format*(*p. 222*), 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:

segment_put_row (seg, buf, row) *write row to segment file*

SEGMENT *seg;
char *buf;
int row;

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*(*p. 223*). **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:

segment_get (seg, value, row, col) *get value from segment file*

SEGMENT *seg;
char *value;
int row, col;

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.

 - 224 -

segment_put (seg, value, row, col) *put value to segment file*

```
SEGMENT *seg;
char *value;
int row, col;
```

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:

segment_flush (seg) *flush pending updates to disk*

```
SEGMENT *seg;
```

Forces all pending updates generated by *segment_put*(*p. 224*) 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*(*p. 224*).

Now the data in segment file can be read row by row and transferred to a normal sequential data file:

segment_get_row (seg, buf, row) *read row from segment file*

```
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*(*p. 223*). **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:

segment_release (seg) *free allocated memory*

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*(*p. 224*) 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*(*p. 222*) will fill the segment file with zeros.

The data can now be accessed directly using *segment_get*(*p. 223*). For example, to get the value at a given row and column:

 - 226 -

```
int value;
SEGMENT seg;

segment_get (&seg, &value, row, col);
```

Similarly *segment_put*(*p. 224*) 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.

 - 227 -

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 Programs* [*p. 57*] for a complete discussion of Gmakefiles.

- 228 -

Chapter 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 programs.[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 programs which use this prefix.

An alphabetic index is provided in *§25.4 Appendix J. Index to Vask Library* [*p. 311*].

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 program. 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.

---

[1] The functions in this library make use of the curses library and termcap descriptions. 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.*

V_clear ( ) *initialize screen description*

This routine initializes the screen description information, and must be called before each new screen layout description.

V_line (num, text) *add line of text to screen*

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.

V_const (value, type, row, col, len) *define screen constant*

V_ques (value, type, row, col, len) *define screen question*

Ctype *value; (*Ctype* is one of int, long, float, double, or char)
char type;
int row, col, len;

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]

V_float_accuracy (num) *set number of decimal places*

int num;

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).

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( ).

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.

[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.

V_intrpt_msg (text) *change ctrl-c message*

char *text;

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 program 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 */

   printf ("%d %f %s\n", i, f, s) ;                 /* ESC, so print results */
   exit(0);
}
```

The user is presented with the following screen:

 - 233 -

```
Enter an Integer                    0____

Enter a Decimal                     0.00_

Enter a character string            _________


AFTER COMPLETING ALL ANSWERS, HIT <ESC> 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 program with a return value of 1.
CTRL-C	returns control to the calling program 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 program reloading. However, the compile rule specifies $(OBJ) and $(VASK), rather than $(OBJ) and $(VASKLIB). This is because $(VASK) specifies both the UNIX curses and termcap libraries as well as $(VASKLIB).

See *§11 Compiling and Installing GRASS Programs* [*p. 57*] 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:

 - 235 -

```
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 US Army CERL

which is a product of the US Army CERL






AFTER COMPLETING ALL ANSWERS, HIT <ESC> 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 program using the *Vask Library* should not call curses library routines directly.**

[4] Specifically, memory allocated by initscr( ) was not freed by endwin( ).

- 236 -

Chapter 21

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.

This chapter is relevant for the GRASS v4.1 version of **v.digit** only. The GRASS v4.0 version of v.digit is now named **v.digit2** and is included in the GRASS v4.1 release. See *§22 Writing a Digitizer Driver* [*p. 251*] for information on writing a digitizer for **v.digit2**.

21.1. 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

 - 238 -

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 #?secton|digitizer.file.commands

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/track-ball) 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:

21.2. Digitizer Configuration File Commands

The following is an in-depth description of each command available in the .dgt digitizer files.

21.2.1. Setup

This data is used to setup the communication link with the digitizer and is used during interpretation of the digitizer data.

21.2.1.1. Serial Line Characteristics

baud = n   This line is optional, default = 9600 if not specified. If

 - 239 -

	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.

21.2.1.2. 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.

 - 241 -

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.

21.2.1.3. 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
}
```

21.2.2. 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 program 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.

21.2.2.1. 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 program 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.

21.2.2.2. 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 programs 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 program.

v.digit requires that a STARTQUERY group exists.

21.2.2.3. Example of Start Groupings

```
startrun
{
 send = \027%R
 baud = 2400
 wait = 0.6
 read = 3
 wait = 0.1
 send = \027%S
}


startpoint
{
   send = \027%^12\013  # set output format to format 12
   send = \027%P\013             # set to run mode
}


startquery
{
   send = \027%^12\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
}
```

21.2.2.4. Example of a Query Grouping

```
query
{
  send = !\013      #  send prompt
}
```

21.2.2.5. Example of a Stop Grouping

```
stop
{  send = \027%K
   wait = 0.1
   send = \027%*
}
```

21.2.3. 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 program 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).

21.2.3.1. 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).

21.2.3.2. 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.

21.2.3.3. Example of ASCII Format Grouping

```
format
{
   ascii =?xxxxx,yyyyy,??bcl
}
```

21.2.3.4. 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 4th 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.

 - 246 -

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).

21.2.3.5. 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 9th parity bit).

0s and 1s are used for syncing the input, and should all occur in the same bit column.

21.2.3.6. 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
```

 - 247 -

```
 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
}
```

21.3. Examples of Complete Files

The following are complete examples of digitizer files.

21.3.1. Example 1

```
setup
{
   digname = Calcomp
   description = Calcomp digitizer, ascii format 5
   buttonoffset = 1
   buttons = 16
   buttonstart = 0
   baud = 9600
   units_per_inch = 1000
}
```

- 248 -

```
startrun
{
   send = \027%^5\013    # set to format 5
   send = \027%R\013
}


startpoint
{
   send = \027%^5\013    # set to format 5
   send = \027%P\013
}


startquery
{
   send = \027%^5\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
}
```

21.3.2. Example 2

 - 249 -

```
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
}
```

 - 250 -

```
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
}
```

21.4. 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.

Chapter 22

Writing a Digitizer Driver

22.1. Introduction

This chapter is relevant only for **v.digit2**. For more information on configuration files for the GRASS v4.1 v.digit, and an explanation of **v.digit** and **v.digit2**, see *§21 Digitizer/Mouse/Trackball Files (.dgt)* [*p. 237*].

A digitizer device driver consists of a library of device-dependent functions that are linked into digitizer programs. This chapter describes those functions that are needed to create a digitizer device driver compatible with GRASS map development software.

Section *§22.2 Writing the Digitizer Device Driver* [*p. 251*] explains how digitizer drivers are written, while section *§22.3 Discussion of the Finer Points (Hints)* [*p. 259*] describes problems and pitfalls encountered during the development of the Altek driver.

22.2. Writing the Digitizer Device Driver

Source code for the digitizer drivers is kept in

$GISBASE/src/mapdev/digitizers[1]

Separate subdirectories contain the individual drivers. When a new driver is written, it should be placed here in a new subdirectory.

It is helpful to examine the source code for existing drivers located here, and to attend a demonstration of the GRASS digitizing program *v.digit*, before developing a new driver.

[1] $GISBASE is the directory where GRASS is installed. See *§10.1 UNIX Environment* [*p. 53*] for details.

22.2.1. Functions that must be Written

This section describes the device-dependent library functions that must be written. Each of these functions must be present in the library. Function descriptions are organized by file name. (The file names are those used by current GRASS digitizer drivers. File names are printed in bold, along the left-hand margin of the page.) These files and functions can be copied from one of the existing digitizer driver libraries and altered to suit the needs of a particular driver.

Note. Although it is strongly recommended that the programmer use the file names listed below (for reasons set forth in *§22.2.3 Compiling the Device Driver* [*p. 258*]), other files names may be used instead.

dig_menu.h

This file contains the menu that is displayed while digitizing. The menu should indicate the purpose of the buttons on the cursor for the particular digitizer. The menu is stored in **dig_menu**:

```
char *dig_menu[ ] ;
```

An example of how the Altek driver uses this function to create a menu is given below:

```
# define dig_menu_lines        16

char  *dig_menu[ ] = {
"   GRASS-DIGIT Version 3.0                                  Digitizing menu   ",
"                                                                               ",
"   ALTEK digitizer                  | AMOUNT DIGITIZED                         ",
"      Cursor keys:                  |   # Lines:                               ",
"      <0>  digitize point           |   # Area edges:                          ",
"      <1>  quit digitizing          |                                          ",
"      <2>  update monitor           |                                          ",
"      <3>  toggle point/stream mode |   Total points:                          ",
"                                                                               ",
"                                    |  CURRENT DIGITIZER PARAMS.               ",
"                                    |                                          ",
"                                    |                                          ",
"                                    |  MODE          TYPE                      ",
"                                    |   point         line                     ",
"                                    |   stream        area edge                ",
"                                    |                                          "
} ;
```

Note. The menu must be exactly as it appears here, except that the text in **bold** may be replaced by the appropriate text for the digitizer.

dig_curses.c

This file only contains # includes. It is used to set up the digitizing menu in the "dig_menu.h" file. This file must look like this:

```
#include <curses.h>

#include "dig_menu.h"
#include "../../digit/digit.h"
#include "../../digit/menu.h"
#include "../../libes/head.h"

#include "../../digit/curses.c"
```

setup_driver.c

```
D_setup_driver (device)
      char  *device ;
```

This function opens the device (which is a *tty* port) and initializes the digitizer.

Note. This function should not set the origin. The origin is set later by the function *D_setup_origin*(*p. 255*).

dig_dev.c

```
D_get_scale(scale)
      float  *scale ;
```

This function sets **scale** to the digitizer resolution in units of lines per inch.[2] For example, on a digitizer having a resolution of 1000 lines per inch, **scale** would be set to .001.

coll_pts.c

```
#include   "digit.h"
#include   "globals.h"

collect_points (mode, type, np, x, y)
      int  mode,  type ;
      int  *np ;
      double  **x,  **y ;
```

This routine is called to collect points that represent a single vector (or arc) from the digitizer.

The points should be collected into static arrays or dynamically allocated arrays, transformed from digitizer coordinates to database coordinates using *transform_a_into_b*(*p. 257*), and plotted on the graphics monitor using *plot_points*(*p. 257*). Then **x** and **y** are set to point to these arrays, and **np** set to the number of points collected.

[2] Almost all digitizers describe their resolution in lines per inch (lpi). This is essentially equivalent to pixels per inch, or dots per inch.

The digitizing **mode** may be either STREAM or POINT: STREAM indicates that the digitizer should collect a continuous stream of points; POINT indicates that the digitizer should collect points under user control (i.e., each time the user presses a button, the foot-switch, or a key on the keyboard). The *collect_points*( ) function can be written to allow interactive toggling between the two modes during a single call.

The **type** is set to AREA when the vector to be collected is an area edge, and to LINE when it is is a linear feature. The **type** is of no interest to *collect_points*( ) itself, but is passed to the function *plot_points*(*p. 257*), which draws lines on the graphics monitor.

This function should return 1 if digitizing in STREAM mode occurred (i.e., either because **mode** was initially STREAM, or because the user changed to STREAM mode), and 0 otherwise.[3]

Note. This routine is responsible for plotting the vector on the graphics monitor, but it should do it responsibly. This means that while digitizing in POINT mode, the line-segments should be plotted immediately; while digitizing in STREAM mode, the points should be plotted only when the digitizing is finished, or when the user toggles to POINT mode.

Note. If the cursor has buttons, they can be used to change the digitizing **mode** as well as end the digitizing. If the digitizer has a foot-switch instead of buttons, the foot-switch should be used to end the digitizing (toggling modes would not be supported in this case). If the digitizer has neither buttons nor a foot-switch, then the keyboard must be used, even in STREAM mode. (See GeoGraphics driver for code that polls the keyboard.)

interface.c

This file contains a number of functions. The following functions return information about digitizer capabilities:

D_cursor_buttons( )

If the digitizer cursor buttons are to be used by the digitizing programs, there must be at least five buttons. This function returns 1 if the cursor has five or more buttons; otherwise, it returns 0.

D_foot_switch( )

This function returns 1 if there is a usable foot-switch. It returns 0 if the digitizer has no foot-switch.

Note. If there are five or more buttons on the cursor, the value returned by

[3] STREAM mode indicates to *digit* that the resulting vector should be pruned.

D_foot_switch( ) is ignored (i.e., it is assumed that there is no foot-switch). See *D_cursor_buttons*(*p. 254*).

D_start_button( )

This function tells the driver how the cursor buttons are labeled (i.e., the labels that the user sees on the buttons). If the first button is labeled 1, then this routine returns 1. If the first button is labeled 0, then this routine returns 0.

It should return -1 if the digitizer cursor buttons are not being used by the driver. See *D_cursor_buttons*(*p. 254*).

For example, if the digitizer buttons are labeled 0-9, then this routine would return 0. If the digitizer buttons are labeled 1-16, then this routine would return 1.

The following routines perform digitizer configuration:

D_setup_origin( )

This routine sets the digitizer's origin (0,0). This routine should only return if successful, and should return a value of 0. If it fails, an error message should be sent to the terminal screen with *Write_info*(*p. 257*), and the program terminated with a call to *close_down*(*p. 256*).

Note. Frequently, the location of the digitizer's origin can be set to some default value, without any input from the user. Otherwise, this routine must ask the user to set the origin. The routine *Write_info*(*p. 257*) should be used to print instructions for the user. (Refer to the GeoGraphics digitizer driver, which instructs users to set the origin in the lower left corner of the digitizing tablet.)[4]

D_clear_driver( )

This function clears any button presses on the digitizer that have been queued. (Refer to *§22.3 Discussion of the Finer Points (Hints)* [*p. 259*] for an explanation of why this is necessary.) This routine should only return if successful, and should return a value of 0. If it fails, an error message should be sent to the user with *Write_info*(*p. 257*), and the program terminated with a call to *close_down*(*p. 256*).

The following two routines read the current digitizer coordinates:

---

[4] Due to the design of the GeoGraphics digitizer, it is not possible to detect whether or not the user properly sets the origin. If the origin is improperly set, the map will be improperly registered.

 - 256 -

D_read_raw (x, y)
double *x, *y ;

Gets the current location of the digitizer cursor, and places the digitizer coordinates in the variables **x** and **y.**

If a digitizer button was pressed, this routine returns the button's value. The return value must be in the range of 1 through 16. This means that if the first button is labeled 0 this routine must add 1 to the button number that is returned.

If no button was pressed, this routine returns 0.

Foot-switch. If the digitizer has a foot-switch, instead of cursor buttons, then the foot-switch must be treated as if it were button 1. If the digitizer has neither a foot-switch nor cursor buttons, then this routine should return 0.

D_ask_driver_raw (x, y)
double *x, *y ;

Waits for a button to be pressed and then gets the current location of the digitizer cursor, and places the digitizer coordinates in the variables **x** and **y.**

This routine returns the button's value. The return value must be in the range of 1 through 16. This means that if the first button is labeled 0 this routine must add 1 to the button number that is returned.

Foot-switch. If the digitizer has a foot-switch, instead of cursor buttons, then the foot-switch must be treated as if it were button 1, and this routine should wait for the foot-switch to be pressed. If the digitizer has neither a foot-switch nor cursor buttons, then this routine should return 0 *without* waiting.

22.2.2. Functions Available For Use

There are functions which have already been written that can be called by the digitizer driver. These are described below.

Note. These functions exist in libraries. The libraries that contain these functions are described in *§22.2.3 Compiling the Device Driver* [*p. 258*].

close_down (status)
int status ;

This function gracefully exits the calling program. Call this function with **status** set to -1 when an irrecoverable error has occurred (e.g., when the digitizer does not respond, or returns an error). Otherwise, call this routine with **status** set to 0.

plot_points (type, np, x, y, line_color, point_color)

int type, np;
double *x, *y ;
int line_color, point_color ;

This function is to be called by *collect_points*(*p. 253*). It draws the vector defined by the points in the **x** and **y** arrays on the graphics monitor. The number of points in the vector is **np.**

The *plot_points* ( ) function expects to receive points from *collect_points*(*p. 253*) in the coordinate system of the database. Digitizer coordinates can be translated to database coordinates using *transform_a_into_b*(*p. 257*).

The **type** indicates whether the vector is an AREA or a LINE. AREA and LINE are defined in the include file "dig_defines.h".

The **line_color** and **point_color** indicate whether the lines and points are to be highlighted or erased. The constant CLR_HIGHLIGHT indicates highlighting, and the constant CLR_ERASE indicates erase (CLR_HIGHLIGHT and CRL_ERASE are defined in "globals.h"). The colors actually used to highlight or to erase lines and points are specified by the user in *digit.*

transform_a_into_b (Xraw, Yraw, X, Y)

double Xraw, Yraw ;
double *X, *Y ;

This function converts the digitizer coordinates **Xraw,Yraw** into the database coordinates **X,Y.** This function is used by the driver function *collect_points*(*p. 253*).

Note. The transformation rule used by this routine is generated by *digit* when the user registers the map to the database. The rule is already in place by the time *collect_points*(*p. 253*) calls *transform_a_into_b* ( ).

Write_info (line, message)

int line ;
char *message ;

This function prints a **message** in the four line window at the bottom of the user's terminal in *digit*. The variable **line** must be a number 1 through 4, which represents the line number inside the window. The **message** must not exceed 76 characters and should not contain \n.

22.2.3. Compiling the Device Driver

Programs (e.g., *digit*) that use the digitizer driver functions are stored in libraries. When the digitizer driver is compiled, it links with those different libraries and creates the programs. Each driver should contain a *Gmakefile* that contains compilation instructions for *gmake.*[5] The *Gmakefile* for the digitizer driver is complex. Rather than attempting to construct a completely new *Gmakefile,* it is generally simpler to copy an existing *Gmakefile* from another driver and modify it to meet the needs of the new digitizer driver.

The following libraries are needed by the digitizer driver when it is compiled:

$GISBASE/src/mapdev/digit/libdigit.a
$GISBASE/src/mapdev/libes/libtrans.a
$GISBASE/src/mapdev/lib/libdig.a
$LIBDIR/libdig_atts.a

Some include files (*.h) must also be compiled into the driver. These files are located in the following directories:

$GISBASE/src/mapdev/libes
$GISBASE/src/mapdev/lib

Compile the device driver by executing *gmake.* This will create the *digit* program and any other programs dependent on the digitizer driver code.

22.2.4. Testing the Device Driver

There are three crucial points at which the *digit* program calls the digitizer driver. The first occurs just after *digit* has prompted the user for a file name. *Digit* will try to open the driver and initialize the digitizer; if this fails, it is because *D_setup_driver*(*p. 253*) has failed. The second occurs when the user registers the map to the digitizer. If the program fails at this point, there is a problem with the *D_read_raw*(*p. 256*) function. A final test of the driver is performed when the *collect_points*(*p. 253*) function is called, which occurs when vectors are being digitized.

Before testing any programs, review the *Grass Installation Guide* to ensure that the digitizer is set up correctly. If more information is needed, read the file $GISBASE/src/mapdev/README.

[5] See *§11 Compiling and Installing GRASS Programs* [*p. 57*] for a discussion of *gmake* and *Gmakefiles.*

22.3. Discussion of the Finer Points (Hints)

This section offers several hints and pitfalls to avoid when writing the digitizer driver. It has three subsections: Setting up the Digitizer, Program Logic, and Specific Driver Issues.

22.3.1. Setting up the Digitizer

The process of setting up a computer system and digitizer can be divided into three steps:

(1) Setting the internal switches on the digitizer (hardware)
(2) Running a cable between the digitizer and the computer (hardware)
(3) Setting up the serial port on the computer (software)

22.3.1.1. Setting the internal switches

The switches on the digitizer must be set so that the digitizer will run under *request* or *prompt* mode, which means that the digitizer will only send output when it is requested or prompted by the program. Thus, the program controls the timing of the output from the digitizer and will only receive information when it is ready to process it. Refer to the manual included with the digitizer for specific information on its set-up.

Note. The digitizer must be able to use an RS232 serial interface and transmit information only when prompted by the program. If the digitizer cannot transmit information on command, then it cannot be used as a GRASS digitizer.

22.3.1.2. Running a cable between the digitizer and computer

A cable must be made to connect the digitizer to a RS232 serial port on the computer. Different model computers, even when from the same maker, may require different cable configurations. For example, one computer may need a straight-through cable, while another computer may need pins 6, 8, and 20 looped back on the computer side. A break-out box can be used to deduce digitizer cable requirements and ensure that the digitizer is actually talking to the computer.

22.3.1.3. Configuring the serial port

The digitizer is plugged into a serial port (*/dev/tty??*) on the computer, which must be configured for a digitizer to run on it. To set up the *tty* for the digitizer, turn that *tty*'s getty off, and make the *tty* readable and writable by anyone.

A final suggestion: document the information that has been learned. The file *$GISBASE/src/mapdev/digitizers/altek/INSTALL.ALTEK* can be used as an example. It contains the switch settings for the Altek, cable configurations, and other useful information. Such documentation is invaluable when another digitizer is added, problems arise, or if the digitizer switch settings have to be changed because other software is using the digitizer.

22.3.2. Program Logic

All digitizing programs follow the same basic steps, whether they test the digitizer, or appear in a complex digitizing program like *digit*. The following sequence gives the programmer a feel for how the digitizer driver is used by the calling programs.

(1) Link the program to the digitizer (open the *tty*)
(2) Set the *tty* to the appropriate state (ioctl calls)
(3) Initialize the digitizer (setting resolution, setting origin, ...)
(4) Ask the digitizer for data containing a set of coordinates
(5) Read the data from the digitizer
(6) Interpret the data into usable coordinates (x, y)
(7) Display the coordinates (x, y)
(8) Loop back for more data or until user wants to quit

In order to become familiar with the architecture of a digitizer driver, it is useful to write a simple program to test the digitizer. If a digitizing problem arises, the diagnostic program can help isolate the cause of the problem (hardware, software, cable, etc.).

22.3.3. Specific Driver Issues

The writing of digitizer device drivers can be complex. This section explores four issues in greater depth:

(1) Connecting to the digitizer
(2) Initializing and reading the digitizer
(3) Synchronizing the digitizer and computer
(4) Digitizer cursors with buttons

Connecting to the digitizer :

In GRASS, the computer communicates directly with the digitizer to which (through the serial port *tty*) the digitizer is connected. The *tty* to which the digitizer is connected is opened, read, and written to just like a file.

D_setup_driver(*p. 253*) will open the *tty,* set file permissions to read and write, and set the running state of the *tty.* Some experimenting with the different line disciplines (CBREAK, RAW) may be necessary to determine the best state for the *tty,* but RAW seems to be the norm. Changing the running state of a *tty* consists of changing the structures associated with that particular *tty* and reflecting the changes to the operating system by using *ioctl*( ). Unfortunately, the information is stored differently under different operating systems.

GRASS digitizer drivers have been written under the System V (AT&T) and Berkeley (UCB) UNIX operating systems. A major difference between these two operating systems is the way they handle terminal interfaces (ttys). Terminal information is contained in structures in <termio.h> under System V, and in <sgtty.h> under Berkeley. In other words, the structures, and the names used in the structures, will differ depending on the operating system. All *tty* related system-dependent code has C preprocessor *#ifdef SYSV*[6] statments around it in the existing drivers. System-dependent code is defined as either being under System V (SYSV) or Berkeley. This issue will only arise when the *tty* to which the digitizer is connected is being opened, using *D_setup_driver*(*p. 253*).

Initializing and reading the digitizer:

The driver and the digitizer communicate by using the UNIX *read*( ) and *write*( ) functions. *D_setup_driver*(*p. 253*) sets up the digitizer software by writing command strings to the *tty.* Since each digitizer is different, the digitizer's user manual frequently proves to be the only source of information on how to initialize and read the digitizer.

Setting up a consistently good function to read the digitizer is the most difficult part of writing the digitizer driver. The *read*( ) function, when reading from a *tty,* may not read as many characters as requested. For example, if six bytes are requested, *read*( ) can return anywhere from zero to six bytes.

One approach is to request six bytes, and then, if the number of bytes actually read is not six, issue another *read*( ), this time asking only for the number of bytes remaining. In other words, if six bytes were requested but only two were received, then another read for four bytes is issued. If that read returned one byte, then another read is requested for three bytes, etc. This would continue until either all six bytes were read, or a timeout occurred. This approach worked well in the Altek driver.

Another approach that was tried was to request six bytes, and then, if less than six bytes were received, the bytes were thrown away, and another six bytes were requested. This was repeated until the read returned six bytes. This approach worked some of the time, but sometimes gave unreliable coordinates, and was abandoned. Other digitizer drivers have been written that read ascii characters from the digitizer and use *sscanf*( ) to strip out the needed information.

The number of characters actually read to get one set of coordinates will depend on the digitizer and on the information stated in the digitizer's user manual.

Another problem, in the case of the Altek, is that the cursor is only active in certain portions of the tablet. This means that either there will be no output, or a specific flag will be on/off, until the cursor is within the active area of the tablet. Because no external markings on the tablet delineate the active area, individuals commonly

[6] SYSV is defined by *gmake.* See *§11 Compiling and Installing GRASS Programs* [*p. 57*].

attempt to digitize within the tablet's inactive area, leading them to the false assumption that the digitizer is acting strangely. Depending on the digitizer, this will have to be handled by fine tuning the reads and/or checking the status byte(s).

A word of warning - if the *tty* is not set up properly in *D_setup_driver*(*p. 253*), the *read*( ) function can return confusing information (i.e., it may include garbage with the data or be unable to read the number of characters specified).

Synchronizing the digitizer and computer :

Driver checking has been added to post-3.0 drivers, to warn the user when the driver is out of sync with the digitizer. For example, the Altek has the high bit turned *on* in the first byte of the six bytes that are read. The driver checks to make sure that the high byte is turned *on* ; if it is not, the digitizer and driver are out of sync. The driver warns the user, resets the digitizer and then reinitializes the digitizer.

Digitizer cursors with buttons:

Drivers can be written to use the digitizer buttons or the keyboard for input while digitizing. Where drivers use the digitizer buttons, some digitizers will queue up any button hits. (This may depend on what running state the digitizer was set up with when it was initialized.) This means that if a person pushes the digitizer cursor buttons a number of times and then begins to digitize, the program must clear the queue of button hits before beginning to digitize. Other digitizers will only say that a button has been hit if the button has been hit *and* the digitizer has been prompted for a coordinate.

Chapter 23

Writing a Graphics Driver

23.1. Introduction

GRASS application programs 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 program 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 programs to choose from. They are managed by the program *d.mon.*

Porting GRASS graphics programs from device to device simply requires the creation of a new graphics driver program. Once completed and working, all GRASS graphics programs will work exactly as they were designed without modification (or recompilation). This section is concerned with the creation of a new graphics driver.

23.2. 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., 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.

---

[1] $GISBASE is the directory where GRASS is installed. See *§10.1 UNIX Environment* [*p. 53*] for details.

23.3. 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).

23.3.1. 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.

23.3.2. Return Edge and Color Values

The four raster edge values set in the *Graph_Set( )* routine above are retrieved with the following routines.

 - 265 -

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.

23.3.3. 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.

23.3.4. 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*

```
int 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.

23.3.5. 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.**

23.3.6. 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.

23.4. Optional Routines

All of the above must be created for any new driver. The GRASS *Rasterlib,* which provides the application program 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 programs 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.

 - 270 -

Chapter 24

Writing a Paint Driver

24.1. 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 program 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 program. The executable program 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 program.

The user first selects a printer using the *p.select* program. 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.

24.2. 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

[1] See *§10.2 GRASS Environment* [*p. 54*].

[2] $GISBASE is the directory where GRASS is installed. See *§10.1 UNIX Environment* [*p. 53*] for details.

program, the shell script, and a *Gmakefile* with rules that tell the GRASS *gmake* command how to compile the driver.[3]

24.3. The Paint Driver Executable Program

A *paint* device driver program 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.

24.3.1. 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.

---

[3] See *§11 Compiling and Installing GRASS Programs* [*p. 57*] for details on the GRASS compilation process.

[4] This, and other, environment variables are set in the driver shell script which is described in *§24.4 The Device Driver Shell Script* [*p. 277*].

Poutc (c) *write a character to printer*

unsigned char c;

Sends the character **c** to the printer. This routine can be implemented as follows:

```
Poutc(c) unsigned char c;
{
  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.

24.3.2. Initialization

The following routine will be called after *Popen*(*p. 272*) to initialize the printer:

[5] See *§24.6 Paint Driver Library* [*p. 280*].

Pinit ( ) *initialize the printer*

Initializes the printer. Sends whatever codes are necessary to get the printer ready for printing.

24.3.3. 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 honor *Ptext*(*p. 274*) 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.

24.3.4. 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*(*p. 275*).[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*(*p. 275*). 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.

---

[6] The programmer should, of course, code defensively. If the number of columns is too large, the driver should exit with an error message.

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*(*p. 275*). 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.

24.3.5. 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*(*p. 272*) 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.

---

[7] See *§24.8 Creating 125 Colors From 3 Colors* [*p. 282*].

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*(*p. 276*).

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*

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).

24.4. 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*(*p. 272*). Only in very special cases can drivers justify either ignoring this value or allowing it not to be set.

The drivers distributed by USACERL 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.

---

[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.

[9] The driver shell script may set any other variables that the programmer has determined the driver needs.

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*(*p. 272*) 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.

Note. The application programs 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 have 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).

[10] Apologies are offered for this admittedly awkward design.

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
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
```

24.5. Programming Considerations

The *paint* driver uses its standard input and standard output to communicate with the *paint* application program. It is very important that neither the driver shell nor the driver program write to stdout or read from stdin.

Diagnostics, error messages, etc., should be written to stderr. There is an error routine which driver programs 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.

24.6. 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*(*p. 272*), *Pout*(*p. 272*), *Poutc*(*p. 273*), *Pouts*(*p. 273*), *Pflush*(*p. 273*), and *Pclose*(*p. 273*) 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*(*p. 276*), *Pcolorlevels*(*p. 276*), *Pcolornum*(*p. 277*), and *Pcolorvalue*(*p. 277*) for the 125 color logic described in *§24.8 Creating 125 Colors From 3 Colors* [*p. 282*].

24.7. 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* :

---

[11] See *§24.7 Compiling the Driver* [*p. 280*] for an example of how to load this library code.

[12] See *§11 Compiling and Installing GRASS Programs* [*p. 57*] for details on the GRASS compilation process.

```
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 \
                      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*(*p. 272*), *Pout*(*p. 272*), *Poutc*(*p. 273*), *Pouts*(*p. 273*), *Pflush*(*p. 273*), and *Pclose*(*p. 273*) for serial and parallel ports. It also loads *colors125.o* which contains versions of *Pncolors*(*p. 276*), *Pcolorlevels*(*p. 276*), *Pcolornum*(*p. 277*), and *Pcolorvalue*(*p. 277*) for 125 colors.

lock library (LOCKLIB)
: The driver loads the lock library. This is a GRASS library which must be loaded if the *Popen*(*p. 272*) 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

[13] See also *§24.6 Paint Driver Library* [*p. 280*]).

in

$GISBASE/etc/paint/driver[14]

and the driver shell in

$GISBASE/etc/paint/driver.sh.

24.8. 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*(*p. 276*), *Pcolorlevels*(*p. 276*), *Pcolornum*(*p. 277*), and *Pcolorvalue*(*p. 277*), provided that the color *numbers* are assigned as follows:

```
color_number = red_pixels * 25 + green_pixels * 5 + blue_pixels ;
```

[14] $GISBASE is the directory where GRASS is installed. See *§10.1 UNIX Environment* [*p. 53*] for details.

[15] See *§24.6 Paint Driver Library* [*p. 280*].

Chapter 25

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.

25.1. 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.

25.2. 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. 53*]

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.

25.3. 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.

25.4. 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

[2] The other variables will be empty as well.

 - 285 -

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.

- 286 -

Appendix A

Annotated Gmakefile Predefined Variables

The predefined Gmakefile variables are defined in the files *head* 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 Programs* [p. 57].

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* file:

Variable	Value	Description
ARCH	= sun3	Architecture to compile on
GISBASE	= /usr/grass4.1	Location of GRASS program
UNIX_BIN	= /usr/local/bin	Miscellaneous GRASS commands
DEFAULT_DATABASE	= /usr/grass/data	Location of default database
DEFAULT_LOCATION	= spearfish	Name of defaut database
COMPILE_FLAGS	= -O	Compiler flags
LDFLAGS	= -s	Loader flags
DIGIT_FLAGS	=	
MATHLIB	= -lm	Math libraries
TERMLIB	= -ltermlib	Terminal emulation libraries
CURSES	= -lcurses $(TERMLIB)	Curses libraries
LIBRULE	= ar ruv $@ $?; ranlib $@	Library archiver
#LIBRULE	= ar rc $@ ‘lorder $(OBJ) | tsort‘	Alternate form of library archiver command
#USE_TERMIO	= -DUSE_TERMIO	Use TERMIO or not?
USE_MTIO	= -DUSE_MTIO	Use MTIO?

[1] $GISBASE is the directory where GRASS is installed. See *§10.1 UNIX Environment* [p. 53] for details.

 - 288 -

Variable	Value	Description
DIGITFLAGS	=	

make.mid

The *make.mid* file uses the variables in *makehead* 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
GARDEN_BIN	= $(GISBASE)/garden/bin	Garden commands
GARDEN_ETC	= $(GISBASE)/garden/etc	
BIN_MAIN_INTER	= $(ETC)/bin/main/inter	Main interactive commands
BIN_MAIN_CMD	= $(ETC)/bin/main/cmd	Main command-line commands
BIN_ALPHA_INTER	= $(ETC)/bin/alpha/inter	Alpha interactive
BIN_ALPHA_CMD	= $(ETC)/bin/alpha/cmd	Alpha command-line
BIN_CONTRIB_INTER	= $(ETC)/bin/contrib/inter	Contributed interactive
BIN_CONTRIB_CMD	= $(ETC)/bin/contrib/cmd	Contributed command-line
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 $@; $(LIBRULE)	All library archiver flags

- 289 -

Variable	Value	Description
MANROFF	= tbl -TX $(SRC)/man.help/man.version $(SRC)/man.help/man.header $? | nroff -Tlp | col -b > $@	Manual formatter command and options
MAKEALL	= $(GMAKE) -all	Command to make GRASS
LIBDIR	= $(SRC)/libes	GRASS libraries
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
GISLIB	= $(LIBDIR)/libgis.a	GIS libraries
ICONLIB	= $(LIBDIR)/libicon.a	
LOCKLIB	= $(LIBDIR)/liblock.a	
IMAGERYLIB	= $(LIBDIR)/libI.a	
ROWIOLIB	= $(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	

- 290 -

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., printf( ), 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 */

printf ("%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:[1]

```
char userbuf[128];
CELL c;
long x;

printf ("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.*

[1] This example does not check for valid inputs, EOF, etc., which good code must do.

 - 292 -

Appendix C

Index to GIS Library

Here is an index of GIS Library routines, with calling sequences and short function descriptions.

GIS Library

GIS Library

GIS Library

 - 296 -

GIS Library

- 297 -

GIS Library

 - 298 -

Appendix D

Index to Vector Library

Here is an index of vector Library routines, with calling sequences and short function descriptions.

vector Library

- 300 -

 - 301 -

Appendix E

Index to Imagery Library

Here is an index of Imagery Library routines, with calling sequences and short function descriptions.

Imagery Library

- 302 -

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

Display Graphics Library

 - 305 -

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

- 306 -

- 307 -

Appendix H

Index to Rowio Library

Here is an index of Rowio Library routines, with calling sequences and short function descriptions.

Rowio Library

- 308 -

 - 309 -

Appendix I

Index to Segment Library

Here is an index of Segment Library routines, with calling sequences and short function descriptions.

Segment Library

- 310 -

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	page
V_call	( )	interact with the user	231
V_clear	( )	initialize screen description	230
V_const	(value, type, row, col, len)	define screen constant	230
V_float_accuracy	(num)	set number of decimal places	231
V_intrpt_msg	(text)	change ctrl-c message	232
V_intrpt_ok	( )	allow ctrl-c	231
V_line	(num, text)	add line of text to screen	230
V_ques	(value, type, row, col, len)	define screen question	230

- 312 -

 - 313 -

Appendix K

Permuted Index for Library Subroutines

 - 318 -

 - 320 -

 - 322 -

- 324 -

- 325 -

 - 326 -

 - 329 -

 - 330 -

 - 335 -

Index

Throw this page away and replace it with the real index.

- 336 -

Table of Contents

 - iv -