GRASS 3.0 Programmer’s Manual

by
Michael Shapiro
James Westervelt
Dave Gerdes
Michael Higgins
Marjorie Larson

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

Approved for public release; distribution is unlimited.
The contents of this report are not to be used for advertising, publication, or promotional purposes. Citation of trade names does not constitute an official endorsement or approval of the use of such commercial products. The findings of this report are not to be construed as an official Department of the Army position, unless so designated by other authorized documents.

DESTROY THIS REPORT WHEN IT IS NO LONGER NEEDED

DO NOT RETURN IT TO THE ORIGINATOR
This manual introduces the reader to the Geographic Resources Analysis Support System from the programming perspective. Design theory, system support libraries, systems maintenance, and system enhancement are all presented.
Notice to Program Recipients

This program is furnished by the U.S. Government and is accepted and used by the recipient with the express understanding that the Government makes no warranty, expressed or implied, concerning the accuracy, completeness, reliability, usability, or suitability for any particular purpose of the information and data contained in this program or furnished in connection therewith, and the United States shall be under no liability whatsoever to any person by reason of any use made thereof.

The program belongs to the Government. Therefore, the recipient further agrees not to assert any proprietary rights therein or to represent this program to anyone as other than a Government program. The recipient also agrees that the program and all documents related thereto, including all copies and versions (except when expressly authorized otherwise) in possession thereof, will be discontinued from use or destroyed upon request by the Government.

The program is to be used only in the public interest and/or the advancement of science and will not be used by the recipient to gain unfair advantage over any client or competitor. Whereas the recipient may charge clients for the ordinary costs of applying the program, the recipient agrees not to levy a charge, royalty or proprietary usage fee (except to cover any normal copying and/or distribution costs) upon any client for the development or use of the received program. Recipients desiring to modify and remarket the program will be required to comply with a separate agreement. Only minor or temporary modifications will be made to the program (e.g., necessary corrections or changes in the format of input or output) without written approval from the Government. Should the program be furnished by the recipient to a third party the recipient is responsible to that third party for any support and upkeep of the program. Information on the source of the program will be furnished to anyone requesting such information.

The accuracy of this program depends entirely on user-supplied input data. It is the user’s responsibility to understand how the input data affects the program output and to use the output data only as intended.

All documents and reports conveying information obtained as a result of the use of the program by the recipient will acknowledge the Corps of Engineers, Department of the Army, as the origin of the program. All such documentation will state the name and version of the program used by the recipient.

<table>
<thead>
<tr>
<th>Accession For</th>
</tr>
</thead>
<tbody>
<tr>
<td>NTIS CRAI</td>
</tr>
<tr>
<td>DTIC TAB</td>
</tr>
<tr>
<td>Unannounced</td>
</tr>
<tr>
<td>Justification</td>
</tr>
</tbody>
</table>

| By |
| Distribution |

| Availability Code |
| Dist | Avail and/or Special |
| A-1 |   |
Foreword

This work was performed for the U.S. Army Engineering and Housing Support Center (USAEHSC) under the work units A896-NN-TS9 entitled "Imagery Data for Training Area Management," A896-NN-TF9 entitled "Terrain Modeling for Planning Training Events and Natural Resources Management," and A896-NN-TJ9 entitled "GRASS Application Macros." Additional sponsorship came from the U.S. Department of Agriculture Soil Conservation Service (SCS) Cartographic and Geographic Information Systems Division under the work unit RD8P69NVC8 entitled "Enhancements to SCS-GRASS." The USAEHSC Technical Monitor was Ms. Jamie Clark of the Natural and Cultural Resources Division. The SCS Technical Monitor was Mr. Dick Liston.

The authors would like to acknowledge Ms. Mary Martin for her assistance in reviewing, editing, and preparing this document.

This work was performed by the Environmental Division (EN) of the U.S. Army Construction Engineering Research Laboratory (USACERL). Dr. R.K. Jain is Chief of USACERL-EN.

COL Carl O. Magnell is Commander and Director of USACERL, and Dr. L.R. Shaffer is Technical Director.
Table of Contents

Chapter 1. Introduction ........................................................................................................... 1
  1.1. Background .................................................................................................................. 1
  1.2. Objective ..................................................................................................................... 1
  1.3. Approach .................................................................................................................... 1
  1.4. Scope ........................................................................................................................... 2
  1.5. Mode of Technology Transfer ..................................................................................... 2
  1.6. GRASS Information Center ....................................................................................... 4

Chapter 2. Development Guidelines .................................................................................. 5
  2.1. Intended GRASS Audience ....................................................................................... 5
  2.2. Programming Standards ............................................................................................. 6
  2.3. Documentation Standards .......................................................................................... 7

Chapter 3. Multi-Level ....................................................................................................... 9
  3.1. General User ............................................................................................................... 9
  3.2. GRASS Programmer ................................................................................................ 10
  3.3. Driver Programmer ................................................................................................... 12
  3.4. GRASS System Designer .......................................................................................... 13

Chapter 4. Database Structure ......................................................................................... 15
  4.1. Programming Interface .............................................................................................. 15
  4.2. Gisdbase .................................................................................................................... 16
  4.3. Locations ................................................................................................................... 16
  4.4. Mapsets ..................................................................................................................... 16
  4.5. Mapset Structure ....................................................................................................... 17
  4.5.1. Mapset Files .......................................................................................................... 17
  4.5.2. Elements ................................................................................................................ 18
  4.6. Permanent Mapset ..................................................................................................... 19
  4.7. Database Access Rules .............................................................................................. 20
  4.7.1. Mapset Search Path ............................................................................................... 20
  4.7.2. UNIX File Permissions ......................................................................................... 20

Chapter 5. Grid Cell Maps ................................................................................................. 23
  5.1. What is a Grid Cell Map Layer? ................................................................................. 23
  5.2. Grid Cell File Format ................................................................................................. 24
  5.3. Cell Header Format .................................................................................................... 26
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.3.1.</td>
<td>Regular Format</td>
<td>26</td>
</tr>
<tr>
<td>5.3.2.</td>
<td>Reclass Format</td>
<td>27</td>
</tr>
<tr>
<td>5.4.</td>
<td>Cell Category File Format</td>
<td>28</td>
</tr>
<tr>
<td>5.5.</td>
<td>Cell Color Table Format</td>
<td>28</td>
</tr>
<tr>
<td>5.6.</td>
<td>Cell History File</td>
<td>29</td>
</tr>
<tr>
<td>5.7.</td>
<td>Cell Range File</td>
<td>29</td>
</tr>
<tr>
<td>6.1.</td>
<td>What is a Vector Map Layer?</td>
<td>31</td>
</tr>
<tr>
<td>6.2.</td>
<td>Ascii Arc File Format</td>
<td>32</td>
</tr>
<tr>
<td>6.2.1.</td>
<td>Header Section</td>
<td>33</td>
</tr>
<tr>
<td>6.2.2.</td>
<td>Arc Section</td>
<td>34</td>
</tr>
<tr>
<td>6.3.</td>
<td>Vector Category Attribute File</td>
<td>35</td>
</tr>
<tr>
<td>6.4.</td>
<td>Vector Category Label File</td>
<td>36</td>
</tr>
<tr>
<td>6.5.</td>
<td>Vector Index and Pointer File</td>
<td>37</td>
</tr>
<tr>
<td>6.6.</td>
<td>Digitizer Registration Points File</td>
<td>37</td>
</tr>
<tr>
<td>6.7.</td>
<td>Vector Topology Rules</td>
<td>37</td>
</tr>
<tr>
<td>6.8.</td>
<td>Importing Vector Files Into GRASS</td>
<td>38</td>
</tr>
<tr>
<td>7.1.</td>
<td>What is a Site List?</td>
<td>39</td>
</tr>
<tr>
<td>7.2.</td>
<td>Site File Format</td>
<td>39</td>
</tr>
<tr>
<td>7.3.</td>
<td>Programming Interface to Site Files</td>
<td>40</td>
</tr>
<tr>
<td>8.1.</td>
<td>Introduction</td>
<td>41</td>
</tr>
<tr>
<td>8.2.</td>
<td>What is a Group?</td>
<td>41</td>
</tr>
<tr>
<td>8.2.1.</td>
<td>A List of Cell Files</td>
<td>42</td>
</tr>
<tr>
<td>8.2.2.</td>
<td>Image Registration and Rectification</td>
<td>42</td>
</tr>
<tr>
<td>8.2.3.</td>
<td>Image Classification</td>
<td>42</td>
</tr>
<tr>
<td>8.3.</td>
<td>The Group Structure</td>
<td>43</td>
</tr>
<tr>
<td>8.3.1.</td>
<td>The REF File</td>
<td>43</td>
</tr>
<tr>
<td>8.3.2.</td>
<td>The POINTS File</td>
<td>44</td>
</tr>
<tr>
<td>8.3.3.</td>
<td>The TARGET File</td>
<td>44</td>
</tr>
<tr>
<td>8.3.4.</td>
<td>Subgroups</td>
<td>44</td>
</tr>
<tr>
<td>8.4.</td>
<td>Imagery Programs</td>
<td>45</td>
</tr>
<tr>
<td>8.5.</td>
<td>Programming Interface for Groups</td>
<td>46</td>
</tr>
<tr>
<td>9.1.</td>
<td>Window</td>
<td>47</td>
</tr>
<tr>
<td>9.2.</td>
<td>Mask</td>
<td>48</td>
</tr>
<tr>
<td>9.3.</td>
<td>Variations</td>
<td>49</td>
</tr>
</tbody>
</table>
Chapter 10. Environment Variables .................................................. 51
10.1. UNIX Environment ................................................. 51
10.2. GRASS Environment ............................................. 52
10.3. Difference Between GRASS and UNIX Environments ............. 52
Chapter 11. Compiling GRASS Programs Using Gmake ................... 55
11.1. Gmake ............................................................ 55
11.2. Gmake Variables ................................................ 55
11.3. Constructing a Gmakefile ......................................... 58
11.3.1. Building programs from source (.c) files .................... 58
11.3.2. Include files ................................................. 59
11.3.3. Building object libraries .................................... 60
11.3.4. Building more than one target ............................... 61
11.3.5. Don't bypass .o files ...................................... 61
Chapter 12. GIS Library ...................................................... 63
12.1. Introduction ....................................................... 63
12.2. Library Initialization ........................................... 64
12.3. Diagnostic Messages ............................................ 64
12.4. Environment and Database Information ........................... 66
12.5. Fundamental Database Access Routines ........................... 68
12.5.1. Prompting for Database Files ................................ 68
12.5.2. Finding Files in the Database ................................ 70
12.5.3. Legal File Names ............................................. 72
12.5.4. Opening an Existing Database File for Reading ............... 72
12.5.5. Opening an Existing Database File for Update ............... 73
12.5.6. Creating and Opening a New Database File .................. 74
12.5.7. Database File Management ................................... 75
12.6. Memory Allocation ............................................... 75
12.7. The Window ....................................................... 76
12.7.1. The Database Window ....................................... 77
12.7.2. The Active Program Window ................................ 78
12.8. Cell File Processing ............................................. 80
12.8.1. Prompting for Cell Files ..................................... 81
12.8.2. Finding Cell Files in the Database ........................... 82
12.8.3. Opening an Existing Cell File ................................ 83
12.8.4. Creating and Opening New Cell Files ......................... 84
12.8.5. Allocating Cell I/O Buffers ................................ 85
12.8.6. Reading Cell Files .......................................... 86
12.8.7. Writing Cell Files ........................................... 87
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>13.2.1. Initialization/Termination</td>
<td>124</td>
</tr>
<tr>
<td>13.2.2. Reading Arcs</td>
<td>126</td>
</tr>
<tr>
<td>13.3. Level Two Read Access</td>
<td>127</td>
</tr>
<tr>
<td>13.3.1. Initialization/Termination</td>
<td>127</td>
</tr>
<tr>
<td>13.3.2. Area Retrieval</td>
<td>128</td>
</tr>
<tr>
<td>13.3.3. Arc Retrieval</td>
<td>130</td>
</tr>
<tr>
<td>13.3.4. Area Analysis Tools</td>
<td>131</td>
</tr>
<tr>
<td>13.3.5. Arc Analysis Tools</td>
<td>132</td>
</tr>
<tr>
<td>13.4. Writing Binary Dig files</td>
<td>133</td>
</tr>
<tr>
<td>13.5. Miscellaneous Tools</td>
<td>134</td>
</tr>
<tr>
<td>13.6. Loading the Dig Library</td>
<td>135</td>
</tr>
<tr>
<td>Chapter 14. Imagery Library</td>
<td>137</td>
</tr>
<tr>
<td>14.1. Introduction</td>
<td>137</td>
</tr>
<tr>
<td>14.2. Group Processing</td>
<td>138</td>
</tr>
<tr>
<td>14.2.1. Prompting for a Group</td>
<td>138</td>
</tr>
<tr>
<td>14.2.2. Finding Groups in the Database</td>
<td>139</td>
</tr>
<tr>
<td>14.2.3. REF File</td>
<td>139</td>
</tr>
<tr>
<td>14.2.4. TARGET File</td>
<td>142</td>
</tr>
<tr>
<td>14.2.5. POINTS File</td>
<td>143</td>
</tr>
<tr>
<td>14.3. Loading the Imagery Library</td>
<td>144</td>
</tr>
<tr>
<td>14.4. Imagery Library Data Structures</td>
<td>144</td>
</tr>
<tr>
<td>14.4.1. struct Ref</td>
<td>144</td>
</tr>
<tr>
<td>14.4.2. struct Control_Points</td>
<td>145</td>
</tr>
<tr>
<td>Chapter 15. Raster Graphics Library</td>
<td>147</td>
</tr>
<tr>
<td>15.1. Introduction</td>
<td>147</td>
</tr>
<tr>
<td>15.2. Connecting to the Driver</td>
<td>148</td>
</tr>
<tr>
<td>15.3. Colors</td>
<td>148</td>
</tr>
<tr>
<td>15.4. Basic Graphics</td>
<td>150</td>
</tr>
<tr>
<td>15.5. Poly Calls</td>
<td>152</td>
</tr>
<tr>
<td>15.6. Raster Calls</td>
<td>153</td>
</tr>
<tr>
<td>15.7. Text</td>
<td>154</td>
</tr>
<tr>
<td>15.8. User Input</td>
<td>156</td>
</tr>
<tr>
<td>15.9. Loading the Raster Graphics Library</td>
<td>157</td>
</tr>
<tr>
<td>Chapter 16. Display Graphics Library</td>
<td>159</td>
</tr>
<tr>
<td>16.1. Introduction</td>
<td>159</td>
</tr>
<tr>
<td>16.2. Window Management</td>
<td>159</td>
</tr>
<tr>
<td>16.3. Window Contents Management</td>
<td>161</td>
</tr>
<tr>
<td>16.4. Coordinate Transformation Routines</td>
<td>162</td>
</tr>
</tbody>
</table>
16.5. Raster Graphics ................................................................. 164
16.6. Window Clipping ............................................................... 165
16.7. Pop-up Menus ................................................................. 166
16.8. Colors .............................................................................. 167
16.9. Loading the Display Graphics Library .............................. 167

Chapter 17. Lock Library ................................................................. 169
17.1. Introduction ........................................................................ 169
17.2. Lock Routine Synopses ..................................................... 169
17.3. Use and Limitations .......................................................... 170
17.4. Loading the Lock Library ................................................. 170

Chapter 18. Rowio Library ............................................................... 173
18.1. Introduction ........................................................................ 173
18.2. Rowio Routine Synopses ................................................. 174
18.3. Rowio Programming Considerations ................................. 176
18.4. Loading the Rowio Library .............................................. 177

Chapter 19. Segment Library .......................................................... 179
19.1. Introduction ........................................................................ 179
19.2. Segment Routines ............................................................ 180
19.3. How to Use the Library Routines ........................................ 183
19.4. Loading the Segment Library ........................................... 185

Chapter 20. Vask Library ............................................................... 187
20.1. Introduction ........................................................................ 187
20.2. Vask Routine Synopses ..................................................... 187
20.3. An Example Program ....................................................... 190
20.4. Loading the Vask Library ................................................ 191
20.5. Programming Considerations .......................................... 192

Chapter 21. Writing a Digitizer Driver .......................................... 195
21.1. Introduction ........................................................................ 195
21.2. Writing the Digitizer Device Driver ................................... 195
21.2.1. Functions to be Written .............................................. 195
21.2.2. Functions Available For Use ....................................... 200
21.2.3. Compiling the Device Driver ...................................... 202
21.2.4. Testing the Device Driver .......................................... 202
21.3. Discussion of the Finer Points (Hints) ............................... 203
21.3.1. Setting up the Digitizer ............................................... 203
21.3.2. Program Logic ............................................................ 204
21.3.3. Specific Driver Issues ............................................... 204

Chapter 22. Writing a Graphics Driver ......................................... 207
Appendix I. Index to Segment Library ........................................ 253
Appendix J. Index to Vask Library ............................................. 255
Appendix K. Permutted Index for Library Subroutines .................. 257
Index ......................................................................................... 275
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 the development of new versions of GRASS.
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 by USACERL in 1987). However, there were numerous and fundamental changes made in GRASS 3.0 (released in 1988). Rather than revise the existing Programmer's Manual, USACERL researchers elected to draft a new and more complete GRASS Programmer's Reference Manual for GRASS 3.0. The approach used in the development of this manual involves a systematic effort to describe GRASS development guidelines, user interfaces, data structures, programming libraries and peripheral drivers.

1.4. Scope
Information in this manual is valid for GRASS version 3.0, released in November, 1988. 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 GRASS Clippings 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 Steering Committee,
- An annual GRASS/GIS User Group Meeting,
- GRASSClippings, a quarterly 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 Steering 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 Steering 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, approximately four times a year, 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.
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 4005, Champaign, IL, 61824-4005; or by electronic mail at: grass@cerl.cec.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.¹

INTERACTIVE -
GRASS has a strong interactive component. Its multi-level design allows users to work either at a very user-friendly level, at a more flexible command level, or at

¹ The UTM projection allows GRASS to assume equal area cells anywhere in the database. It also makes distance calculations simple and straightforward. This will change as future releases allow other coordinate systems (e.g., longitude/latitude). The changes will probably not affect overlay operations, but will most likely change the methodology for distance and area calculations.
a programming level.

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

FOR NON-PROGRAMMER -
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 be compilable 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 (UCB 4.2) 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:

Full Interactive Level -
The new and occasional user works at this level. As of the first writing of this document, only one program, the GRASS menu, exists at this level. It is expected that specialized models, natural language interfaces, graphic popup 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 Interactive Level -
This is the level most used. Using the user’s login shell, GRASS commands are made available through modification of the PATH variable. Commands
at this level are highly interactive. Help and on-line manual commands are available. Historically, these programs have included both user interface and program function capabilities. In the future, more and more commands at this level will actually contain only user interface code; after the user is thoroughly interrogated, a command line will be constructed which then drives a program at the Command Level:

Command Level -

Commands at this level form the G, D, P, etc. languages. They are distinguished by being non-interactive. All information necessary for the execution of the command is provided either in the command line or in the standard input stream (with no prompting). Built on top of these commands may be commands at either of the above two levels. The advanced user who wants greater flexibility in the analysis options may use these directly. Further, the system analyst can use these commands as a high-level GIS programming language in concert with other UNIX utilities.

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 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. Those functions most critical are those that interface the data. It is believed that these functions will be more permanent than the database. Though the database 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 liberally sprinkled with 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. Final documents have been kept separate from the GRASS directories, though it is suggested that they appear with appropriate "makefiles" under $GISBASE/tutorials.

---

2 This package, invoked with the -ms option to nroff, is documented in section 7 of the UNIX manual.
3 $GISBASE is the directory where GRASS is installed. See §10.1 UNIX Environment [p. 51] for details.
Chapter 3

Multi-Level

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

- Full Interactive Level
- Command Interactive Level
- 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, may be afraid of keyboards, know nothing of UNIX, and may struggle with the learning curve for GRASS. Such users should select the Full Interactive Level, where they are guided through the options 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. As of this writing the menu version of GRASS is the sole representative of this interface. 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 commands. These 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 available, most general users of GRASS interface the system through the Command Interactive Level level.
The Command Interactive 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 enters a very friendly and interactive environment. 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 Full Interactive Level tools will be:

a. Written for the occasional user;
b. Verbose in their prompting;
c. Have available lots 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, interacting 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 windowing and masking of cell 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, the GRASS programs are also written in C. While UNIX machines offer a Fortran 77,
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 §24 Writing GRASS Shell Scripts (p. 229).

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 Gmake -
GRASS code is compiled using the Gmake command, which is a front-end to the UNIX make utility. Gmake combines some pre-defined variables with a file called Gmakefile in the source directory to create a proper makefile, and then runs make to compile the program. Each source code directory must have a Gmakefile, written by the programmer, containing instructions for making the binary executables, manual and help entries, and other items from the directory's contents. The Gmakefile does not contain hard-coded references to programs, libraries, or directories outside the current directory. Variables defining these items are used instead. Gmakefiles remain identical system to system thus providing consistency for system installation and compilation. See §11 Compiling GRASS Programs Using Gmake (p. 55) for more details.

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. Each library has a definition in Gmake to aid in linking the library during program compilation and loading. The libraries are listed briefly below.

GIS Library. This library contains all of the routines necessary to read and write the GRASS grid cell data layers and their support files. A standardized method to prompt the user for map names is available. The library also provides some general purpose tools like memory allocation, buffer zeroing, string analysis, and data searching. Ninety-nine percent of all GRASS programs use routines from this library. See §12 GIS Library (p. 63).

Segment Library. For programs that need random access to an entire map layer,
the segment library provides an efficient paging scheme for grid cell maps. While virtual memory operating systems perform paging, this library provides better control and efficiency of paging. See §19 Segment Library (p. 179).

**Dig Library.** While GRASS is primarily a grid cell 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 grid cell 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 GRASS has chosen to implement its own internal vector format. The format is a variant of arc-node. The Dig Library provides access to the GRASS vector database. See §13 Dig Library (p. 123).

**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. 187).

**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 rastertlb, are involved in generating graphics. The rastertlb 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 rastertlb 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 window management routines, coordinate conversion capabilities, and grid cell data to raster graphic conversions. See §16 Display Graphics Library (p. 159) and §15 Raster Graphics Library (p. 147).

### 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 §22 Writing a Graphics Driver [p. 207].

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 §21 Writing a Digitizer Driver [p. 195].

Similarly, GRASS sites may wish to write code to support different hardcopy color printers (inkjet, thermal, etc.). See §23 Writing a Paint Driver [p. 215].

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.
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 the GIS Library, which interfaces with the GRASS database. It is described in detail in §12 GIS Library (p. 63). 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 hard-coded 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. 51) 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. 67), must be used instead.

¹ This file is grassr under GRASS 3.0.
² using gisenv; see §24 Writing GRASS Shell Scripts (p. 229)
4.2. Gis database

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. 67).

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. 66).

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

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

4.4. Map sets

Subdirectories under any location are known as map sets. Users select a map set when entering GRASS. New map sets can be created during the selection step. The selected map set is known as the current map set. It is named in the environment variable $MAPSET and returned by G_mapset(p. 66).

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

Modifications to the database can only be made in the current map set. Users may only select (and thus modify) a map set that they own (i.e., have created). However, data in all map sets 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 map set, they are actually selecting one of the map set directories.

Note. The full UNIX directory name for the current map set is $GISDBASE/$LOCATION_NAME/$MAPSET and is returned by the library routine.

§4 Database Structure
G\textunderscore location\_path(p. 67).

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 \textit{Permanent Mapset} (p. 19).

4.5. Mapset Structure

Mapsets will contain files and subdirectories, known as database \textit{elements}. In the diagram below, the elements are indicated by a trailing `/`.

\begin{center}
\begin{tabular}{|c|c|c|c|c|}
\hline
\text \[\text{MAPSET}\] & \text \[\text{SEARCH\_PATH}\] & \text \[\text{WIND}\] & \text \[\text{cats/}\] & \text \[\text{cell/}\] & \text \[\text{paint/}\] & \text \[\text{windows/}\] \\
\hline
\end{tabular}
\end{center}

4.5.1. Mapset Files

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

\begin{center}
\begin{tabular}{|l|l|}
\hline
\text \[\text{files}\] & \text \[\text{function}\] \\
\hline
\text \[\text{GROUP}\] & \text current\_imagery\_group \\
\text \[\text{SEARCH\_PATH}\] & \text mapset\_search\_path \\
\text \[\text{WIND}\] & \text current\_window \\
\hline
\end{tabular}
\end{center}

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 window. This file is created when the mapset is created and is modified by the \textit{window} command. The contents of WIND are returned by \textit{G\_get\_window(p. 77).} See §9.1 \textit{Window} (p. 47) for a discussion of the GRASS window.

SEARCH\_PATH contains the \textit{mapset search path}. This file is created and modified by the \textit{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 use the mapset search path to find database files. See §4.7.1 \textit{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. 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., Gask). 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.

---

3 See §12.5.6 Creating and Opening a New Database File (p. 74).
Here is list of some of the elements used by GRASS programs written at USACERL:

<table>
<thead>
<tr>
<th>element</th>
<th>function</th>
</tr>
</thead>
<tbody>
<tr>
<td>cell</td>
<td>data layers (cell files)</td>
</tr>
<tr>
<td>cellhd</td>
<td>header files for data layers</td>
</tr>
<tr>
<td>cats</td>
<td>category information for data layers</td>
</tr>
<tr>
<td>colr</td>
<td>color table for data layers</td>
</tr>
<tr>
<td>colr2</td>
<td>secondary color tables for data layers</td>
</tr>
<tr>
<td>cell_misc</td>
<td>miscellaneous cell support files</td>
</tr>
<tr>
<td>hist</td>
<td>history information for data layers</td>
</tr>
<tr>
<td>dig</td>
<td>binary vector data</td>
</tr>
<tr>
<td>dig_ascii</td>
<td>ascii vector data</td>
</tr>
<tr>
<td>dig_att</td>
<td>vector attribute support</td>
</tr>
<tr>
<td>dig_plus</td>
<td>vector topology support</td>
</tr>
<tr>
<td>dig_cats</td>
<td>vector category label support</td>
</tr>
<tr>
<td>reg</td>
<td>digitizer point registration</td>
</tr>
<tr>
<td>bdlg</td>
<td>binary bdlg files</td>
</tr>
<tr>
<td>dig</td>
<td>ascii dig files</td>
</tr>
<tr>
<td>icons</td>
<td>icon files used by paint</td>
</tr>
<tr>
<td>paint</td>
<td>label and comment files used by paint</td>
</tr>
<tr>
<td>group</td>
<td>imagery group support data</td>
</tr>
<tr>
<td>site_lists</td>
<td>site lists for sites program</td>
</tr>
<tr>
<td>windows</td>
<td>pre-defined windows</td>
</tr>
<tr>
<td>COMBINE</td>
<td>combine scripts</td>
</tr>
<tr>
<td>WEIGHT</td>
<td>weight scripts</td>
</tr>
</tbody>
</table>

Note. The mapset database elements can be simple directory names (e.g., cats, colr) or multi-level 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 map layers 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_mynam(66).

DEFAULT_WIND contains the default window for the location. This file is used to initialize the WIND file for new mapsets. The contents of this file are returned by G_get_default_window(78).
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" cell file, the system looks in the various database mapsets for a cell 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" cell 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 mapsets or Gmapsets commands.

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 in PERMANENT" be displayed.

4.7.2. UNIX File Permissions

GRASS 3.0 creates all files with read/write permission enabled for everyone and directories with read/write/search permission enabled for everyone. This implies that all users can read anyone else's data files.

---

4 This means -rw-rw-rw for files, and drwxrwxrwx for directories. It is accomplished by setting the umask to 0 in all GRASS programs.

5 It also implies that all users can modify and remove anyone else's files. Although GRASS code won't create or modify files in other users' mapsets, the database is wide open to standard UNIX access. A planned improvement will be to set the umask to 022 so that the permissions are -rw-r--r- for files and drwxr-xr-x for directories. This will allow complete control of access to the database.
While there is no mechanism currently in GRASS to modify these access permissions, access to a mapset can be controlled by removing (or adding) the read and search permissions on the mapset directory itself using the UNIX `chmod` command, without adversely affecting GRASS programs. For example, suppose that the full UNIX name of the mapset is `/grass/data/spearfish/xyz`. To set the permissions so that only the mapset owner can access the `xyz` mapset:

```
chmod 0700 /grass/data/spearfish/xyz
```

To reset the permissions so that everyone can read from the mapset:

```
chmod 0755 /grass/data/spearfish/xyz
```

**Warning.** Since the PERMANENT mapset contains global database information, all users must have read and search access to the PERMANENT mapset directory.\(^6\) Don't 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.
Chapter 5

Grid Cell Maps

This chapter provides an explanation of how grid cell map layers are accommodated in the GRASS database.\(^1\)

5.1. What is a Grid Cell Map Layer?

GRASS grid cell 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 grid cell data are stored as a matrix of grid cells. Each grid cell covers a known, rectangular (generally square) patch of land. Each 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.

<table>
<thead>
<tr>
<th>land cover</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 3 3 3 4 4</td>
</tr>
<tr>
<td>2 3 3 3 4 4</td>
</tr>
<tr>
<td>2 3 3 3 4 4</td>
</tr>
<tr>
<td>1 2 3 3 3 4</td>
</tr>
<tr>
<td>1 1 1 3 3 4</td>
</tr>
<tr>
<td>1 1 3 3 4 4</td>
</tr>
</tbody>
</table>

1 = urban 2 = prairie 3 = forest 4 = wetlands

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

\(^1\) The descriptions given here are for GRASS 3.0 data formats only. Previous formats, still supported by GRASS but no longer generated, are described in documents from earlier releases of GRASS.
database directories are:

<table>
<thead>
<tr>
<th>directory</th>
<th>function</th>
</tr>
</thead>
<tbody>
<tr>
<td>cell</td>
<td>grid cell data files</td>
</tr>
<tr>
<td>cellhd</td>
<td>grid cell header files</td>
</tr>
<tr>
<td>cats</td>
<td>map layer category information</td>
</tr>
<tr>
<td>colr</td>
<td>map layer color tables</td>
</tr>
<tr>
<td>colr2</td>
<td>alternate map layer color tables</td>
</tr>
<tr>
<td>hist</td>
<td>map layer history information</td>
</tr>
<tr>
<td>cell_misc</td>
<td>miscellaneous map layer support information</td>
</tr>
</tbody>
</table>

For example, a map layer 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 cell files are described in §12.8 Cell File Processing [p. 80].

5.2. Grid Cell File Format

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

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

Uncompressed format. The uncompressed cell file actually looks like an NxM matrix. Each byte (or set of bytes for multi-byte data) represents a cell of the 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 cell 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 Multi-byte 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 cell file itself contains no useful information. The reclass
information is stored in the cell header file.

Multi-byte data format. When the data values in the cell 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:


<table>
<thead>
<tr>
<th>cell value</th>
<th>base 256</th>
<th>stored as</th>
</tr>
</thead>
<tbody>
<tr>
<td>868</td>
<td>(3*256 + 100)</td>
<td>3 100</td>
</tr>
<tr>
<td>137,304</td>
<td>(2<em>256^2 + 24</em>256 + 88)</td>
<td>2 24 88</td>
</tr>
<tr>
<td>174,058,106</td>
<td>(10<em>256^3 + 95</em>256^2 + 234*256 + 122)</td>
<td>10 95 234 122</td>
</tr>
</tbody>
</table>

Negative values are stored as a signed quantity, i.e., with the highest bit set to 1:\(^3\)


<table>
<thead>
<tr>
<th>cell value</th>
<th>base 256</th>
<th>stored as</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>(-(1))</td>
<td>1 0 0 0 1</td>
</tr>
<tr>
<td>-868</td>
<td>(-(3*256 + 100))</td>
<td>1 0 0 3 100</td>
</tr>
<tr>
<td>-137,304</td>
<td>(-(2<em>256^2 + 24</em>256 + 88))</td>
<td>1 0 2 24 88</td>
</tr>
<tr>
<td>-174,058,106</td>
<td>(-(10<em>256^3 + 95</em>256^2 + 234*256 + 122))</td>
<td>1 10 95 234 122</td>
</tr>
</tbody>
</table>

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 [3100].

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 multi-byte format described above is, except for negative values,
machine-independent. If cell 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 3.0 format cell files will transfer between machines, with two restrictions: (1) if the file contains negative values, the size of an integer on the two machines must be the same; and (2) the size of the file must be within the seek capability of the lseek() call. The pre-3.0 compressed format is not stored in a machine-independent format, and cannot generally be used for inter-machine transfer. It will transfer if the two machines have the same integer and long integer format.

5.3. Cell Header Format
The cell 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 cell header file. The format of the cell 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 cell header file are described in §12.9.1 Cell Header File [p.89].

5.3.1. Regular Format
The regular map layer cell header contains the information describing the physical characteristics of the cell file. The cell header has the following fields:

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>proj:</td>
<td>1</td>
</tr>
<tr>
<td>zone:</td>
<td>18</td>
</tr>
<tr>
<td>north:</td>
<td>4660000.00</td>
</tr>
<tr>
<td>south:</td>
<td>4570000.00</td>
</tr>
<tr>
<td>east:</td>
<td>770000.00</td>
</tr>
<tr>
<td>west:</td>
<td>710000.00</td>
</tr>
<tr>
<td>e-w resol:</td>
<td>50.00</td>
</tr>
<tr>
<td>n-s resol:</td>
<td>100.00</td>
</tr>
<tr>
<td>format:</td>
<td>0</td>
</tr>
<tr>
<td>compressed:</td>
<td>0</td>
</tr>
</tbody>
</table>

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

0 is unreferenced x,y (imagery data)
1 is UTM
2 is State Plane

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

4 This usually means that the size of a long integer on the two machines is the same.
5 State Plane is not yet fully supported in GRASS.

§5 Grid Cell Maps
example above, the projection is UTM, the zone 18.

north, south, east, west

The geographic boundaries of the cell 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 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.

format

The format field describes how many bytes per cell are required to represent the grid cell data. 0 means 1 byte, 1 means 2 bytes, etc.

compressed

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

rows, cols

The rows and columns of the grid matrix are not stored in the cell header. They are computed from the geographic boundaries as follows:

\[
\begin{align*}
\text{rows} &= (\text{north} - \text{south}) / (\text{ns resol}) \\
\text{cols} &= (\text{east} - \text{west}) / (\text{ew resol})
\end{align*}
\]

5.3.2. Reclass Format

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

<table>
<thead>
<tr>
<th>reclass cell header</th>
</tr>
</thead>
<tbody>
<tr>
<td>reclass</td>
</tr>
<tr>
<td>name: county</td>
</tr>
<tr>
<td>mapext: PERMANENT</td>
</tr>
<tr>
<td>#5</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>2</td>
</tr>
</tbody>
</table>

first category in reclass

5 is reclassified to 1
6 is reclassified to 0
7 is reclassified to 1
8 is reclassified to 0
9 is reclassified to 2

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

The # as the first character of the fourth line in the file indicates that this is a 3.0 format reclass cell header file.

5.4. Cell 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 # as the first character of the first line in the file indicates that this is a 3.0 format category file. The number which follows it is the largest category value in the cell 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 cell category file are described in §12.9.2 Cell Category File [p.91].

5.5. Cell Color Table Format
The color table contains one line of a color description for each category of data, including the "no data" category. The colors are represented as levels of red, green, and blue, where 0 represents the lowest intensity and 255 represents the highest intensity.
The # as the first character of the first line in the file indicates that this is a 3.0 format color file. The number which follows is the first data value which has a color (and should be the lowest non-zero category value in the cell file). The next line is the color for category 0. The remaining lines are the colors for the other categories. There are 3 columns representing the red, green, and blue levels respectively. If all 3 values are identical (i.e., a grey scale color), only the red value need be present.

Note that the color file format is a modest attempt to allow color tables for files like elevation, which have their lowest non-zero data value above 1 (often above 1000). In these cases the color table doesn't have to start with 1 and create unused colors.

Note. GIS Library routines which read and write the cell color table are described in §12.9.3 Cell Color Table (p.94).

5.6. Cell History File
The history file contains historical information about the cell file: creator, date of creation, comments, etc. In most applications, the programmer need not be concerned with the history file. It is generated automatically along with the cell file. The GRASS layer.info program allows the user to view this information, and the support program allows the user to update it.

Note. GIS Library routines which read and write the cell history file are described in §12.9.4 Cell History File (p.98).

5.7. Cell Range File
The range file contains the minimum and maximum values which occur in a cell file. It is generated automatically for all new cell files. This file lives in the cell_misc element as "cell_misc/name/range" where name is the related cell 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 cell 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.

§5 Grid Cell Maps
Note. *GIS Library* routines which read and write the cell range file are described in §12.9.5 *Cell Range File* (p.99).
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 non-intersecting curves called arcs. An arc is stored as a series of x,y coordinate pairs. The two end-points of an arc are called nodes. Two consecutive x,y pairs define an arc segment.

The arcs, either singly, or in combination with others, form higher level map features: lines (e.g., roads or streams) or areas (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.

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 Grid Cell Map Layer? [p.23] for more information about GRASS category values.

---

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

<table>
<thead>
<tr>
<th>directory</th>
<th>function</th>
</tr>
</thead>
<tbody>
<tr>
<td>dig</td>
<td>binary arc file</td>
</tr>
<tr>
<td>dig_ascii</td>
<td>ascii arc file</td>
</tr>
<tr>
<td>dig_att</td>
<td>vector category attribute file</td>
</tr>
<tr>
<td>dig_cats</td>
<td>vector category labels</td>
</tr>
<tr>
<td>dig_plus</td>
<td>vector index/pointer file</td>
</tr>
<tr>
<td>reg</td>
<td>digitizer registration points</td>
</tr>
</tbody>
</table>

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 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 Dig Library [p. 123].

### 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 import.to.vect, a.b.vect, and b.a.vect 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:

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

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:

---

* 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 window which should encompass all the data in the vector file.

‡ The MAP THRESH is set by the *digit* program. If the data comes from outside GRASS, this field can be set to 0.0.

§6 Vector Maps
6.2.2. Arc Section

The arc information appears in the second section of the dig_ascii file (following VERIT, 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 line8), 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

8 Other types may be added in the future.
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 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_asci file.

6.3. Vector Category Attribute File

As was mentioned in §6.1 What is a Vector Map Layer? (p.31), 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 the last one).

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., vect.to.cell), unlabeled areas will convert as category zero, and unlabeled lines will be ignored.

9 This can be written with the Fortran format: A1,IX,I4.
10 These can be written with the Fortran format: 2(IX,F12.2).

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

<table>
<thead>
<tr>
<th>columns</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Type of map feature (A or L)*</td>
</tr>
<tr>
<td>2-3</td>
<td>spaces</td>
</tr>
<tr>
<td>4-15</td>
<td>Easting (x) of the marker, right justified</td>
</tr>
<tr>
<td>16-17</td>
<td>spaces</td>
</tr>
<tr>
<td>18-29</td>
<td>Northing (y) of the marker, right justified</td>
</tr>
<tr>
<td>30-31</td>
<td>spaces</td>
</tr>
<tr>
<td>32-39</td>
<td>Category number, right justified</td>
</tr>
<tr>
<td>40-49</td>
<td>spaces</td>
</tr>
<tr>
<td>50</td>
<td>newline †</td>
</tr>
</tbody>
</table>

This format is required by programs which modify the vector map (e.g., `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 `support.vect` 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 grid cell category file described in §5.4 Cell Category File Format [p.28], and the reader is referred to that section for details.

Note. The program `support.vect` allows the user to enter and modify the vector category labels. The program `vect.to.cell` copies the `dig_cats` file to the cell category file during the vector to raster conversion.

Note. Library routines which read and write the `dig_cats` file are described under §12.10.6 Vector Category File [p.108].
6.5. Vector Index and Pointer File
The dig plus file contains information that accelerates vector queries. It is created by
the program buildvect (which is run by digit when a vector file is created or modified,
and by support vect 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 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:

<table>
<thead>
<tr>
<th>Easting</th>
<th>Northing</th>
</tr>
</thead>
<tbody>
<tr>
<td>383000.0000</td>
<td>4429000.0000</td>
</tr>
<tr>
<td>383000.0000</td>
<td>4456000.0000</td>
</tr>
<tr>
<td>404000.0000</td>
<td>4456000.0000</td>
</tr>
<tr>
<td>404000.0000</td>
<td>4429000.0000</td>
</tr>
</tbody>
</table>

Note. This file is used by 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:

\texttt{dig\_ascii}

The \texttt{dig\_ascii} file, described in §6.2 *Ascii Arc File Format* [p.32], is required.

\texttt{dig\_att}

The \texttt{dig\_att} file, described in §6.3 *Vector Category Attribute File* [p.35], is essentially required. While the \texttt{dig\_ascii} file alone is sufficient for simple vector display, the \texttt{dig\_att} file is required for vector to cell conversion, as well as more sophisticated vector query.

\texttt{dig\_cats}

The \texttt{dig\_cats} file, described in §6.4 *Vector Category Label File* [p.36], while not required, allows map feature descriptions to be imported as well.

\textbf{Note.} The \texttt{dig\_plus} file, described in §6.5 *Vector Index and Pointer File* [p.37], is created by the GRASS program \texttt{import.to.vect} when converting the \texttt{dig\_ascii} file to the binary \texttt{dig} file.

\section*{§6 Vector Maps}
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 sites\(^1\) 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 sites program. The word name must be all lower case letters.

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

---


\(^2\) Other GRASS programs which read site lists include Gsites, dsites and point.

\(^3\) See §4.5.2 Elements [p. 10] 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 sites 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:

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).

Note. The character | is used to separate the fields in the records.

7.3. Programming Interface to Site Files
The programming interface to the site list files is described in §12.11 Site List Processing (p. 105) and the programmer should refer to that section for details.
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 infra-red 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:

\begin{itemize}
  \item A list of related cell files.
  \item A place to store control points for image registration and rectification, and
\end{itemize}

\(^1\) See §8.4 Imagery Programs (p. 45) 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. 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 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 GraspCalc, etc.

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

§8 Image Data: Groups
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 within 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. 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/
       |
       REFS POINTS TARGET subgroup/
```

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

<table>
<thead>
<tr>
<th>#</th>
<th>image</th>
<th>target</th>
<th>status</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>east</td>
<td>north</td>
<td>east</td>
</tr>
<tr>
<td></td>
<td>east</td>
<td>north</td>
<td></td>
</tr>
<tr>
<td>#</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>504.00</td>
<td>-2705.00</td>
<td>379145.30</td>
</tr>
<tr>
<td>1</td>
<td>458.00</td>
<td>-2713.00</td>
<td>378272.67</td>
</tr>
<tr>
<td>1</td>
<td>2285.80</td>
<td>-2206.00</td>
<td>415610.08</td>
</tr>
<tr>
<td>0</td>
<td>2397.00</td>
<td>-2564.00</td>
<td>417043.22</td>
</tr>
<tr>
<td>1</td>
<td>2158.00</td>
<td>-2944.00</td>
<td>411067.79</td>
</tr>
<tr>
<td>0</td>
<td>2148.00</td>
<td>-2913.00</td>
<td>410834.61</td>
</tr>
<tr>
<td>1</td>
<td>2288.80</td>
<td>-2336.20</td>
<td>415497.19</td>
</tr>
</tbody>
</table>

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*) 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. 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*.

§8 Image Data: Groups
In this example, the subgroups are named 123, 234, 1357, etc. 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 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.
image extraction
  i.tape.mss  Landsat Multi-Spectral 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. 137] and the reader is referred to that chapter for details.
Chapter 9

Window 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 window and the mask. The user is allowed to set a window which defines a rectangular area of coverage on the earth, and optionally further limit the coverage by specifying a "cookie-cutter" mask. The window and mask are stored in the database under the user's current mapset. GRASS programs automatically retrieve only data that fall within the window. Furthermore, if there is a mask, only data that fall within the mask are retained. Programs determine the window and mask from the database rather than asking the user.

9.1. Window

The user's current database window\(^1\) is set by the user using the GRASS window, Gwindow, or d.window commands. It is stored in the WIND file in the mapset. This file not only specifies the geographic boundaries of the window rectangle, but also the window resolution which implicitly grids the window into rectangular "cells" of equal size.

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

\(^1\) The choice of the term "window" is unfortunate. It is used in other contexts as well (e.g., graphics windows) leading to much user confusion. A better term would have been "coverage." When confusion arises, refer to this window as the "database window" or the "mapset window," and to windows on the graphics screen as "graphics windows."
The WIND file contains the following fields:

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>north</td>
<td>4660000.00</td>
</tr>
<tr>
<td>south</td>
<td>4570000.00</td>
</tr>
<tr>
<td>east</td>
<td>770000.00</td>
</tr>
<tr>
<td>west</td>
<td>710000.00</td>
</tr>
<tr>
<td>e-w resol</td>
<td>50.00</td>
</tr>
<tr>
<td>n-s resol</td>
<td>100.00</td>
</tr>
<tr>
<td>proj</td>
<td>1</td>
</tr>
<tr>
<td>zone</td>
<td>18</td>
</tr>
</tbody>
</table>

north, south, east, west

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

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 window 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, window grid cells need not be square.

proj, zone

The projection field specifies the type of cartographic projection: 0 is unreferenced x,y (imagery data), 1 is UTM, 2 is State Plane. 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 WIND file format is very similar to the format for the cell header files. See §5.3 Cell Header Format [p.26] for details about cell header files.

9.2. Mask

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

2 State Plane is not yet fully supported in GRASS.

3 The mask program creates MASK as a reclass file because the reclass function is fast and uses less disk space, but it doesn't actually matter that MASK is a reclass file. Any cell format can be used. The only thing that really matters is that the cell file be called MASK.
The following diagram gives a visual idea of how the mask works:

<table>
<thead>
<tr>
<th>input</th>
<th>MASK</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 4 4</td>
<td>0 1 1</td>
<td>0 4 4</td>
</tr>
<tr>
<td>3 3 4</td>
<td>1 1 0</td>
<td>3 3 0</td>
</tr>
<tr>
<td>2 3 3</td>
<td>1 0 0</td>
<td>2 0 0</td>
</tr>
</tbody>
</table>

9.3. Variations

If a GRASS program does not obey either the window or the mask, the variation must be noted in the user documentation for the program, and the reason for the variation given. For example, the slope.aspect program which generates aspect and slope maps from elevation data uses the resolution of the elevation data itself, and not the current window resolution (which may differ). The program documentation for slope.aspect warns the user about this: The current window and mask settings are ignored. The elevation file is read directly to insure that data is not lost or inappropriately resampled.
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 commands GRASS3 and grass3 set the following UNIX environment variables:

1. GISBASE: top level directory for the GRASS programs
2. GIS_LOCK: process id of the start-up shell script
3. 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, the menu files would be found in /grass/menu, 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.66).

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. Under GRASS 3.0 this file is `.grassrc`\(^2\) in the user's home directory.

### 10.2. GRASS Environment

All GRASS users will have a file in their home directory named `.grassrc`\(^3\) 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:

<table>
<thead>
<tr>
<th>Variables</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GISDBASE</td>
<td>top level database directory</td>
</tr>
<tr>
<td>LOCATION_NAME</td>
<td>location directory</td>
</tr>
<tr>
<td>MAPSET</td>
<td>mapset directory</td>
</tr>
</tbody>
</table>

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. 67)`, `G_location(p. 66)`, and `G_mapset(p. 66)` 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:

<table>
<thead>
<tr>
<th>Variables</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MONITOR</td>
<td>currently selected graphics monitor</td>
</tr>
<tr>
<td>PAINTER</td>
<td>currently selected paint output device</td>
</tr>
<tr>
<td>DIGITIZER</td>
<td>currently selected digitizer</td>
</tr>
</tbody>
</table>

These variables are accessed and set from C programs using the general purpose routines `G.getenv(p. 67)` and `G.setenv(p. 67)`. The GRASS program `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

---

\(^2\) Under previous versions of GRASS this file was named `.gism`

\(^3\) 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 `.csrc` and `.execc` files.
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 GRASS application programs. Second, since the variables remain in the file unless explicitly removed, they are available from session to session.
Chapter 11

Compiling GRASS Programs Using Gmake

GRASS programs are compiled using the Gmake front-end to the UNIX make command. Gmake reads a file named Gmakefile to construct a makefile and then runs make. (It is assumed that the programmer is familiar with make and its accompanying makefiles.)

11.1. Gmake

The GRASS Gmake utility allows make compilation rules to be developed without having to specify machine- and installation-dependent information. Gmake combines pre-defined variables that specify the machine- and installation-dependent information with the file Gmakefile, which the programmer must write, to create a makefile. (The pre-defined variables and the construction of a Gmakefile are described below.)

Gmake is invoked as follows:

Gmake [source directory] [target]

If run without arguments, Gmake 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, Gmake will change into this directory and then proceed as above. If run with a target argument as well, then make will be run on the specified target.

11.2. Gmake Variables

The pre-defined Gmake 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.58). The full set of variables can be seen in Appendix A. Annotated Gmake Pre-defined Variables (p.233).

1 Gmake lives under $GISBASE/arc/CMD. You must either set your $PATH to include this directory, or run $GISBASE/arc/CMD/Gmake. $GISBASE is the directory where GRASS is installed. See §10.1 UNIX Environment (p.51) for details.
Variables marked with (-) are not commonly used.

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

GIS (-) This is the UNIX directory where GRASS is installed. It corresponds to the GRASS environment variable GISBASE (see §10 Environment Variables [p. 51]). This variable is generally not used explicitly in a *Gmakefile*. It is mostly used by *Gmake* to construct other variables.

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 and include header files live. For example, "gis.h" can be found here. *Gmake* 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. 63] for details about this library, and §12.18 Loading the GIS Library [p. 122] 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. 187] for details about this library, and §20.4 Loading the Vask Library [p. 191] for a sample *Gmakefile* which loads this library.

SEGMENTLIB This names the Segment Library, which manages large matrix data. See §19 Segment Library [p. 179] for details about this library, and §20.4 Loading the Vask Library [p. 191] 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. 147] for
details about this library, and §15.9 Loading the Raster Graphics Library [p. 157] 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. 159] for details about this library, and §16.9 Loading the Display Graphics Library [p. 167] 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. Don’t use $(CURSES) if you use $(VASK).

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

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

AR This variable specifies the rule that must be used to build object libraries.

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

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

GMAKE This is the full name of the Gmake 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 Gmake 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 won’t 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 doesn’t 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 doesn’t 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

§11 Compiling GRASS Programs Using Gmake
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.$^2$

$(BIN)$ is a *Gmake* variable which names the UNIX directory where GRASS commands live. Specifying the target as $(BIN)/xyz$ will cause *Gmake* 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 almost never have to 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): myheader.h
$(GISLIB): # in case library changes
```

$^2$ GRASS *Gmakefiles* presently use cc instead of $(CC)$. This will be modified in future releases.
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 = -L
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 -L. 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. Gmake requires that these libraries be built using the $(AR) rule as follows:

```
OBJ = sub1.o sub2.o sub3.o
lib.a: $(OBJ)
    $(AR)
```

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

§11 Compiling GRASS Programs Using Gmake
11.3.4. Building more than one target

Many target:dependency lines may be given. However, it is the first one in the 
Gmakefile which is built by Gmake. If there are more targets to be built, the first 
target must explicitly or implicitly cause Gmake 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) $(G8LIB)
  $(CC) $(LDFLAGS) -o $@ $(ABC) $(GISLIB)

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

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

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

```
all:
  $(GMAKE) subdir.1
  $(GMAKE) subdir.2
  $(GMAKE) subdir.3
```

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.

11.3.5. Don't bypass .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 don't 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 Compiling GRASS Programs Using Gmake
Chapter 12

GIS Library

12.1. Introduction

The GIS Library is the primary programming library provided with the GRASS system. Programs must use this library 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 cell 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 Grid Cell Maps [p. 23] for details about map layers in GRASS, and §9 Window and Mask [p. 47] which discusses windowing and masking.

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 inter-relationships 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. 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 §24.5 Appendix C. Index to GIS Library [p. 239].

---

1 The GRASS compilation process, described in §11 Compiling GRASS Programs Using Grumpy [p. 53], 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)

```
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. 118)`. 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)

**G_warning** (message)

```
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` 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

---

2 $GISBASE is the directory where GRASS is installed. See §10.1 UNIX Environment (p. 571) for details.
want the messages displayed graphically instead of on the standard error output. If the programmer wants to handle the error messages differently, the following routines can be used to modify the error handling:

**G_set_error Routine (handler)**

```c
int (*handler)();
```

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

```c
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 ()**

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

**G_sleep_on_error (flag)**

```c
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)**

```c
int flag;
```

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

**Note.** This routine has no effect on `G_fatal_error()`. 

812 GIS Library
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. 51) 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, "%/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.

```c

char *
G_gisdbase ()

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 ()

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.

```c

char *
G_getenv (name)

query GRASS environment variable

char *
G__getenv (name)

query GRASS environment variable

char *name;

Thee 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__setev (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 GIS Library
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, there are routines to process cell files which should be used instead (see §12.8 Cell File Processing (p.80)).

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, cell 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. 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:

```c
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

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

```c
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`

```c
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.

---

4 This line of the prompt can be modified using `G_set.ask_return_msg` [p. 70].
The user is asked to enter the name of an file which exists in the current mapset.

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

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 issued to the user if the file exists.

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

The current msg (as set by G_set_ask_return_msg(p. 70)) is returned.

12.5.2. Finding Files in the Database

Non-interactive 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. GRASS allows the user to specify database file names either as a simple unqualified name, such as "xyz", or as a fully qualified name, such as "xyz in mapset", where mapset is the mapset where the file is
to be found. Often only the unqualified file name is provided on the command line.

The following routines search the database for files:

```c
char *
G_find_file (element, name, mapset)  // find a database file

char *
G_find_file2 (element, name, mapset)  // find a database file
```

Look for the file `name` under the specified `element` in the database. The `mapset` parameter can either be the empty string "", which means search all the mapsets in the user's current mapset search path,\(^5\) or it can be a specific mapset, which means look for the file only in this one mapset (for example, in the current mapset).

If found, the mapset where the file lives is returned. If not found, the NULL pointer is returned.

The difference between these two routines is that if the user specifies a fully qualified file which exists, then `G_find_file2()` modifies `name` by removing the "in mapset" while `G_find_file()` does not.\(^6\) 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_file2()` should be used.

For example, to find a "paint/labels" file anywhere in the database:

```c
char name[50];
char *mapset;

if (m = G_find_file("paint/labels",name,"")) == NULL)
    /* not found */
```

To check that the file exists in the current mapset:

\(^5\) See §4.7.1 Mapset Search Path \(p \) 30 for more details about the search path.

\(^6\) Be warned that `G_find_file2()` should not be used directly on a command line argument, since modifying `argv[1]` may not be valid. The argument should be copied to another character buffer which is then passed to `G_find_file2()`.

§12 GIS Library
12.5.3. 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.69)) 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:

```c
G_legal_filename (name)
```

check for legal database file names

```c
char *name;
```

Returns 1 if name is ok, -1 if it isn't.

12.5.4. 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.69), and non-interactively using G_find_file(p.71) or G_find_file2(p.71).

```c
G_open_old (element, name, mapset)
```

open a database file for reading

```c
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 doesn't 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 doesn't exist, the NULL pointer is returned. Otherwise the file descriptor from the fopen() is returned.

12.5.5. 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.70), and non-interactively using G_find_file(p.71) or G_find_file2(p.71).

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 doesn't 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 doesn't 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.
12.5.6. Creating and Opening a New Database File

The following routines create the new file name in the current mapset\(^7\) under the specified database element and open it for writing. The database element is created, if it doesn’t already exist.

The file name should be obtained interactively using \texttt{G\_ask\_new(p.69)}. If obtained non-interactively (e.g., from the command line), \texttt{G\_legal\_filename(p.72)} 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 \texttt{G\_ask\_new(p.69)} guarantees that name will not exist, but if name is obtained from the command line, name may exist. In this case \texttt{G\_find\_file(p.71)} could be used to see if name exists.

\begin{verbatim}
G\_open\_new(element, name)
    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 doesn't 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)
    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 doesn't 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.
\end{verbatim}

\(^7\) GRASS doesn’t allow files to be created outside the current mapset; see §4.7 Database Access Rules (p. 20).
12.5.7. Database File Management

The following routines allow the renaming and removal of database files in the current mapset.\(^8\)

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

```c
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** doesn't exist, and -1 if there was an error.

**Bug.** This routine doesn't check to see if the **new** name is a valid database file name.

**G_remove** (element, name)

```c
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** doesn't 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 cell file will be remove (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.

\(^8\) 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 on page 30.
char *
G_malloc (size)
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)
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)
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 cell files is discussed in §12.8.5
Allocating Cell I/O Buffers [p.85]

12.7. The Window

The window concept is explained in §9.1 Window [p.47]. It can be thought of as a
two-dimensional matrix with known boundaries and rectangular cells.

There are logically two different windows. The first is the database window that the
user has set in the current mapset. The other is the window that is active in the
program. This active program window is what controls reading and writing of cell file
data.

The routines described below use a GRASS data structure Cell_head to hold window
information. This structure is defined in the "gis.h" header file. It is discussed in
detail under §12.17 GIS Library Data Structures [p.118].
12.7.1. The Database Window

Reading and writing the user's database window are done by the following routines:

**G_get_window** (window)  
struct Cell_head *window;  

Reads the database window as stored in the WIND file in the user's current mapset into window.

An error message is printed and exit() is called if there is a problem reading the window.

*Note.* GRASS applications that read or write cell files should not use this routine, since its use implies that the active program window will not be used. Programs that read or write cell file data (or vector data) can query the active program window using *G_window_rous* (p. 78) and *G_window_cols* (p. 78).

**G_put_window** (window)  
struct Cell_head *window;  

Writes the database window file (WIND) in the user's current mapset from window.

Returns 1 if the window is written ok. Returns -1 if not (no diagnostic message is printed).

*Warning.* Since this routine actually changes the database window, it should only be called by programs which the user knows will change the window. It is probably fair to say that under GRASS 3.0 only the *window*, *Gwindow*, and *d.window* programs should call this routine.

There is another database window. This window is the default window for the location. The default window provides the user with a "starting" window, i.e., a window to begin with and return to as a reference point. The GRASS programs *window* and *Gwindow* allow the user to set their database window from the default window. (See §4.6 Permanent Mapset [p. 19] for a discussion of the default window.) The following routine reads this window:
**G_get_default_window (window)**

```c
struct Cell_head *window;
```

Reads the default window for the location into `window`.

An error message is printed and `exit()` is called if there is a problem reading the default window.

---

### 12.7.2. The Active Program Window

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

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

- **G_window_rows ()**
  
  ```c
  number of rows in active window
  ```

- **G_window_cols ()**
  
  ```c
  number of columns in active window
  ```

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

```c
int rows, cols;
int row, col;

rows = G_window_rows();
ncols = G_window_cols();
for (row = 0; row < rows; row++)
    { read row ...
        for (col = 0; col < cols; col++)
            { process col ...
        }
    }
```
**G_set_window (window)**

struct Cell_head *window;

This routine sets the active window from window. Setting the active window does not change the WIND file in the database. It simply changes the window for the duration of the program.\(^9\)

A warning message is printed and -1 returned if window is not valid. Otherwise 1 is returned.

**Note.** This routine overrides the window 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 window will be the same as what is in the user's WIND file.

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

**G_get_set_window (window)**

struct Cell_head *window;

Gets the values of the currently active window into window. If G_set_window(p. 79) has been called, then the values set by that call are retrieved. Otherwise the user's database window is retrieved.

**Note.** For programs that read or write cell data, and really need the full window information, this routine is preferred over G_get_window(p. 77). However, since G_window_rows(p. 78) and G_window_cols(p. 78) return the number of rows and columns in the active window, the programmer should consider whether or not the full window information is really needed before using this routine.

The following routines return information about the cartographic projection and zone. See §9.1 Window (p. 47) for more information about these values.

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

§12 GIS Library
G_projection ()

query cartographic projection

This routine returns a code indicating the projection for the active window. The current values are:

0 unreferenced x,y (imagery data),
1 UTM,
2 State Plane.11

Others may be added in the future.

char *
G_projection_name (proj)

int proj;

Returns a pointer to a string which is a printable name for projection code proj (as returned by G_projection(p.80)). Returns NULL if proj is not a valid projection.

G_zone ()

query cartographic zone

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

12.8. Cell File Processing

Cell files are the heart and soul of GRASS. All analyses are performed with cell file data. Because of this, a suite of routines which process cell file data has been provided.

The processing of cell files consists of determining which cell file or files are to be processed (either by prompting the user or as specified on the program command line), locating the cell file in the database, opening the cell file, dynamically allocating i/o buffers, reading or writing the cell file, closing the cell file, and creating support files for newly created cell files.

All cell file data is of type CELL12, which is defined in "gis.h".

---

11 State Plane is not yet fully supported in GRASS.
12 See Appendix B. The CELL Data Type [p 237] for a discussion of the CELL type and how to use it (and avoid misusing it).
12.8.1. Prompting for Cell Files

The following routines interactively prompt the user for a cell 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 cell 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. These routines have a built-in 'list' capability which allows the user to get a list of existing cell files.

The user is required to enter a valid cell 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 cell file lives or is to be created is returned. Both the name and the mapset are used in other routines to refer to the cell file.

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

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

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

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

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

Asks the user to enter a name for a cell 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:

11 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 *mapset;
char name[50];

mapset = G_ask_cell_old("Enter cell file to be processed", name);
if (mapset == NULL)
    exit(0);

12.8.2. Finding Cell Files in the Database

Non-interactive programs cannot make use of the interactive prompting routines described above. For example, a command line driven program may require a cell file name as one of the command arguments. GRASS allows the user to specify cell 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 in mapset", where mapset is the mapset where the cell file is to be found. Often only the unqualified cell file name is provided on the command line.

The following routines search the database for cell files:

char *
G_find_cell (name, mapset) find a cell file
char *
G_find_cell2 (name, mapset) find a cell file

    char *name;
    char *mapset;

Look for the cell 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,\textsuperscript{14} or it can be a specific mapset name, which means look for the cell file only in this one mapset (for example, in the current mapset).

If found, the mapset where the cell 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 cell file which exists, then G_find_cell2() modifies name by removing the "in mapset" while G_find_cell() does not.\textsuperscript{15} 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

\textsuperscript{14} See §4.7.1 Mapset Search Path \cite{20} for more details about the search path.

\textsuperscript{15} Be warned that G_find_cell2() 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_cell2().
returned unqualified (for displaying the name to the user, or storing it in a data file, etc.), then G_find_file() should be used.

For example, to find a cell file anywhere in the database:

```c
char name[50];
char *mapset;

if ((mapset = G_find_file(name,"")) == NULL)
    /* not found */
```

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

```c
char name[50];

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

12.8.3. Opening an Existing Cell File

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

The cell file name and mapset can be obtained interactively using G_ask_cell_old(p.81) or G_ask_cell_in_mapset(p.81), and non-interactively using G_find_cell(p.82) or G_find_file2(p.82).

```c
G_open_cell_old (name, mapset)
```

This routine opens the cell file name in mapset for reading.

A non-negative 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 cell files will be resampled into the current window, the resampling index for the cell file is prepared by this routine after the file is opened. The resampling is based on the active program window. Preparation required for reading the various cell file formats is also done.

---

16 See also §12.7 The Window (p.76).
17 See §5.2 Grid Cell File Format (p.34) for an explanation of the various cell file formats.
12.8.4. Creating and Opening New Cell Files

The following routines create the new cell file name in the current mapset and open it for writing. The cell file name should be obtained interactively using G_ask_cell_new(p.81). If obtained non-interactively (e.g., from the command line), G_legal_filename(p.72) 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 cell files are actually created as temporary files and moved into the cell directory when closed. This allows an existing cell file to be read at the same time that it is being re-written. The interactive routine G_ask_cell_new(p.81) 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.82) 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.108), is named using the current process id. If the new cell file is opened by a parent process which exits after creating a child process using fork(), the cell 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 cell file. (See the discussion under G_tempfile(p.108) for more details.)

G_open_cell_new(name)

open a new cell file (sequential)

char *name;

Creates and opens the cell file name for writing by G_put_map_row(p.88) which writes the file row by row in sequential order. The cell file data will be compressed as it is written.

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

---

18 GRASS doesn't allow files to be created outside the current mapset. See §4.7 Database Access Rules (p.20).

19 See also G_fork(p.116).
**G_open_cell_new_random** (name)  

open a new cell file (random)

```
char *name;
```

Creates and opens the cell file `name` for writing by `G_put_map_row_random(p, 88)` which allows writing the cell file in a random fashion. The file will be created uncompressed.\(^\text{20}\)

A non-negative 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 cell file (uncompressed)

```
char *name;
```

Creates and opens the cell file `name` for writing by `G_put_map_row(p, 88)` which writes the file row by row in sequential order. The cell file will be in uncompressed format when closed.

A non-negative 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.\(^\text{21}\) This routine is provided so the `uncompress` program can create uncompressed cell files.

### 12.8.5. Allocating Cell I/O Buffers

Since there is no pre-defined limit for the number of columns in the window,\(^\text{22}\) buffers which are used for reading and writing cell data must be dynamically allocated.

---

\(^\text{20}\) 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("compress name").
```

\(^\text{21}\) At present, automatic cell file compression will create files which, in most cases, are smaller than if they were uncompressed. In certain cases, the compressed cell file may be larger. This can happen with imagery data, which don’t 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 cell files in all cases.

\(^\text{22}\) See `G_window_cols(p, 78)` to find the number of columns in the window.
CELL *
G_allocate_cell_buf ()

allocate a cell buffer

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

CELL *cell;
    cell = G_allocate_cell_buf();

If larger buffers are required, the routine G_malloc(p. 76) 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 cell buffer

CELL *buf;

This routine assigns each member of the cell buffer array buf to zero. It assumes that buf has been allocated using G_allocate_cell_buf(p. 86).

12.8.6. Reading Cell Files

Cell file 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 window, not the cell file itself. Cell file data is always read resampled into the window. This allows the user to specify the coverage of the database during analyses. It also allows databases to consist of cell files which do not cover exactly the same area, or do not have the same grid cell resolution. When cell files are resampled into the window, they all "look" the same.

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

---

23 The GRASS window is discussed from a user perspective in §9.1 Window (p. 47) and from a programmer perspective in §12.7 The Window (p. 76). The routines which are commonly used to determine the number of rows and columns in the window are G_window_rows(p. 76) and G_window_cols(p. 78).
G_get_map_row (fd, cell, row)  
    int fd;
    CELL *cell;
    int row;

This routine reads the specified row from the cell file open on file descriptor fd (as returned by G_open_cell_old(p.83)) into the cell buffer. The cell buffer must be dynamically allocated large enough to hold one full row of cell data. It can be allocated using G_allocate_cell_buf(p.86).

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

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

This routine reads the specified row from the cell 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 cell file. Otherwise a non-negative 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 Gdescribe, which reads the cell file directly to report all data values in a cell file, or Gslope.aspect, which produces slope and aspect from elevation, ignore both the mask and the window. However, the number of GRASS programs which do this should be minimal. See §9.2 Mask (p.48) for more information about the mask.

12.8.7. Writing Cell Files

The routines described here write cell file data.
G_put_map_row (fd, buf)

`int fd;`  
`CELL *buf;`  

This routine writes one row of cell data from `buf` to the cell file open on file descriptor `fd`. The cell file must have been opened with `G_open_cell_new()`. The cell buffer `buf` must have been allocated large enough for the window, perhaps using `G_allocate_cell_buf()`.  

If there is an error writing the cell 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 cell file `name` is to be created:

```c
int fd, row;  
nrows;  
CELL *buf;  

fd = G_open_cell_new (name);  
if (fd < 0)  
    /* oops - can't open cell file */  

buf = G_allocate_cell_buf();  
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 cell file (random)`

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

This routine allows random writes to the cell file open on file descriptor `fd`. The cell file must have been opened using `G_open_cell_new_random()`. The cell buffer `buf` contains `ncells` columns of data and is to be written into the cell file at the specified `row`, starting at column `col`.  

§12 GIS Library
12.8.8. Closing Cell Files

All cell files are closed by one of the following routines, whether opened for reading or for writing.

**G_close_cell (fd)**  
*close a cell file*

```c
int fd;
```

The cell file opened on file descriptor **fd** is closed. Memory allocated for cell processing is freed. If open for writing, skeletal support files for the new cell file are created as well.

**Note.** If a program wants to explicitly write support files (e.g., a specific color table) for a cell file it creates, it must do so after the cell file is closed. Otherwise the close will overwrite the support files. See §12.9 Map Layer Support Routines [p. 89] for routines which write cell support files.

**G_unopen_cell (fd)**  
*unopen a cell file*

```c
int fd;
```

The cell file opened on file descriptor **fd** is closed. Memory allocated for cell processing is freed. If open for writing, the cell file is not created and the temporary file created when the cell file was opened is removed (see §12.8.4 Creating and Opening New Cell Files [p. 84]).

This routine is useful when errors are detected and it is desired to not create the new cell file. While it is true that the cell 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.

12.9. Map Layer Support Routines

GRASS map layers have a number of support files associated with them. These files are discussed in detail in §5 Grid Cell Maps (p. 23). The support files are the cell 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.

12.9.1. Cell Header File

The cell 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 cell file. The format of this file is described in §5.3 Cell Header Format (p. 26). The routines
described below use the Cell_head structure which is shown in detail in §12.17 GIS Library Data Structures [p. 118].

G_get_cellhd (name, mapset, cellhd) 

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

The cell header for the cell file name in the specified mapset is read into the cellhd structure.

If there is an error reading the cell header file, a diagnostic message is printed and -1 is returned. Otherwise, 0 is returned.

Note. If the cell file is a reclass file, the cell header for the referenced cell file is read instead. See §5.3.2 Reclass Format [p. 27] for information about reclass files, and G_is_reclass(p. 91) for distinguishing reclass files from regular cell files.

Note. It is not necessary to get the cell header for a map layer in order to read the cell file data. The routines which read cell file data automatically retrieve the cell header information and use it for resampling the cell file data into the active window.24 If it is necessary to read the cell file directly without resampling into the active window,25 then the cell header can be used to set the active window using G_set_window(p. 79).

G_put_cellhd (name, cellhd) 

```
char *name;
struct Cell_head *cellhd;
```

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

If there was an error creating the cell 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. 89) to give new cell files correct cell header files, and by the support program to give users a means of creating or modifying cell headers.

24 See §12.7 The Window [p. 76].
25 but see §9 Window and Mask [p. 47] for a discussion of when this should and should not be done.
G_is_reclass (name, mapset, r_name, r_mapset)
    char *name;
    char *mapset;
    char *r_name;
    char *r_mapset;

This function determines if the cell file name in mapset is a reclass file. If it is, then the name and mapset of the referenced cell file are copied into the r_name and r_mapset buffers.

Returns 1 if name is a reclass file, 0 if it isn’t, and -1 if there was a problem reading the cell header for name.

12.9.2. Cell Category File
GRASS map layers have category labels associated with them. The category file is structured so that each category in the cell file can have a one-line description. The format of this file is described in §5.4 Cell Category File Format [p.28].

The routines described below manage the category file. Some of them use the Categories structure which is described in §12.17 GIS Library Data Structures [p.118].

12.9.2.1. Reading and Writing the Cell Category File
The following routines read or write the category file itself:

G_read_cats (name, mapset, cats)
    char *name;
    char *mapset;
    struct Categories *cats;

The category file for cell 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)

    char *name;
    struct Categories *cats;

    Writes the category file for the cell 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)

    char *name;
    char *mapset;

    If only the map layer title is needed, it isn't necessary to read the entire category file into memory. This routine gets the title for cell 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 doesn't have a title, then a pointer to the empty string """" is returned.

char *
G_put_cell_title (name, title)

    char *name;
    char *title;

    If it is only desired to change the title for a map layer, it isn't necessary to read the entire category file into memory, change the title, and rewrite the category file. This routine changes the title for the cell file name in the current mapset directly in the category file. It returns a pointer to the title.

12.9.2.2. Querying and Changing the Categories Structure

The following routines query or modify the information contained in the category structure:
The pointer that is returned points to a hidden static buffer. Successive calls to `G_get_cat()` overwrite this buffer.

```c
char *
G_get_cat (n, cats) future feedback
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 doesn't 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.

```
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.91), G_init_cats(p.93) and G_set_cat(p.94).

12.9.3. Cell Color Table

GRASS map layers have colors associated with them. The color tables are structured so that each category in the cell file has its own color. The format of this file is described in §5.5 Cell Color Table Format (p.28).

The following routines read, create, modify, and write color tables. They use the Colors structure which is described in detail in §12.17 GIS Library Data Structures (p.118).

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

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

The color table for the cell 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.
The color table is written for the cell 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.96) to initialize the structure and *G_set_color*(p.96) to set the category colors.

**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 cell file which is in another *mapset*. However, this isn’t what actually happens. It is very useful for users to create their own color tables for cell files in other *mapsets*, but without overwriting other users’ color tables for the same cell 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 color2 element as: color2/mapset/name.

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.

---

26 These routines are called by higher level routines which read or create entire color tables, such as *G_read_colors*(p.94) or *G_make_color_ramp*(p.97).
**G_init_colors** (colors)  
initialize color structure

```c
struct Colors *colors;
```

The `colors` structure is initialized for subsequent calls to `G_set_color(p.96)`.

**G_set_color** (cat, red, green, blue, colors)  
set a category color

```c
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.

**Note.** The `colors` structure must have been initialized by `G_init_colors(p.96)`.

**G_free_colors** (colors)  
free color structure memory

```c
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.94)` as well as after `G_init_colors(p.96)`.

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 can be obtained, for example, using `G_read_range(p.99)`.

**Note.** The color tables are generated without information about any particular cell file. These color tables may be created for a cell 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)  
struct Colors *colors;
CELL min, max;

Generates a color table for aspect data.

G_make_color_ramp (colors, min, max)  
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 like elevation.

G_make_color_wave (colors, min, max)  
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.

Note. This routine requires that the $(MATHLIB)$ be loaded as well.

G_make_grey_scale (colors, min, max)  
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)  
struct Colors *colors;
CELL min, max;

Generates a color table based on rainbow colors. The table generated here uses yellow, green, blue, indigo, violet, red. (Normal rainbow colors are red, orange, yellow, green, blue, indigo, and violet.) This table is good for continuous data like elevation.

§12 GIS Library
**G_make_random_colors** (colors, min, max)  

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

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

**G_make_red_yel_grn** (colors, min, max)  

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

Generates a color table similar to what **G_make_rainbow_colors** creates, except that the table starts at red, passes through yellow, and ends with green.

---

### 12.9.4. Cell History File

The history file contains documentary information about the cell file: who created it, when it was created, what was the original data source, what information is contained in the cell file, etc. This file is discussed in §5.6 Cell History File [p.29].

The following routines manage this file. They use the `History` structure which is described in §12.17 GIS Library Data Structures [p.118].

**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)  

```c
char *name;
char *mapset;
struct History *history;
```

This routine reads the history file for the cell 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)  

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

This routine writes the history file for the cell 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.99)`.

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

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

This routine initializes the history structure, recording the date, user, program name and the cell file name structure. The type is an anachronism from earlier versions of GRASS and should be specified as "cell".

**Note.** This routine only initializes the data structure. It does not write the history file.

### 12.9.5. Cell Range File

The following routines manage the cell range file. This file contains the minimum and maximum values found in the cell file. The format of this file is described in §5.7 Cell Range File (p.29).

The routines below use the `Range` data structure which is described in §12.17 GIS Library Data Structures (p.118).

**G_read_range** (name, mapset, range)  

```c
char *name;
char *mapset;
struct Range *range;
```

This routine reads the range information for the cell 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)

char *name;
struct Range *range;

This routine writes the range information for the cell 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)

struct Range *range;

Initializes the range structure for updates by G_update_range(p. 100) and G_row_update_range(p. 100).

G_update_range (cat, range)

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)

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

This routine updates the range data just like G_update_range(p. 100), but for n values from the cell array.

12.10. 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 Dig Library, which is described in §13 Dig Library [p. 123].
12.10.1. 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. 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.

```c
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:
```

27 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].

§12 GIS Library
char *mapset;
char name[50];

mapset = G_ask_vector_old("Enter vector file to be processed", name);
if (mapset == NULL)
    exit(0);

12.10.2. Finding Vector Files in the Database

Non-interactive 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,28 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." Normality, 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.


---

28 See §4.7.1 Mapset Search Path [p. 20] for more details about the search path.
29 Be warned that G_find_vector2() should not be used directly on a command line argument, since modifying argv[1] may not be valid. The argument should be copied to another character buffer which is then passed to G_find_vector2().

§12 GIS Library
For example, to find a vector file anywhere in the database:

```c
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:

```c
char name[50];

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

### 12.10.3. 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. 101)` or `G_ask_vector_in_mapset(p. 101)`, and non-interactively using `G_find_vector(p. 102)` or `G_find_vector2(p. 102)`.

```c
FILE *
G_open_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 Dig Library [p. 123].)

    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.
```
12.10.4. Creating and Opening New Vector Files

The following routine creates the new vector file \texttt{name} in the current mapset\textsuperscript{30} and opens it for writing. The vector file \texttt{name} should be obtained interactively using \texttt{G\_ask\_vector\_new(p. 101)}. If obtained non-interactively (e.g., from the command line), \texttt{G\_legal\_filename(p. 72)} should be called first to make sure that \texttt{name} is a valid GRASS file name.

\textbf{Warning.} If \texttt{name} already exists, it will be erased and re-created empty. The interactive routine \texttt{G\_ask\_vector\_new(p. 101)} guarantees that \texttt{name} will not exist, but if \texttt{name} is obtained from the command line, \texttt{name} may exist. In this case \texttt{G\_find\_vector(p. 102)} could be used to see if \texttt{name} exists.

\begin{verbatim}
FILE *
G\_fopen\_vector\_new (name)
char *name;

Creates and opens the vector file \texttt{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 Dig Library (p. 123).)

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 \texttt{dig} file), is created. None of the other vector support files are created, removed, or modified in any way.
\end{verbatim}

12.10.5. Reading and Writing Vector Files

Reading and writing vector files is handled by routines in the Dig Library. See §13 Dig Library (p. 123) for details.

12.10.6. 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.

\textsuperscript{30} GRASS doesn't allow files to be created outside the current mapset. See §4.7 Database Access Rules (p. 20).
The routines described below read and write the vector category file. They use the Categories structure which is described in §12.17 GIS Library Data Structures [p.118].

Note. The vector category file has exactly the same structure as the cell category file. In fact, it exists so that the program vect.to.cell can convert a vector file to a cell file that has an up-to-date category file.

The routines described in §12.9.2.2 Querying and Changing the Categories Structure [p.92] 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).

12.11. Site List Processing  
GRASS has a point database capability called *sites*, which manages a database of point or site information. The *sites* 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.39] for more details about the site list files.
12.11.1. 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. These routines have a built-in 'list' capability which allows the user to get a list of existing site list files.

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

```c
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.
```

```c
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.
```

```c
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:

---

31 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].

§12 GIS Library
char *mapset;
char name[50];

mapset = G_ask_sites_old("Enter site list file to be processed", name);
if (mapset == NULL)
    exit(0);

12.11.2. 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.

12.11.3. 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.107)), and desc is set to point to the description of the site.

    Returns: 1 got a site; -1 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 ("%If %If %sn", 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)  \quad write site list file

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

Writers the east and north coordinates and site description desc to the site file
opened on file descriptor fd (as returned by G_fopen_sites Neu(p. 107)).

12.12. 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( )  \quad 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() 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.

12.13. Command Line Parsing
The following two routines provide a mechanism for command line parsing. Use of these routines will standardize GRASS commands that expect command line arguments.

The routines are described first, followed by a short example (on page 112) of their usage.

G_parse_command (argc, argv, keys, stash)

parse command line

int argc;
char *argv[ ];
struct Command_keys *keys;
int (*stash)();

This routine parses command lines in any of the following formats:

- command value1 value2 value3 value4
  the options are in the correct positions
- command value1 - - value4
  the options are in the correct positions, where minuses (-) are interpreted as "accept the default for this position"
- command opt2=value2 opt4=value4 opt3=value3 opt1=value1
  the options are in mixed order, but the correct position is ascertained by looking for the "opt" string in the keys structure, which contains the "correct" position for the option.
- command value1 - opt4=value4
  a mixed form of the above formats

32 See also G_fork(p 116).
The command line parameters \texttt{argv} and the number of parameters \texttt{argc} from the \texttt{main()} routine are passed directly to \texttt{G\_parse\_command}().

The option names and positions are specified in \texttt{keys}, which is an array of \texttt{Command\_keys} structures, defined as:

```c
struct Command_keys
{
    char *alias;
    int position;
};
```

The \texttt{keys} array is terminated by a NULL \texttt{alias}. For example:

```c
struct Command_keys keys[] =
{
    {"name", 1},
    {"color", 2},
    {NULL, 0}
};
```

Once a position is determined, either by actual position or by deduction, the position number and option value are sent to the specified routine \texttt{stash()}, which should "stash" the information somewhere for later use by the program. This routine must be defined as:

```c
stash (position, value)
    int position;
    char *value;
```

and return 0 if the \texttt{value} is valid, 1 otherwise.

\texttt{G\_parse\_command}() returns the following codes:

1. There are no arguments on the command line, or the first argument is the word "help" (a usage message is printed for the user).
2. There were no errors on the command line. (This doesn't imply that all parameters were specified, just that those specified were valid).
3. There are errors on the command line (nothing is printed for the user).
This routine prints a standard usage message for the program (usually argv[0]) based on the options described in the keys parameter (which is the same as that passed to \textit{G\_parse\_command}(p,109)). The format of the message may either be \texttt{USAGE\_SHORT} for a terse format, or \texttt{USAGE\_LONG} for a longer format.
Example. The following example parses a command which expects two arguments: a
name, and a color:

```c
#include "gish"

struct Command_keys keys[] =
{
    {"name", 1},
    {"color", 2},
    {NULL, 0}
};

static char name[50];
static char color[50];
static int have_name = 0;
static int have_color = 0;

static
stash(position, value)
int position;
char *value;
{
    switch (position)
    {
        case 1:
            strcpy (name, value);
            have_name = 1;
            return 0;
        case 2:
            strcpy (color, value);
            have_color = 1;
            return 0;
    }
    return 1;
}

main (argc, argv) char *argv[];
{
    int stat;
    G_gisinit (argv[0]);
    stat = G_parse_command (argc, argv, keys, stash);
    if (stat != 0 || !have_name || !have_color)
    {
        if (stat <= 0)
            G_parse_command_usage (argv[0], keys, USAGE_LONG);
            exit1);
    }

    /* parsing complete. proceed to function implementation */
    exit(0);
}
```

§12 GIS Library
12.14. 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.

```c
char *
G_strcpy (dst, src)

char *dst, *src;

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

```c
char *
G_strncpy (dst, src, n)

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 strcpy() in that G_strncpy() ensures that dst is NULL terminated, while strcpy() does not.
```

```c
char *
G_strcat (dst, src)

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.

```c
char *
G_squeeze (s)

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.

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.

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 non-printing 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.117) for printing the
   user's interrupt character:
char G_intr_char();
char *G_unctrl();

printf("Your interrupt character is %sn", G_unctrl(G_intr_char()));

Note. G_unctrl() uses a hidden static buffer which is overwritten from call to call.

12.15. 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.

12.15.1. 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 don't protect themselves. 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.

33 Programmers who use /bin/sh know that programs run in the background (using & on the command line) are not automatically protected from keyboard interrupts. To protect a command that is run in the background, /bin/sh users must do nohup command&. Programmers who use the /bin/csh (or other variants) do not know, or forget that the C-shell automatically protects background processes from keyboard interrupts.
**G_fork()**

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.

### 12.15.2. 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.

**G_system(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.

### 12.16. 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 re-print 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. 114) 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 \(\frac{n}{total}*100\), 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. 64).

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.

12.17. 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.
12.17.1. struct Cell_head

The cell header data structure is used for two purposes. It is used for cell header information for map layers. It also used to hold window values. The structure is:

```c
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 cell headers. The `format` field describes the number of bytes per cell data value and the `compressed` field indicates if the cell file is compressed or not. The other fields apply both to cell headers and windows. 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 cell file, or in the window. See §5.3 Cell Header Format [p.26] for more information about cell headers, and §9.1 Window [p.47] for more information about windows.

The routines described in §12.9.1 Cell Header File [p.89] use this structure.

12.17.2. struct Categories

The category data structure contains map layer title and category labels. It is used both for cell files and vector files. The structure is:
The Categories structure contains a title for the map layer, the largest category in the map layer (num), an automatic label generation rule for missing labels (fmt, ml, a1, m2, a2), and a list of category labels for count specific categories.

This structure should be accessed using the routines described in §12.9.2 Cell Category File [p. 91].

12.17.3. struct Colors

The color data structure holds red, green, and blue color intensities for cell categories. The structure is:

```c
struct Colors
{
    CELL min, max; /* min, max color numbers */
    uchar *red; /* red, green, blue (0-255) */
    uchar *grn; /* allocated as needed */
    uchar *blu; /* red, green, blue for cat 0 */
    uchar r0, g0, b0; /* red, green, blue for cat 0 */
};
```

Except for category zero, the color intensities are stored in the (unsigned char) arrays red, grn, and blu. The minimum and maximum categories which have colors are min and max.

The routines described in §12.9.3 Cell Color Table [p. 94] use this structure.

The routine G_get_color(p. 95) should be used to get individual colors from the structure. However, for completeness, to find the colors for category n:
if (n != 0 && n >= min && n <= max)
{
    red[n-min]
    gmr[n-min]
    blu[n-min]
}

The color for category zero is represented by r0, g0 and b0.

## 12.17.4. struct History

The History structure is used to document cell files. The information contained here is for the user. It is not used in any operational way by GRASS. The structure is:

```c
#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 datascr_1[RECORD_LEN];
    char datascr_2[RECORD_LEN];
    char keywrd[RECORD_LEN];
    int edlinecnt;
    char edhist[MAXEDLINES][RECORD_LEN];
};
```

The `mapid` and `mapset` are the cell file name and mapset, `title` is the cell file title, `creator` is the user who created the file, `maptype` is the map type (which should always be "cell"), `datascr_1` and `datascr_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 §12.9.4 Cell History File [p.98] 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 cell 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 cell 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.
12.17.5. struct Range

The Range structure contains the minimum and maximum values which occur in a cell file. The structure is:

```c
struct Range {
    CELL nmin; /* min negative */
    CELL nmax; /* max negative */
    CELL pmin; /* min positive */
    CELL pmax; /* max positive */
};
```

Note that the range is divided into positive and negative ranges. The positive range is represented by \( pmin \) and \( pmax \), and the negative range by \( nmin \) and \( nmax \). If there are no negative values in the cell file, then both \( nmin \) and \( nmax \) will be zero. Also if there are no positive values in the file, then both \( pmin \) and \( pmax \) will be zero.

The following idiomatic expression is used to determine the full data range:

```c
min = nmin ? nmin : pmin;
max = pmax ? pmax : nmax;
```

The routines described in §12.9.5 Cell Range File [p.99] use this structure.

12.18. 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:

```bash
Gmakefile for $(GISLIB)
OBJ = main.o sub1.o sub2.o

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

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

Chapter 13

Dig Library

13.1. Introduction
The Dig Library provides the GRASS programmer with routines to process the binary dig 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.31] for details about vector files in GRASS.

The routines in the Dig 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 inter-relationships 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 the prefix dig.\(^2\) 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 §24.5 Appendix D. Index to Dig Library [p.243].

13.1.1. Include Files
The following files contain definitions and structures required by some of the routines in this library. The programmer should therefore include these files in code that uses this library.\(^3\)

\(^1\) Some of these programs are a.b.vect, b.a.vect, vect.to.cell, Dvect, Gpoly, Pmap, and Vpatch.

\(^2\) Warning. There are also 6 additional global variables and/or routines which do NOT begin with this prefix: debug, head, sample.thresh, Lines.In.Memory, Mem.Line_Ptr, and Mem.cur.position.

\(^3\) The GRASS compilation process, described in §11 Compiling GRASS Programs Using GnuCc [p.55], automatically tells the C compiler how to find this and other GRASS header files.
#include "dig_defines.h"
#include "dig_structs.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.31) and the reader should review that section. Briefly, vector data is stored as arcs representing linear, area, or point\(^4\) features. These arc types are coded as LINE, AREA, and DOT respectively, (and are #defined in the file "dig_defines.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.

*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.

Note. 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. Level One Read Access
*Level One* access allows the reading of arcs from a vector file. Most of the routines require a file descriptor fd open to read a vector file, as returned by G fopen_vector_old(p.103).

13.2.1. Initialization/Termination
The following routines perform initialization and termination actions for *Level One*

---

\(^4\) Point data in vector files is not supported under GRASS 3.0, but there are plans to support it in later versions. The routines in this library are written with this upgrade in mind.
vector access:

**dig_init (fd)**

*initialize level one vector access*

FILE *fd;

Initialize for *Level One* access. The file descriptor *fd* is rewound, the header information is extracted and stored away, and *fd* is positioned to read the first arc in the file.

Returns 0 if ok, or a negative value if error.

**Note.** This routine MUST be called before using any other *Level One* routines.

**dig_rewind (fd)**

*rewind vector file*

FILE *fd;

The file descriptor *fd* is rewound, the header information is extracted and stored away, and *fd* is positioned to read the first arc in the file.

**Note.** This routine is the same as *dig_init*p. 125).

Returns 0 if ok, or a negative value on error.

**dig_print_header ()**

*display vector header information*

After calling *dig_init*p. 125), selected information from the vector file header can be printed to stdout using this routine.

Return value is undefined.

**Warning.** It is permissible to have more than one vector file open for *Level One* access. However, this routine prints the header information extracted by the previous call to either *dig_init*(p. 125) or *dig_rewind*(p. 125).

**dig_fini (fd)**

*end level one vector access*

FILE *fd;

Terminate *Level One* access. To be called when finished accessing the vector file with *Level One* routines.

Return value is undefined.

**Note.** This routine does not close the file descriptor *fd*. Use *fclose ()* to close the file descriptor.
13.2.2. Reading Arcs
The next routines read arcs sequentially from the vector file.

`dig_read_next_line (fd, np, x, y)`

```c
FILE *fd;
int *np;
double **x, **y;
```

The points constituting the next arc in the vector file open on `fd` are read into hidden arrays. Pointers to these arrays are placed in `x` and `y`, and `np` is set to the number of points in the arc.

Returns the arc type: LINE or AREA (as defined in "dig_defines.h"), or -2 if no more arcs, or -1 on error.

Note. The DOT type is skipped by this routine.

Note. The programmer must pass `x` and `y` as addresses of pointers. For example:

```c
FILE *fd;
int np;
double *x, *y;

dig_read_next_line (fd, &np, &x, &y);
```

`dig_read_next_line_type (fd, np, x, y, type)`

```c
FILE *fd;
int *np;
double **x, **y;
int type;
```

Same as `dig_read_next_line(p. 126)` except that it limits the search to the specified `type`, which can be any combination of LINE, AREA, or DOT.

For example, to read the next LINE or AREA:

```c
FILE *fd;
int np;
double *x, *y;

dig_read_next_line_type (fd, &np, &x, &y, LINE | AREA);
```

Returns the arc type: LINE, AREA or DOT (as defined in "dig_defines.h"), or -2 if no more arcs, or -1 on error.
dig_init_box (N, S, E, W)

double N, S, E, W;

Define a window within which to search for arcs using dig_read_line_in_box(p. 127). This allows the programmer to limit the arcs retrieved to those within the window specified by N (north), S (south), E (east), and W (west).

The window must have N > S, and E > W.

Returns 0 if window is valid, or negative on error.

Note. This routine does NOT change the position of the file pointer. In particular, it does not rewind the file.

dig_read_line_in_box (fd, np, x, y)

FILE *fd;
int *np;
double **x, **y;

Same as dig_read_next_line(p. 126) except that it only looks inside the bounding box set by dig_init_box(p. 127).

Note. This routine only ignores arcs which are completely outside the bounding box. If any part of the arc falls within the bounding box, the entire arc is read, including the parts outside the box. No clipping is performed.

13.3. Level Two Read Access

This level provides full access to all the information contained in the vector file and its support files. Arc, area, and node information is available, including the internal indexes for each entity, as well as category attributes.

The indexes are unique, and can be used to distinguish one area from another, or one arc from another. Note, however, that different areas may have the same category attribute (as may different arcs).

13.3.1. Initialization/Termination

The following routines perform initialization and termination actions for Level Two
vector access:

dig_P_init (name, mapset, map)

    char *name;
    char *mapset;
    struct Map_info *map;

Initialize Level Two read access to vector file \texttt{name} in \texttt{mapset}. This routine opens any files it will need.

Return value is undefined. This routine will exit on any error and print a description of the error.

\textbf{Note.} This routine MUST be called before calling any other Level Two routines.

dig_P_finl (map)

    struct Map_info *map;

Terminate Level Two access for \texttt{map}. This routine closes any files opened by \texttt{dig_P_init(p. 128)}.

dig_P_tmp_close (map)

    struct Map_info *map;

Temporarily close access to \texttt{map}. This is useful to free one open file while not needed.

Return is undefined.

dig_P_tmp_open (map)

    struct Map_info *map;

Reopen a \texttt{map} that has been closed with \texttt{dig_P_tmp_close(p. 128)}.

Return value is undefined. If \texttt{map \rightarrow digit} == NULL, then the call failed.

13.3.2. Area Retrieval

The following routines retrieve area information.
dig_P_num_areas (map)

struct Map_info *map;

Return total number of areas in the vector map.

Note. The area indexes are numbered from 1 to n, where n is the number of areas in the vector file, as returned by this routine.

dig_P_get_area_xy (map, n, np, x, y)

struct Map_info *map;
int n;
int *np;
double **x, **y;

Given area index n, all the points for the area are read into hidden arrays. Pointers to these arrays are placed in x and y. Points are in clockwise order. The pointers x and y are valid until the next call to this routine.

Returns 0 if found, or negative on error.

Note. The programmer must pass x and y as addresses of pointers:

struct Map_info *map;
int n, np;
double *x, *y;

dig_P_get_area_xy (&map, n, &np, &x, &y);

dig_P_get_area (map, n, pa)

struct Map_info *map;
int n;
P_AREA **pa;

Given area index n, the P_AREA information for the area is read into a hidden structure. A pointer to this structure is placed in pa. The pointer pa is valid until the next call to this routine.

Returns 0 if found, or negative on error.
dig_P_area_att (map, n)
struct Map_info *map;
int n;

Given area index n, return its category number.

Returns 0 if not an area or if unlabeled.

dig_P_get_area_bbox (map, n, N, S, E, W)
struct Map_info *map;
int n;

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.

13.3.3. Arc Retrieval
The following routines retrieve arc information.

dig_P_num_lines (map)
struct Map_info *map;

Returns total number of arcs in the vector map.

Note. The arc indexes are numbered from 1 to n, where n is the number of arcs in the vector file, as returned by this routine.

dig_P_read_line (map, n, p)
struct Map_info *map;
int n;
struct line_pnts **p;

Given arc index n, the points for the arc are read into a hidden line_pnts structure. A pointer to this structure is placed in p. The pointer p is valid until the next call to this routine or to dig_P_read_next_line(p.131).

Returns the same values as dig_read_next_line(p.126).
dig_P_read_next_line (map, p)
struct Map_info *map;
struct line_pnts **p;

The points for the next arc in the vector map are read into a hidden line_pnts
structure. A pointer to this structure is placed in p. The pointer p is valid until
the next call to this routine or to dig_P_read_line(p. 130).

Returns the same values as dig_read_next_line(p. 126).

dig_P_rewind (map)
struct Map_info *map;

Resets the next-arc pointer to beginning of list. For use with
dig_P_read_next_line(p. 131).

Return is undefined.

dig_P_line_att (map, n)
struct Map_info *map;
int n;

Given arc index n, return its category number.

Returns 0 if not labeled or on error.

dig_P_get_line_bbox (map, n, N, S, E, W)
struct Map_info *map;
int n;

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 negative on error.

13.3.4. Area Analysis Tools
The following routines provide some area-related analyses.
**dig_point_to_area** (map, x, y)

struct Map_info *map;
double x, y;

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

**dig_point_in_area** (map, x, y, pa)

struct Map_info *map;
double x, y;
P_AREA *pa;

Given a filled P_AREA structure pa, determines if x,y is within the area. The structure pa can be filled with dig_P_get_area(p. 129).

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.

---

**13.3.5. Arc Analysis Tools**

The following routines provide some arc-related analyses.

**dig_point_to_line** (map, x, y, type)

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 (e.g., LINE | AREA), or (char)-1 if you want to search all arc types.

**dig_check_dist** (map, n, x, y, d)

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.

Returns the number of the segment that was closest, or -1 on error. The segment number, in combination with **dig_P_read_line**(p. 130) can be used to determine the end-points of the closest line-segment in the arc:
struct Map_info map;
double x,y,d;
double x1,y1,x2,y2;
int n;
struct line_pnts *p;

if ((s = dig_check_dist(&map, n, x, y, &d)) > 0)
{
    dig_P_read_line (&map, n, &p);
    x1 = p->x[s-1];
    y1 = p->y[s-1];
    x2 = p->x[s];
    y2 = p->y[s];
}

13.4. Writing Binary Dig files
The following routines are provided for import and export capabilities.

Note. The file descriptors required by these routines should be either open for writing, or for reading, but not for both writing and reading.

long
dig_Write_line (fd, type, x, y, np)

    FILE *fd;
    char type;
    double *x, *y;
    int np;

   Writes the arc, defined by the np points in the x and y arrays, to the end of the binary dig vector file open on file descriptor fd. The arc type must be one of LINE, AREA, or DOT.

Returns the offset in the file where the arc was written. This offset can be used with dig_Read_line(p. 133).

dig_Read_line (fd, offset, x, y, np)

    FILE *fd;
    long offset;
    double **x, **y;
    int *np;

    Seeks to the specified offset on file descriptor fd and reads the arc which begins there into hidden arrays. Pointers to these arrays are then placed into x and y and np is set to the number of points in the arc.

    Return is the same as dig_read_next_line(p. 126) from Level One.
Note. The programmer must pass \( x \) and \( y \) as addresses of pointers:

```c
FILE *fd;
long offset;
int np;
double *x, *y;
```

dig_read_line (fd, offset, &x, &y, &np);

**dig_read_head_binary** (fd, header)

read vector header

```c
FILE *fd;
struct dig_head *header;
```

Reads the header from the binary dig vector file open on file descriptor \( fd \). It can be used to position \( fd \) ready to read the first arc in the file.

Currently only returns 0.

**Note.** If using *Level One* routines, it is unnecessary to call this routine.

**dig_write_head_binary** (fd, header)

write vector header

```c
FILE *fd;
struct dig_head *header;
```

Writes the header information to the binary dig vector file open on file descriptor \( fd \). This routine must be the first to write to a new vector file. After the header has been written, arcs can be sequentially written to the file. It can also be used to rewrite the header information after the entire file has been written, if necessary.

Currently only returns 0.

### 13.5. Miscellaneous Tools

**double**

**dig_distance2_point_to_line** (x, y, x1, y1, x2, y2)

distance to line-segment

```c
double x, y;
double x1, y1, x2, y2;
```

Computes the square of the minimum distance from point \( x,y \) to the line-segment \( x1,y1,x2,y2 \).

Returns the distance squared.
double
dig_xy_distance2_point_to_line (x,y,x1,y1,x2,y2)

distance to line-segment

double *x, *y;
double x1, y1, x2, y2;

Returns the square of the minimum distance from point x,y to the line-segment x1,y1,x2,y2.

Changes x,y to the point on the segment x1,y1,x2,y2 which is closest to x,y.

dig_prune (p, threshold)

prune a dense arc

struct line_pnts *p;
double threshold;

Given a filled line_pnts structure p, prune it within the specified threshold. This function is used to reduce the number of points needed to define an arc within a given accuracy.

Returns the new number of points.

dig_bound_box (p, N, S, E, W)

get arc bounding box

struct line_pnts *p;

Given a filled line_pnts structure p containing a list of X,Y coordinates, compute the bounding box of this list.

Returns non-zero on error.

13.6. Loading the Dig Library

The library is loaded by specifying $(DIGLIB)\footnote{This variable was NOT defined in releases 3.0 and 3.0A. Edit the file $GISEASE/src/CMC/make.mid and add the line: DIGLIB=$(SRC)/mapcele/lib/diglib.a at the bottom of the file.} in the Gmakefile. The following example is a complete Gmakefile which compiles code that uses this library:
Note. EXTRA_CFLAGS tells the C compiler where additional #include files are located. This is necessary since the required #include files do not live in the normal GRASS #include directory.

See §11 Compiling GRASS Programs Using Gmake (p.55) for a complete discussion of Gmakefiles.
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. 41] for a description of imagery groups, and §5 Grid Cell 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 inter-relationships 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

---

1 Since this is a new library, it is expected to grow. Hopefully, image analysis functions will be added to complement the database functions already in the library.

2 See §8.4 Imagery Programs [p. 45] for a list of some imagery programs.

3 The GRASS compilation process, described in §11 Compiling GRASS Programs Using Gnucke [p. 55], automatically tells the C compiler how to find this and other GRASS header files.
variables or routines in their own programs which use this prefix.

An alphabetic index is provided in §24.5 Appendix E. Index to Imagery Library [p.245].

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. 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. 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.

\[\text{Lask \_ group \_ old (prompt, group)}\]

\[
\begin{align*}
\text{char *prompt;}
\text{char *group;}
\end{align*}
\]

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

\[\text{Lask \_ group \_ new (prompt, group)}\]

\[
\begin{align*}
\text{char *prompt;}
\text{char *group;}
\end{align*}
\]

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].
139

139

L_ask_group_any (prompt, group)  

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 L_find_group(p. 139) 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:

```c
char group[50];
if (!L_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:

L_find_group (group)  

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. 144].

The contents of the REF file are read or updated by the following routines:
I_get_group_ref (group, ref)  
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)  
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 doesn't already exist.

I_get_subgroup_ref (group, subgroup, ref)  
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)  
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 doesn’t 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. 140) and I_get_subgroup_ref(p. 140) 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 isn't 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 cell 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. 144].

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)
   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)
   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 shouldn't be used by other programs.

Note. This routine does not validate the target information.

I_put_target (group, location, mapset)
   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 shouldn't 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. 144].

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)  
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 doesn’t exist.

I_new_control_point (cp, e1, n1, e2, n2, status)  
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

---

6 Use of this routine implies that the point is probably good, so status should be set to 1.
I_put_control_points (group, cp)

char *group;
struct Control_Points *cp;

Writes the control points from the cp structure to the POINTS file for the specified group.

Note. Points in cp with a negative status are not written to the POINTS file.

14.3. Loading the Imagery Library

The library is loaded by specifying $(IMAGERYLIB) in the Gmakefile. The following example is a complete Gmakefile which compiles code that uses this library:

```bash
OBJ = main.o sub1.o sub2.o
pgmx $(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. 63) for details on that library.


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 cell files), file (the name and mapset of each file), and red, grn, blu (color information for the group or subgroup).

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. 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 cell 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
", 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 don't exist.

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

§14 Imagery Library
The number of control points is `count`. Control point `i` is `el[i]`, `n1[i]`, `e2[i]`, `n2[i]`, and its status is `status[i]`. 

```c
struct Control_Points {
    int count;    /* number of control points */
    double *el;   /* image east (column) */
    double *n1;   /* image north (row) */
    double *e2;   /* target east */
    double *n2;   /* target north */
    int *status;  /* status of control point */
};
```
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 $R_{\text{screen\_left}}$ (p. 150), $R_{\text{screen\_rite}}$ (p. 150), $R_{\text{screen\_bot}}$ (p. 150), and $R_{\text{screen\_top}}$ (p. 150).

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

An alphabetic index is provided in §24.5 Appendix G. Index to Raster Graphics Library (p.249).

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 monitor 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 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 graphics screen.

Returns 0 if successful, non-zero if unsuccessful.

R_color_table_float()  
Select 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 device driver program so that the colors involved point to the closest available color on the device.

R_reset_color(red, green, blu, num)

    unsigned char red, green, blue;
    int num;

Set color number num to the intensities represented by red, green, and blu.

R_reset_colors(min, max, red, green, blue)

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

Set color numbers min through max to the intensities represented in the arrays red, green, and blu.

R_color(color)

    int color;

Selects the color to be used in subsequent draw commands.
R_standard_color(color)

```c
int color;
```

Selects the standard color to be used in subsequent draw commands. The color value is best retrieved using D_translate_color(p.167). See §16 Display Graphics Library (p.159).

R_RGB_color(red,green,blue)

```c
int red, green, blue;
```

When in float mode (see R_color_table_float(p.149)), 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()

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

R_screen_top()

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

R_screen_left()

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

R_screen_rite()

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

R_move_abs(x,y)

```c
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)

```c
text dx, dy;
```

*move current location*

Shift the current screen location by the values in dx and dy:

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

Nothing is drawn on the screen.

**R_cont_abs** (x,y)

```c
text x, y;
```

*draw line*

Draw a line using the current color, selected via *R_color(p. 149)*, from the current location to the location specified by x,y. The current location is updated to x,y.

**R_cont_rel** (dx,dy)

```c
text dx, dy;
```

*draw line*

Draw a line using the current color, selected via *R_color(p. 149)*, from the current location to the relative location specified by dx and dy. The current location is updated:

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

**R_box_abs** (x1,y1,x2,y2)

```c
int x1,y1;
int x2,y2;
```

*fill a box*

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)

```c
int dx, dy;
```

*fill a box*

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:

```c
Newx = Oldx + dx;
Newy = Oldy + dy;
```
R.\texttt{erase} ()

Erases the entire screen to black.

R.\texttt{flush} ()

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

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.

\texttt{R.polydots.abs}(x,y,\texttt{num})

draw a series of dots

\begin{verbatim}
int *x, *y;
int num;
\end{verbatim}

Pixels at the \texttt{num} absolute positions in the \texttt{x} and \texttt{y} arrays are turned to the current color. The current position is left updated to the position of the last dot.

\texttt{R.polydots.rel}(x,y,\texttt{num})

draw a series of dots

\begin{verbatim}
int *x, *y;
int num;
\end{verbatim}

Pixels at the \texttt{num} relative positions in the \texttt{x} and \texttt{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.

\texttt{R.polygon.abs}(x,y,\texttt{num})

draw a closed polygon

\begin{verbatim}
int *x, *y;
int num;
\end{verbatim}

The \texttt{num} absolute positions in the \texttt{x} and \texttt{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)

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)

int *x, *y;
int num;

The num absolute positions in the x and y arrays are used to generate a multi-segment 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)

int *x, *y;
int num;

The num relative positions in the x and y arrays are used to generate a multi-segment 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)

```c
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)

```c
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. 154) 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)

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

This is useful only in fixed color mode (see **R_color_table_fixed** (p. 149)). 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 sent with **R_set_RGB_color** (p. 154). 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 programmers specifications.
R_set_window (top,bottom,left,right) set text clipping window

int top, bottom, left, right;

Subsequent calls to R_text(p.156) will have text strings clipped to the screen window defined by top, bottom, left, right.

R_font (font) choose font

cchar *font;

Set current font to font. Available fonts are:

<table>
<thead>
<tr>
<th>Font Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cyrillic</td>
<td>cyrillic</td>
</tr>
<tr>
<td>gothgbt</td>
<td>Gothic Great Britain triplex</td>
</tr>
<tr>
<td>gothgbt</td>
<td>Gothic German triplex</td>
</tr>
<tr>
<td>gothitt</td>
<td>Gothic Italian triplex</td>
</tr>
<tr>
<td>greekc</td>
<td>Greek complex</td>
</tr>
<tr>
<td>greeks</td>
<td>Greek complex script</td>
</tr>
<tr>
<td>greekp</td>
<td>Greek plain</td>
</tr>
<tr>
<td>greekp</td>
<td>Greek plain script</td>
</tr>
<tr>
<td>greeks</td>
<td>Greek simplex</td>
</tr>
<tr>
<td>italicc</td>
<td>Italian complex</td>
</tr>
<tr>
<td>italics</td>
<td>Italian complex small</td>
</tr>
<tr>
<td>italict</td>
<td>Italian triplex</td>
</tr>
<tr>
<td>romanp</td>
<td>Roman complex</td>
</tr>
<tr>
<td>romancs</td>
<td>Roman complex small</td>
</tr>
<tr>
<td>romand</td>
<td>Roman duplex</td>
</tr>
<tr>
<td>romans</td>
<td>Roman simplex</td>
</tr>
<tr>
<td>scriptc</td>
<td>Script complex</td>
</tr>
<tr>
<td>scripts</td>
<td>Script simplex</td>
</tr>
</tbody>
</table>

R_text_size (width, height) set text size

int width, height;

Sets text pixel width and height to width and height.
R_text (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)

    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)

    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)

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

Similar to R_get_location_with_pointer(p.156) 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,rx,ny,button) get mouse location using a box

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

    Identical to R_get_location_with_line(p. 156) 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
    # 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. 63] for details on that library.

This library is usually loaded with the $(DISPLAYLIB). See §16 Display Graphics Library [p. 159] for details on that library.

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 3.0 graphics commands operate. Such programs like Dvect, Dgraph, and Dcell demonstrate how these routines work together. The routines fall into four basic sets: 1) window creation and management, 2) coordinate conversion routines, 3) specialized efficient raster display routines, and 4) assorted miscellaneous routines like command line parsing 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 §24.5 Appendix F. Index to Display Graphics Library [p. 247].

16.2. Window Management

The following set of routines creates, destroys, and otherwise manages graphics windows.

§16 Display Graphics Library
**D_new_window** (name, top, bottom, left, right)  
create new graphics window

```c
char *name;
int top, bottom, left, right;
```

Creates a new window `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 window

```c
char *name;
```

Selects the window `name` to be the current window. The previous current window (if there was one) is outlined in grey. The selected current window is outlined in white.

**D_get_cur_wind** (name)  
identify current graphics window

```c
char *name;
```

Captures the name of the current window in string `name`.

**D_show_window** (color)  
outlines current window

```c
int color;
```

Outlines current window in `color`. Appropriate colors are found in `$GISBASE/src/D/libes/colors.h`\(^1\) and are spelled with lower-case letters.

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

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

Returns current window's coordinates in the pointers `top`, `bottom`, `left`, and `right`.

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

```c
struct Cell_head *window;
```

Graphics windows can have GRASS map windows associated with them. This routine passes the map `window` to the current graphics window. If a GRASS window is already associated with the graphics window, its information is copied into `window` for use by the calling program. Otherwise `window` is associated with the current graphics window.

---

\(^1\) `$GISBASE` is the directory where GRASS is installed. See §10.1 UNIX Environment (p.51) for details.
D_reset_screen_window (top, bottom, left, right) resets current window position
    int top, bottom, left, right;
    Re-establishes the screen position of a window at the location specified by top, bottom, left, and right.

D_timestamp () give current time to window
    Timestamp the current window. This is used primarily to identify which windows are on top of which others.

D_erase_window () erase current window
    Erases the window on screen using the currently selected color.

D_remove_window () remove a window
    Remove any trace of current window.

D_clear_window () clears information about current window
    Removes all information about current window. This includes the map window and the window content lists.

16.3. Window Contents Management
    This special set of graphics window management routines maintains lists of window contents.

D_add_to_list (string) add command to window 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 (grid cell) maps. The D_set_cell_name() routine is used for this special case.
D_set_cell_name (name)
char *name;
Stores the cell file name in the information associated with the current window.

D_get_cell_name (name)
char *name;
Retuns the name of the cell file associated with the current window.

D_clear_window ()
clear window display lists
Removes all display information lists associated with the current window.

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

<table>
<thead>
<tr>
<th>Coordinate system</th>
<th>Origin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Display screen</td>
<td>upper left (NW)</td>
</tr>
<tr>
<td>Earth window</td>
<td>lower left (SW)</td>
</tr>
<tr>
<td>Array window</td>
<td>upper left (NW)</td>
</tr>
</tbody>
</table>

Display screen coordinates are the physical coordinates of the display screen and are referred to as x and y. Earth window coordinates are from the GRASS database windows and are referred to as east and north. Array coordinates are the columns and rows relative to the GRASS window and are referred to as column and row.

The routine D_do_conversions() is called to establish the relationships between these different systems. Then a wide variety of accompanying calls provide access to conversion factors as well as conversion routines.

D_do_conversions (window, top, bottom, left, right)
struct Cell_head *window;
int top, bottom, right, left;

The relationship between the earth window and the top, bottom, left, and right screen coordinates is established, which then allows conversions between all three coordinate systems to be performed.

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 stands for UTM).

```c
double D_u_to_a_row(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) {
    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) {
    double row;
    Returns a y value in the screen coordinate system when provided the corresponding row value in the array coordinate system.
}

double D_a_to_d_col(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) {
    double north;
    Returns a y value in the screen coordinate system when provided the corresponding north value in the earth coordinate system.
}
```

§16 Display Graphics Library
double
D_u_to_d_col (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)
    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)
    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)
    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)
    double x;
    Returns a column value in the array coordinate system when provided the corresponding x value in the screen coordinate system.

If the above routines prove too inefficient, the programmer can examine the source code for these routines to see how the conversions are done and create new conversion routines.

16.5. 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 `combine` and `weight` which, under the user's option, display graphics results immediately to the screen.

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

```c
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.160)` for the current window.

**D_draw_cell_row** (row, raster)  
render a raster row

```c
int row;
CELL *raster;
```

The `row` gives the map array row. The `raster` array provides the categories for each map grid cell in that row. 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_overlay_cell_row** (row, raster)  
render a raster row without zeros

```c
int row;
CELL *raster;
```

Equivalent to `D_draw_cell_row()` except that locations with category 0 are left untouched, rather than being covered with the color for category 0.

### 16.6. 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)  

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:

\[
\begin{align*}
    w &< e \\
    s &< n
\end{align*}
\]

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.

16.7. Pop-up Menus

D_popup (bcolor, tcolor, dcolor, top, left, size, options)  

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. For examples of how to use this routine see the source code for the GRASS 3.0 display program.\textsuperscript{2} 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

\textsuperscript{2} The source code for display is under $GISBASE/src/D/prog_inter/display.
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.

16.8. Colors

D_reset_colors(colors)

```
struct Colors *colors;
```

set colors in driver

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 §12.9.3 Cell Color Table [p. 94] for GIS Library routines which use this structure.

D_translate_color(name)

```
char *name;
```

color name to number

Takes a color name in ascii and returns the color number for that color. Returns 0 if color is not known. The color number returned is for lines and text, not raster graphics.

16.9. 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)

<table>
<thead>
<tr>
<th>OBJ = main.o sub1.o sub2.o</th>
</tr>
</thead>
<tbody>
<tr>
<td>prog: $(OBJ) $(DISPLAYLIB) $(RASTERLIB) $(GISLIB)</td>
</tr>
<tr>
<td>$(CC) $(LDFLAGS) -o $(OBJ) $(DISPLAYLIB)</td>
</tr>
<tr>
<td>$(RASTERLIB) $(GISLIB)</td>
</tr>
<tr>
<td>$(DISPLAYLIB): # in case the library changes</td>
</tr>
<tr>
<td>$(RASTERLIB): # in case the library changes</td>
</tr>
<tr>
<td>$(GISLIB): # in case the library changes</td>
</tr>
</tbody>
</table>


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

\texttt{lockfile (file, pid)} \hspace{1cm} \textit{create a lock}

char *file;
int pid;

This routine decides if the lock can be set and, if so, sets the lock. If \texttt{file} does not exist, the lock is set by creating the file and writing the \texttt{pid} (process id) into the file. If \texttt{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 \texttt{pid} into the file.

Return codes:

1 \hspace{1cm} \text{ok, lock request was successful}
0 \hspace{1cm} \text{sorry, another process already has the file locked}
-1 \hspace{1cm} \text{error, could not create the file}
-2 \hspace{1cm} \text{error, could not read the file}
-3 \hspace{1cm} \text{error, could not write the file}
unlock_file (file)

    char *file;

    This routine releases the lock by unlinking file. This routine does NOT check to
    see that the process unlocking the file is the one which created the lock. The file
    is simply unlinked. Programs should of course unlock the lock if they created it.
    (Note, however, that the mechanism correctly handles abandoned locks.)

Return codes:

    1  ok, lock file was removed
    0  ok, lock file was never there
   -1  error, lock file remained after attempt to remove it.

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 under GRASS 3.0.
Gn makefile for $(LOCKLIB)

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

pgmr $(OBJ) $(LOCKLIB)
  $(CC) $(LDFLAGS) -o $@ $(OBJ) $(LOCKLIB)

$(LOCKLIB): # in case the library changes
```

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 §24.5 Appendix H. Index to Rowio Library (p. 251).
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"
```

```c
rowio_setup (r, fd, nrows, len, getrow, putrow)
```

```c
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:

```c
getrow (fd, buf, n, len)
```

```c
int fd;
char *buf;
int n, len;
```

When called, getrow() should read data for row n from file descriptor fd into buf for len bytes. It should return 1 if the data is read ok, 0 if not.

The putrow() routine will be called as follows:

¹ The GRASS compilation process, described in §11 Compiling GRASS Programs Using Grmake (p. 55), automatically tells the C compiler how to find this and other GRASS header files.
**putrow** (fd, buf, n, len)

```c
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)

```c
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.174) 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.176), the **putrow()** routine specified in **rowio_setup**(p.174) 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)

```c
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.175) is later modified directly without also writing it to the file. See §18.3 Rowio Programming Considerations [p.176].
rowio_fileno (r)  
ROWIO *r;
Rowio_fileno() returns the file descriptor associated with the ROWIO structure.

rowio_release (r)  
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)  
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.174) is called to write row n to the file.

rowio_flush (r)  
ROWIO *r;
Rowio_flush() forces all rows modified by rowio_put(p.176) 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 isn't 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 $(ROWIOLIB)$ in the Gmakefile. The following example is a complete Gmakefile which compiles code that uses this library:

```
Gnakefile for $(ROWIOLIB)
OBJ = main.o sub1.o sub2.o
pgm: $(OBJ) $(ROWIOLIB)
   $(CC) $(LDFLAGS) -o $@ $(OBJ) $(ROWIOLIB)
$(ROWIOLIB): # in case the library changes
```


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

Large data files which contain data in a matrix format often need to be accessed in a non-sequential or random manner. This requirement complicates the programming. Methods for accessing the data are:

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 non-segmented 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 §24.5 Appendix I. Index to Segment Library [p. 253].

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:

```c
#include "segment.h"
```

The first step is to create a file which is properly formatted for use by the Segment Library routines:

```c
void segment_format (int fd, int nrows, int ncols, int nrows, int ncols, int len);
```

The segmentation routines require a disk file to be used for paging segments in and out of memory. This routine formats the file open for write on file descriptor `fd` for use as a segment file. A segment file must be formatted before it can be processed by other segment routines. The configuration parameters `nrows`, `ncols`, `srows`, `scols`, and `len` are written to the beginning of the segment file which is then filled with zeros.

The corresponding non-segmented 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. 180).

---

1 The GRASS compilation process, described in §11 Compiling GRASS Programs Using Gnake (p. 55), automatically tells the C compiler how to find this and other GRASS header files.

§19 Segment Library
segment_init (seg, fd, nsegs)

```
SEGMENT *seg;
int fd, nsegs;
```

Initializes the seg structure. The file on fd is a segment file created by `segment_format` (p. 180) 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. 180), 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)
```

```
SEGMENT *seg;
char *buf;
int row;
```

Transfers non-segmented matrix data, row by row, into a segment file. Seg is the segment structure that was configured from a call to `segment_init` (p. 181). 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)
```

```
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.


**segment_put (seg, value, row, col)**

```
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)**

```
SEGMENT *seg;
```

Forces all pending updates generated by `segment_put(p. 182)` 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. 182)`.

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)**

```
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. 181)`.

Return codes are: 1 if ok; else -1 could not seek or read segment file.

Finally, memory allocated in the SEGMENT structure is freed:
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:

```c
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_putrow(&seg, buf, row);
}
```

The next step is the conversion of the non-segmented 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:

```c
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_putrow(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_putrow(), could be omitted, since segment_format(p.180) will fill the segment file with zeros.

The data can now be accessed directly using segment_get(p.181). For example, to get the value at a given row and column:
int value;
SEGMENT seg;

segment_get (&seg, &value, row, col);

Similarly segment_put(p.182) 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 won't work */

Once the random access processing is complete, the data would be extracted from the segment file and written to a non-segmented matrix data file as follows:

segment_flush (&seg);

for (row = 0; row < rows; row++)
{
    segment_get_row (&seg, buf, row);
    <code to put buf into a matrix data file for row>
}

Finally, the memory allocated for use by the segment routines would be released and the file closed:

segment_release (&seg);
close (fd);

Note. The Segment Library does not know the name of the segment file. It does not attempt to remove the file. If the file is only temporary, the programmer should remove the file after closing it.
19.4. Loading the Segment Library

The library is loaded by specifying \$(SEGMENTLIB) in the Gmakefile. The following example is a complete Gmakefile which compiles code that uses this library:

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

pgm: $(OBJ) $(SEGMENTLIB)
   $(CC) $(LDFLAGS) -o @$(OBJ) $(SEGMENTLIB)

$(SEGMENTLIB): # in case the library changes
```

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 §24.5 Appendix J. Index to Vask Library (p.255).

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 window command is a good example, as are reclass and mask.
V_clear ()

This routine initializes the screen description information, and must be called before each new screen layout description.

V_line (num, text)

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)
V_quest (value, type, row, col, len)

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_quest() 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_quest() 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_quest() 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_const(), but it is a
stumbling block for \texttt{V\_const()}. Also, the programmer must initialize \texttt{value} prior to calling these routines.\footnote{Technically \texttt{value} needs to be initialized before the call to \texttt{V\_call()} since \texttt{V\_const()} and \texttt{V\_ques()} only store the address of \texttt{value}. \texttt{V\_call()} looks up the values and places them on the screen.}

\texttt{V\_float\_accuracy (num)}  \hspace{1cm} \textit{set number of decimal places}

\begin{verbatim}
int num;
\end{verbatim}

\texttt{V\_float\_accuracy()} defines the number of decimal places in which floats and doubles are displayed or accepted. \texttt{Num} is an integer value defining the number of decimal places to be used. This routine affects subsequent calls to \texttt{V\_const()} and \texttt{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 \texttt{V\_float\_accuracy()} before calls to \texttt{V\_ques()} or \texttt{V\_const()}. \texttt{V\_clear()} resets the number of decimal places to 2.

\texttt{V\_call ()}  \hspace{1cm} \textit{interact with the user}

\texttt{V\_call()} clears the screen and writes the text and data values specified by \texttt{V\_line()}, \texttt{V\_ques()} and \texttt{V\_const()} to the screen. It interfaces with the user, collecting user responses in the \texttt{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. \texttt{V\_call()} ends when the user hits ESC and returns a value of 1 (but see \texttt{V\_intrpt\_ok()} below).

No error checking is done by \texttt{V\_call()}. Instead, all variables used in \texttt{V\_ques()} calls must be checked upon return from \texttt{V\_call()}. If the user has supplied inappropriate information, the user can be informed, and the input prompted for again by further calls to \texttt{V\_call()}.

\texttt{V\_intrpt\_ok ()}  \hspace{1cm} \textit{allow ctrl-c}

\texttt{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 \texttt{V\_intrpt\_ok()} before \texttt{V\_call()}. This allows the user to enter Ctrl-C, which causes \texttt{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 \texttt{V\_call()} is moved up to line 22.

\textbf{Note.} When \texttt{V\_intrpt\_ok()} is called, the programmer must limit the use of \texttt{V\_line()}, \texttt{V\_ques()}, and \texttt{V\_const()} to lines 0-21.
**V_intrpt_msg** (text)

`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.

```c
#define LEN 15
main()
{
    int i;
    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:
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 non-destructively within the field.
- `CTRL-L` moves the cursor forward non-destructively within the field.
- `CTRL-A` writes a copy of the screen to a file named `visualask` 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:


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)`.


### 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, non-numeric responses (e.g., "abc") in an integer field will get turned to a 0, and floating point numbers will be truncated (e.g., 54.87 will become 54).

No error checking (other than matching input with variable type for that input field) is done by `V_call()`. This must be done, by the programmer, upon return from `V_call()`.

Calls to `V_line()`, `V_ques()`, and `V_const()` store only pointers, not contents of memory. At the time of the call to `V_call()`, the contents of memory at these addresses are copied into the appropriate places of the screen description. Care should be taken to use distinct pointers for different fields and lines of text. For example, the following mistake should be avoided:
char text[100];
V_clear();
sprintf(text," Welcome to GRASS ");
V_line(3,text);
sprintf(text," which is a product of the US Army CERL ");
V_line(5,text);
V_call();

since this results in the following (unintended) screen:

```
which is a product of the 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, 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().
Chapter 21

Writing a Digitizer Driver

21.1. Introduction
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 §21.2 Writing the Digitizer Device Driver [p. 195] explains how digitizer drivers are written, while section §21.3 Discussion of the Finer Points (Hints) [p. 203] describes problems and pitfalls encountered during the development of the Altek driver.

21.2. Writing the Digitizer Device Driver
Source code for the digitizer drivers is kept in

$GISBASE/src/mapdev/digitizers

Separate sub-directories contain the individual drivers. When a new driver is written, it should be placed here in a new sub-directory.

It is helpful to examine the source code for existing drivers located here, and to attend a demonstration of the GRASS digitizing program digi, before developing a new driver.

21.2.1. Functions to 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.)

1 $GISBASE is the directory where GRASS is installed. See §10.1 UNIX Environment [p. 51] for details.
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 §21.2.3 Compiling the Device Driver (p.202)), other file 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:

```c
char *dig_menu[];
```

An example of how the Altek driver uses this function to create a menu is given below:

```c
#define dig_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:
```c
#include <curses.h>
#include "dig_menu.h"
#include "../digit/digit.h"
#include "../digit/menu.h"
#include "../liba/headers.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.199).

dig_dev.c
D_get_scale(scale)
    float *scale;

This function sets scale to the digitizer resolution in units of lines per inch. For example, on a digitizer having a resolution of 1000 lines per inch, scale would be set to .001.

collPts.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.201), and plotted on the graphics monitor using plot_points(p.201). 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.

§21 Writing a Digitizer Driver
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.201)`, 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. 198).

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. 198).

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. 202), and the program terminated with a call to close_down(p. 201).

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. 202) 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 §21.3 Discussion of the Finer Points (Hints) [p. 203] 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. 202), and the program terminated with a call to close_down(p. 201).

\(^4\) Due to the design of the GeoGraphics digitizer, it isn’t possible to detect whether or not the user properly sets the origin. If the origin is improperly set, the map will be improperly registered.

§21 Writing a Digitizer Driver
The following two routines read the current digitizer coordinates:

```c
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.

```c
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.

### 21.2.2 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 §21.2.3 Compiling the Device Driver [p.202].
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. 197). 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. 197) in the coordinate system of the database. Digitizer coordinates can be translated to database coordinates using transform_a_into_b(p. 201).

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 CLR_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. 197).

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. 197) calls transform_a_into_b().

§21 Writing a Digitizer Driver
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.

21.2.3. 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. 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/libdigit_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.

21.2.4. 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.197)
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

---

5 See §11 Compiling GRASS Programs Using Gmake (p.59) for a discussion of Gmake and
Gmakefiles.
A final test of the driver is performed when the `collect_points()` function is called, which occurs when vectors are being digitized.

Before testing any programs, review the *Grass 3.0 Installation Guide* to ensure that the digitizer is set up correctly. If more information is needed, read the file `$GISBASE/src/mapdev/README`.

### 21.3. 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.

#### 21.3.1. 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)

#### 21.3.1.1. 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 can't transmit information on command, then it can't be used as a GRASS digitizer.

#### 21.3.1.2. 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.

21.3.1.3. 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/INSTALLALTEK 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.

21.3.2. 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.).

21.3.3. 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

§21 Writing a Digitizer Driver
Connecting to the digitizer:

In GRASS 3.0, 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.

\( \text{D_setup_driver(p. 197)} \) 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 \( \text{ioctl()} \). 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 pre-processor \#ifdef SYSV\(^6\) statements 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 \( \text{D_setup_driver(p. 197)} \).

Initializing and reading the digitizer:

The driver and the digitizer communicate by using the UNIX read() and write() functions. \( \text{D_setup_driver(p. 197)} \) 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 isn't 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 time-out occurred. This approach worked well in the Altek driver.

\(^{6}\) SYSV is defined by Gmake. See §11 Compiling GRASS Programs Using Gmake [p. 53].
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 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` isn't set up properly in `D_setup_driver()` (p. 197), 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 re-initializes 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.

§21 Writing a Digitizer Driver
Chapter 22

Writing a Graphics Driver

22.1. Introduction
GRASS 3.0 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 programs monitor, Dlist.mon, Drelease.mon, Dselect.mon, Dstart.mon, Dstatus.mon, Dstop.mon and Dwhich.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.

22.2. Basics
The various drivers have source code contained under the directory $GISBASE/src/D/devices. 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.

\footnote{$GISBASE$ is the directory where GRASS is installed. See §10.1 UNIX Environment for details.}
22.3. Basic Routines
Described here are the basic routines required for constructing a new GRASS 3.0 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).

22.3.1. Open/Close Device

Graph_Set ()
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:

\[
\begin{align*}
\text{SCREEN_LEFT} &\leq \text{SCREEN_RIGHT} \\
\text{SCREEN_TOP} &\leq \text{SCREEN_BOTTOM}
\end{align*}
\]

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 ()
Close down the graphics processing. This gets called only at driver termination time.

22.3.2. Return Edge and Color Values
The four raster edge values set in the Graph_Set() routine above are retrieved with the following routines.
22 Writing a Graphics Driver

Screen_left (index)

Screen_rite (index)

Screen_top (index)

Screen_bot (index)

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)

int *index;

The number of colors is returned in index.

gt_num_colors ()

The number of colors is returned directly.

22.3.3. 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)

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)

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.

22.3.4. 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() \hspace{1cm} \textit{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.

\texttt{reset_color(number, red, green, blue)} \hspace{1cm} \textit{set a color}

\begin{verbatim}
  int number;
  unsigned char red, green, blue;
\end{verbatim}

The system's color represented by \texttt{number} is set using the color component intensities found in the \texttt{red}, \texttt{green}, and \texttt{blue} variables. A value of 0 represents 0\% intensity; a value of 255 represents 100\% intensity.

\texttt{color(number)} \hspace{1cm} \textit{select a color}

\begin{verbatim}
  int number;
\end{verbatim}

The current color is set to \texttt{number}. This number points to the color combination defined in the last call to \texttt{reset_color()} that referenced this number.

### 22.3.5. Mouse Input

The user provides input through the three following routines.

\texttt{Get_location_with_box(cx,cy,wx,wy,button)} \hspace{1cm} \textit{get location with rubber box}

\begin{verbatim}
  int cx, cy;
  int *wx, *wy;
  int *button;
\end{verbatim}

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 \texttt{cx,cy} coordinate. The opposite coordinate starts out at \texttt{wx,wy} and then tracks the mouse. Upon button depression, the current coordinate is returned in \texttt{wx,wy} and the button pressed is returned in \texttt{button}.

\textbf{§22 Writing a Graphics Driver}
Get location with line (cx, cy, wx, wy, button)

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)

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.

22.3.6. Panels

The following routines cooperate to save and restore sections of the display screen.
Panel_save (name, top, bottom, left, right)

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

The bit display between the rows and cols represented by top, bottom, left, and
cright are saved. The string pointed to by name is a file name which may be
used to save the image.

Panel_restore (name)

cchar *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.

22.4. 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
cfiles, 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
eexample, 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, mrows, array, withzeros, type)

int n;
int mrows;
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 **mrows** 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.
Chapter 23

Writing a Paint Driver

23.1. 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 Pselect program (or the related paint select option). The selected printer is stored in the GRASS environment variable PAINTER. Then the user runs one of the application programs. The principal paint applications that produce color output are Pmap (and the related paint map option) which generates scaled maps, and Pchart (and the related paint chart option) 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.

---

1 See §10.2 GRASS Environment [p. 52].

§23 Writing a Paint Driver
23.2. Creating a Source Directory for the Driver Code
The source code for paint drivers lives in

\$GISBASE/src/paint/Drivers

Each driver has its own sub-directory containing the source code for the executable program, the shell script, and a \textit{Gmakefile} with rules that tell the GRASS \textit{Gmake} command how to compile the driver.\footnote{\textit{Gmake} is the directory where GRASS is installed. See \S 10.1 UNIX Environment (p. 51) for details.}

23.3. The Paint Driver Executable Program
A \textit{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.

23.3.1. Printer I/O Routines
The following routines open the printer port and perform low-level i/o to the printer.

\textbf{\texttt{Popen}} (port) \hspace{1cm} \textit{open the printer port}

\begin{verbatim}
   char *port;

   Open the printer \texttt{port} for output. If the \texttt{port} is a \texttt{tty}, perform any necessary \texttt{tty} settings (baud rate, xon/xoff, etc.) required. No data should be written to the \texttt{port}.

   The \texttt{port} will be the value of the UNIX environment variable MAPLP,\footnote{See \S 11 Compiling GRASS Programs Using \textit{Gmake} (p. 55) for details on the GRASS compilation process.} if set, and NULL otherwise. It is recommended that device drivers use the \texttt{port} that is passed to them so that \textit{paint} has a consistent logic.

   The baud rate should not be hard-coded into \texttt{Popen()}. It should be set in the driver shell as the UNIX environment variable BAUD. \texttt{Popen()} should determine the baud rate from this environment variable.
\end{verbatim}

\footnote{This, and other, environment variables are set in the driver shell script which is described in \S 23.4 The Device Driver Shell Script (p. 222).}
Pout (buf, n)  
unsigned char *buf;
int n;

Output the data in buf. The number of bytes to send is n. This is a low-level request. No processing of the data is to be done. Output is simply to be sent as is to the printer.

It is not required that data passed to this routine go immediately to the printer. This routine can buffer the output, if desired.

It is recommended that this routine be used to send all output to the printer.

Poutc (c)  
unsigned char c;

Send the character c to the printer. This routine can be implemented as follows:

Poutc (c) unsigned char c;
{
    Pout (c, 1);
}

Pouts (s)  
unsigned char *s;

Send 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. Do not close the port.
Pclose ()

Flush any pending output to the printer and close 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.

23.3.2. Initialization

The following routine will be called after Popen(p. 216) to initialize the printer:

Pinit ()

Initialize the printer. Send whatever codes are necessary to get the printer ready for printing.

23.3.3. Alpha-numeric Mode

The following two routines allow the printer to be used for normal text printing:

Palpha ()

Put the printer in alpha-numeric mode. In this mode, the driver should only honor Ptext(p. 218) calls.

Ptext (text)

print text

c char *text;

Print the text string on the printer.

The text will not normally have non-printing 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.

\(^5\) See §23.6 Paint Driver Library (p. 224).
23.3.4. Graphics Mode
The following routines perform raster color graphics:

**Praster()**

Put the printer in raster graphics mode. This implies that subsequent requests will be related to generating color images on the printer.

**Prnpixels(nrows, ncols)**

```c
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., 2x2 pixels) to create more colors, then **ncols** should be set to the number of these larger pixels.\(^6\)

The variable **nrows** should be set to 0. A non-zero value means that the output media does not support arbitrarily long output and **paint** will scale the output to fit into a window **nrows \times 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.

**Ppicotsize(nrows, ncols)**

```c
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 **Prnpixels**(p.219).\(^7\)

There is no limit on the number of rows **nrows** that will be requested. **Paint** assumes that the printer paper is essentially infinite in length. Some printers (e.g., thermal printers like the shinek0635) 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.

\(^6\) The **Pmap** program cannot make use of more than 1024 pixels. It is acceptable for **Prnpixels()** to set **ncols** larger than 1024, but **Pmap** will reset it to 1024. Wide printers will not (currently) be used to their fullest width. When **Pmap** is upgraded, this limitation will disappear.

\(^7\) The programmer should, of course, code defensively. If the number of columns is too large, the driver should exit with an error message.
Pdata (buf, n)  

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.219). 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.

Print (buf, n)  

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.219). 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.

23.3.5. 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.216) has not been called.
Pncolors()  

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 2x2 pixel.8

Poolorlevels(red, green, blue)  

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.

Poolornum(red, green, blue)  

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 1.0.9

The printer color numbers must be in the range 0 to n-1, where n is the number of colors returned by Pncolors(p.221).

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

8 See §23.8 Creating 125 Colors From 3 Colors [p. 227].

9 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.
**Pbxkrvahie (n, red, green, blue)**

```
int n;
floa.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).

---

### 23.4. 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:

- **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.216)`. 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.

- **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.216)` 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 `Pmap` and `paint map` to set the size of the numbers placed on the grid when maps are drawn. The normal value

---

10 The driver shell script may set any other variables that the programmer has determined the driver needs.

---

§23 Writing a Paint Driver
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 Prmap and paint map, as well as the boxes for the printer color chart created by Pchart and paint chart. They have to be determined by trial and error in order to get the numbering to appear under the correct box.\footnote{Apologies are offered for this admittedly awkward design.}

**NBLOCKS**
This positive integer specifies the maximum number of blocks that are to be drawn per line.

**BLOCKSIZE**
This positive integer specifies the number of pixels across the top of an individual box.

**BLOCKSPACE**
This positive integer specifies the number of pixels between boxes.

**TEXTSPACE**
This positive integer specifies the number of space characters to output after each number (printed under the boxes).

**TEXTFUDGE**
This non-negative 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:
23.5. 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:

```c
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.

23.6. Paint Driver Library

The paint system comes with some code that has already be written. This code is in object files under the paint driver library directory. These object files are:

---

12 See §23.7 Compiling the Driver (p. 225) for an example of how to load this library code.
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.216), Pout(p.217), Poute(p.217), Pouts(p.217), Pflush(p.217), and Pclose(p.218) 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.221), Pcolorlevels(p.221), Pcolornum(p.221), and Pcolorvalue(p.222) for the 125 color logic described in §23.8 Creating 125 Colors From 3 Colors [p.227].

23.7. Compiling the Driver

Paint drivers are compiled using the GRASS Gmake utility which requires a Gmakefile containing compilation rules. The following is a sample Gmakefile:

---

13 See §11 Compiling GRASS Programs Using Gmake [p.55] for details on the GRASS compilation process.

§23 Writing a Paint Driver
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. 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. 216), `Pout` (p. 217), `Poutc` (p. 217), `Poutl` (p. 217), `Pflush` (p. 217), and `Pclose` (p. 218) for serial and parallel ports. It also loads `colors125.o` which contains versions of `Pcolors` (p. 221), `Pcolorlevels` (p. 221), `Pcolornum` (p. 221), and `Pcolorvalu` (p. 222) 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. 216) 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

---

14 See also §23.6 `Paint Driver Library` (p. 224).
placed in

$GISBASE/etc/paint/driver\textsuperscript{15}

and the driver shell in

$GISBASE/etc/paint/driver.sh.

23.8. 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 $2 \times 2$ 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):

```c
nPixels = (int) a * 5;
if (nPixels > 4)
nPixels = 4;
```

This logic will agree with the 125 color logic used by the paint driver library\textsuperscript{16} routines $Pcolor(p. 221), Pcolorlevels(p. 221), Pcolornum(p. 221)$, and $Pcolorvalue(p. 222)$, provided that the color numbers are assigned as follows:

```c
colorNumber = redPixels * 25 + greenPixels * 5 + bluePixels;
```

\textsuperscript{15} $GISBASE$ is the directory where GRASS is installed. See §10.1 UNIX Environment \textsuperscript{(p. 51)} for details.

\textsuperscript{16} See §23.6 Paint Driver Library \textsuperscript{(p. 224)}.
Chapter 24

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.

24.1. 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.

24.2. 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, and verify that the variables needed by the shell script are set.

The following accomplishes these two things:

---

1 See §10 Environment Variables [p. 51]
if test "$GISRC" = "" then
echo "Sorry, you are not running GRASS" >&2
exit 1
fi
eval `gisenv`
: ${GISBASE?} ${GISDBASE?} ${LOCATION_NAME?} ${MASPET?}

Note the use of the `:` command. This command simply evaluates its arguments. The first use is as the first character of the file, which signals to UNIX that the script is in fact a Bourne Shell script (see §24.5 Don’t Use `/bin/sh` [p.231]). The second use checks to see that variables are set. The syntax `${GISBASE?}` means that if GISBASE is not set, issue an error message to standard error and exit the shell script.

### 24.3. Gask

The GRASS command `Gask` 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 cell file:

```
Gask old "Select a cell file" cell cell /tmp/$$
.rm -f /tmp/$$
if test "$name" = ""
then
    exit 0
fi
```

The `Gask` 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 cell 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 didn't select any cell file. In this case, something reasonable is done, like

   ```
   rm -f /tmp/$$
   echo "You didn't select a cell file." >&2
   exit 1
   ```

   The other variables will be empty as well.

---

2 The other variables will be empty as well.

§24 Writing GRASS Shell Scripts
24.4. Gfindfile

The Gfindfile command can be used to locate GRASS files that were specified as arguments to the shell script (instead of prompted for with Gask). Assuming that the variable $request contains the name of a cell file, the following checks to see if the file exists. If it does, the variables $name, $mapset and $file will be set to the name, mapset and full UNIX file name for the cell file:

```bash
eval `Gfindfile cell "$request"`
if test "$mapset" = ""
    then
        echo ERROR: cell file "$request" not found >&2
        exit 1
    fi
```

Note. The programmer should use quotes with $request, since it may contain spaces. The user can request a file on the command line of the form "name in mapset" (quotes will preserve the full request). Gfindfile accepts this form and, if found, outputs $name as the name part and $mapset as the mapset part. See the Gfindfile manual entry in the GRASS User's Reference Manual for more details.

24.5. Don’t Use #!/bin/sh

When a user runs a shell script, he or she simply types the name of the shell script just as if it were a compiled program. On systems that have more than one shell, it is the responsibility of UNIX to figure out which shell should interpret the commands in the script. This decision must be made on the basis of the shell for which the script was written.

On systems that have both /bin/sh and /bin/csh, the rule has been: if the first character of the file is #, then the script is given to /bin/csh to interpret; otherwise, it is given to /bin/sh. As the number of shells available grew, the mechanism was expanded to allow the shell script to explicitly specify the interpreter. The rule was modified so that if the first line of the file is:

```
#!/command [args]
```

then the command (with the specified arguments) is invoked as the script interpreter.

This led to /bin/sh scripts starting with #!/bin/sh. However, the authors have found UNIX systems which do not recognize this rule. They simply see the # as the first character, and turn the script over to /bin/csh instead of /bin/sh. Therefore, scripts for

---

3 This form for GRASS file names is discussed under §12.52 Finding Files in the Database
/bin/sh should never start with #. A way to start Bourne Shell script that has worked well on all systems with which the authors have experience, is to use the : command (see §24.2 How a Script Should Start (p.229)).
Appendix A

Annotated Gmake Pre-defined Variables

The pre-defined Gmake variables are defined in the files `makehead` and `make.mid`. These files can be found under `${GISBASE}/src/CMD`.¹

Note. The variables shown here are described in more detail in §11 Compiling GRASS Programs Using Gmake [p. 55].

`makehead`

The `makehead` 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.

¹ `${GISBASE}` is the directory where GRASS is installed. See §10.1 UNIX Environment [p. 51] for details.
## Annotated sample makefile file

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GIS</td>
<td>/grass</td>
<td>GRASS installation directory</td>
</tr>
<tr>
<td>GISDBASE</td>
<td>/grass/data</td>
<td>GRASS database directory</td>
</tr>
<tr>
<td>UNIX_BIN</td>
<td>/usr/local/bin</td>
<td>UNIX command bin directory</td>
</tr>
<tr>
<td>DEFAULT_LOCATION</td>
<td>spearefish</td>
<td>Default Location for new users</td>
</tr>
<tr>
<td>OS</td>
<td>SYSV</td>
<td>Try interface flag</td>
</tr>
<tr>
<td>#OS</td>
<td>BERK</td>
<td>Compiler options</td>
</tr>
<tr>
<td>COMPILE_FLAGS</td>
<td>-O</td>
<td>Loader options</td>
</tr>
<tr>
<td>LDFLAGS</td>
<td>-s</td>
<td>Digitizer compile time flag</td>
</tr>
<tr>
<td>DIGIT_FLAGS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>#DIGIT_FLAGS</td>
<td>-DATT</td>
<td>Math library</td>
</tr>
<tr>
<td>#DIGIT_FLAGS</td>
<td>-DMASSCOMP</td>
<td>Termlib/termcap library</td>
</tr>
<tr>
<td>MATHLIB</td>
<td>-lm</td>
<td>Can use termlib to clear screen</td>
</tr>
<tr>
<td>TERMLIB</td>
<td>-lttermiib</td>
<td></td>
</tr>
<tr>
<td>CLEAR</td>
<td>ok</td>
<td>Library archive rule for systems with ranlib</td>
</tr>
<tr>
<td>#CLEAR</td>
<td>no</td>
<td>Library archive rule for systems without ranlib</td>
</tr>
<tr>
<td>AR</td>
<td>ar run $@ $?;\</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ranlib $@</td>
<td></td>
</tr>
<tr>
<td>#AR</td>
<td>ar rc $@\</td>
<td></td>
</tr>
<tr>
<td></td>
<td>'lorder $(OBJ)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>tsort'</td>
</tr>
</tbody>
</table>
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.

Annotated `make.mid` file

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CFLAGS</td>
<td><code>=$(COMPILE_FLAGS) -$(LIBDIR) -D$(OS) $(EXTRA_CFLAGS)</code></td>
<td>Compiler flags</td>
</tr>
<tr>
<td>GMAKE</td>
<td><code>=$(GIS)/src/CMD/Gmake</code></td>
<td>Gmake command</td>
</tr>
<tr>
<td>MAKEALL</td>
<td>set - *; for d do\</td>
<td>Gmake &quot;all&quot;</td>
</tr>
<tr>
<td></td>
<td>test -f $d/Makefile &amp;&amp; $(GMAKE) $$d;\</td>
<td></td>
</tr>
<tr>
<td></td>
<td>done; exit 0</td>
<td></td>
</tr>
<tr>
<td>MANROFF</td>
<td>tbl -TX \</td>
<td>Nroff rule for manual pages</td>
</tr>
<tr>
<td></td>
<td>$(GIS)/src/man/help/man.header $?\</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>nroff -Tlp</td>
</tr>
<tr>
<td>CURSES</td>
<td><code>= -lcurses $(TERMLIB)</code></td>
<td>Curses libraries</td>
</tr>
<tr>
<td>MAN1</td>
<td><code>=$(GIS)/man/1</code></td>
<td>Man directory, section 1</td>
</tr>
<tr>
<td>MAN2</td>
<td><code>=$(GIS)/man/2</code></td>
<td>Man directory, section 2</td>
</tr>
<tr>
<td>HELP</td>
<td><code>=$(GIS)/man/help</code></td>
<td>Help directory</td>
</tr>
<tr>
<td>BIN</td>
<td><code>=$(GIS)/bin</code></td>
<td>GRASS command directory</td>
</tr>
<tr>
<td>ETC</td>
<td><code>=$(GIS)/etc</code></td>
<td>GRASS command support directory</td>
</tr>
<tr>
<td>SRC</td>
<td><code>=$(GIS)/src</code></td>
<td>GRASS source directory</td>
</tr>
<tr>
<td>LIBDIR</td>
<td><code>=$(GIS)/src/libes</code></td>
<td>GRASS library directory</td>
</tr>
<tr>
<td>GISLIB</td>
<td><code>=$(LIBDIR)/libgis.a</code></td>
<td>GIS library</td>
</tr>
<tr>
<td>IMAGERYLIB</td>
<td><code>=$(LIBDIR)/libim.a</code></td>
<td>Imagery library</td>
</tr>
<tr>
<td>LOCKLIB</td>
<td><code>=$(LIBDIR)/liblock.a</code></td>
<td>Lock library</td>
</tr>
<tr>
<td>SEGMENTLIB</td>
<td><code>=$(LIBDIR)/libsegment.a</code></td>
<td>Segment library</td>
</tr>
<tr>
<td>DLGGLIB</td>
<td><code>=$(LIBDIR)/libdlg.a</code></td>
<td>Dlg library</td>
</tr>
<tr>
<td>RASTERLIB</td>
<td><code>=$(SRC)/D/libes/rasterlib.a</code></td>
<td>Raster library</td>
</tr>
<tr>
<td>DISPLAYLIB</td>
<td><code>=$(SRC)/D/libes/displaylib.a</code></td>
<td>Display library</td>
</tr>
<tr>
<td>VASKLIB</td>
<td><code>=$(LIBDIR)/libvask.a</code></td>
<td>Vask library</td>
</tr>
<tr>
<td>VASK</td>
<td><code>=$(VASKLIB) $(CURSES)</code></td>
<td>Vask + curses library</td>
</tr>
</tbody>
</table>

Annotated Gmake Pre-defined Variables
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 cell file data or category codes (which are CELL values as well) as type CELL.

Under GRASS 3.0 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:

```c
CELL c; // cell 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:

```c
char userbuf[128];
CELL c;
long x;

printf("Which category?\n"); // 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 don't 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.
## Appendix C

### Index to GIS Library

Here is an index of GIS Library routines, with calling sequences and short function descriptions.

<table>
<thead>
<tr>
<th>routine</th>
<th>parameters</th>
<th>description</th>
<th>page</th>
</tr>
</thead>
<tbody>
<tr>
<td>G_allocate_cell_buf</td>
<td>()</td>
<td>allocate a cell buffer</td>
<td>86</td>
</tr>
<tr>
<td>G_ask_any</td>
<td>(prompt, name, element, label, warn)</td>
<td>prompt for any valid file name</td>
<td>70</td>
</tr>
<tr>
<td>G_ask_cell_in_mapset</td>
<td>(prompt, name)</td>
<td>prompt for existing cell file</td>
<td>81</td>
</tr>
<tr>
<td>G_ask_cell_new</td>
<td>(prompt, name)</td>
<td>prompt for new cell file</td>
<td>81</td>
</tr>
<tr>
<td>G_ask_cell_old</td>
<td>(prompt, name)</td>
<td>prompt for existing cell file</td>
<td>81</td>
</tr>
<tr>
<td>G_ask_in_mapset</td>
<td>(prompt, name, element, label)</td>
<td>prompt for existing database file</td>
<td>70</td>
</tr>
<tr>
<td>G_ask_new</td>
<td>(prompt, name, element, label)</td>
<td>prompt for new database file</td>
<td>69</td>
</tr>
<tr>
<td>G_ask_old</td>
<td>(prompt, name, element, label)</td>
<td>prompt for existing database file</td>
<td>69</td>
</tr>
<tr>
<td>G_ask_sites_in_mapset</td>
<td>(prompt, name)</td>
<td>prompt for existing site list file</td>
<td>106</td>
</tr>
<tr>
<td>G_ask_sites_new</td>
<td>(prompt, name)</td>
<td>prompt for new site list file</td>
<td>106</td>
</tr>
<tr>
<td>G_ask_sites_old</td>
<td>(prompt, name)</td>
<td>prompt for existing site list file</td>
<td>106</td>
</tr>
<tr>
<td>G_ask_vector_in_mapset</td>
<td>(prompt, name)</td>
<td>prompt for an existing vector file</td>
<td>101</td>
</tr>
<tr>
<td>G_ask_vector_new</td>
<td>(prompt, name)</td>
<td>prompt for a new vector file</td>
<td>101</td>
</tr>
<tr>
<td>G_ask_vector_old</td>
<td>(prompt, name)</td>
<td>prompt for an existing vector file</td>
<td>101</td>
</tr>
<tr>
<td>G_calloc</td>
<td>(n, size)</td>
<td>memory allocation</td>
<td>76</td>
</tr>
<tr>
<td>G_close_cell</td>
<td>(fd)</td>
<td>close a cell file</td>
<td>89</td>
</tr>
<tr>
<td>G_date</td>
<td>()</td>
<td>current date and time</td>
<td>117</td>
</tr>
<tr>
<td>G_fatal_error</td>
<td>(message)</td>
<td>print error message and exit</td>
<td>64</td>
</tr>
<tr>
<td>G_find_cell2</td>
<td>(name, mapset)</td>
<td>find a cell file</td>
<td>82</td>
</tr>
<tr>
<td>G_find_cell</td>
<td>(name, mapset)</td>
<td>find a cell file</td>
<td>82</td>
</tr>
<tr>
<td>G_find_file2</td>
<td>(element, name, mapset)</td>
<td>find a database file</td>
<td>71</td>
</tr>
<tr>
<td>G_find_file</td>
<td>(element, name, mapset)</td>
<td>find a database file</td>
<td>71</td>
</tr>
<tr>
<td>G_find_vector2</td>
<td>(name, mapset)</td>
<td>find a vector file</td>
<td>102</td>
</tr>
<tr>
<td>G_find_vector</td>
<td>(name, mapset)</td>
<td>find a vector file</td>
<td>102</td>
</tr>
<tr>
<td>G_fopen_append</td>
<td>(element, name)</td>
<td>open a database file for update</td>
<td>73</td>
</tr>
<tr>
<td>G_fopen_new</td>
<td>(element, name)</td>
<td>open a new database file</td>
<td>74</td>
</tr>
<tr>
<td>G_fopen_old</td>
<td>(element, name, mapset)</td>
<td>open a database file for reading</td>
<td>73</td>
</tr>
<tr>
<td>G_fopen_sites_new</td>
<td>(name)</td>
<td>open a new site list file</td>
<td>107</td>
</tr>
<tr>
<td>G_fopen_sites_old</td>
<td>(name, mapset)</td>
<td>open an existing site list file</td>
<td>107</td>
</tr>
<tr>
<td>G_fopen_vector_new</td>
<td>(name)</td>
<td>open a new vector file</td>
<td>104</td>
</tr>
<tr>
<td>G_fopen_vector_old</td>
<td>(name, mapset)</td>
<td>open an existing vector file</td>
<td>103</td>
</tr>
<tr>
<td>G_fork</td>
<td>()</td>
<td>create a protected child process</td>
<td>116</td>
</tr>
<tr>
<td>G_free_cats</td>
<td>(cats)</td>
<td>free category structure memory</td>
<td>94</td>
</tr>
<tr>
<td>G_free_colors</td>
<td>(colors)</td>
<td>free color structure memory</td>
<td>96</td>
</tr>
<tr>
<td>G_getask_return_msg</td>
<td>()</td>
<td>get Hit RETURN msg</td>
<td>70</td>
</tr>
<tr>
<td>G_get_cat</td>
<td>(n, cats)</td>
<td>get a category label</td>
<td>93</td>
</tr>
<tr>
<td>G_get_cats_title</td>
<td>(cats)</td>
<td>get title from category structure</td>
<td>93</td>
</tr>
<tr>
<td>G_get_cellhd</td>
<td>(name, mapset, cellhd)</td>
<td>read the cell header</td>
<td>90</td>
</tr>
<tr>
<td>G_get_celltitle</td>
<td>(name, mapset)</td>
<td>get cell title</td>
<td>92</td>
</tr>
<tr>
<td>G_get_color</td>
<td>(cat, red, green, blue, colors)</td>
<td>get a category color</td>
<td>95</td>
</tr>
<tr>
<td>G_get_default_window</td>
<td>(window)</td>
<td>read the default window</td>
<td>78</td>
</tr>
</tbody>
</table>
## GIS Library

<table>
<thead>
<tr>
<th>routine</th>
<th>parameters</th>
<th>description</th>
<th>page</th>
</tr>
</thead>
<tbody>
<tr>
<td>G getenv</td>
<td>(name)</td>
<td>query GRASS environment variable</td>
<td>67</td>
</tr>
<tr>
<td>G getenv</td>
<td>(name)</td>
<td>query GRASS environment variable</td>
<td>67</td>
</tr>
<tr>
<td>G get_map_row</td>
<td>(fd, cell, row)</td>
<td>read a cell file</td>
<td>87</td>
</tr>
<tr>
<td>G get_map_row_nomask</td>
<td>(fd, cell, row)</td>
<td>read a cell file (without masking)</td>
<td>87</td>
</tr>
<tr>
<td>G gets</td>
<td>(buf)</td>
<td>get a line of input (detect ctrl-z)</td>
<td>117</td>
</tr>
<tr>
<td>G get_set_window</td>
<td>(window)</td>
<td>get the active window</td>
<td>79</td>
</tr>
<tr>
<td>G get site</td>
<td>(fd, east, north, desc)</td>
<td>read site list file</td>
<td>107</td>
</tr>
<tr>
<td>G get_window</td>
<td>(window)</td>
<td>read the database window</td>
<td>77</td>
</tr>
<tr>
<td>G getenv</td>
<td></td>
<td>top level program directory</td>
<td>66</td>
</tr>
<tr>
<td>G getenv</td>
<td></td>
<td>top level database window</td>
<td>67</td>
</tr>
<tr>
<td>G get/site</td>
<td></td>
<td>initialize gis library</td>
<td>64</td>
</tr>
<tr>
<td>G get/site</td>
<td></td>
<td>user's home directory</td>
<td>117</td>
</tr>
<tr>
<td>G gets</td>
<td></td>
<td>initialize category structure</td>
<td>93</td>
</tr>
<tr>
<td>G gets</td>
<td></td>
<td>initialize color structure</td>
<td>96</td>
</tr>
<tr>
<td>G get/site</td>
<td></td>
<td>initialize range structure</td>
<td>100</td>
</tr>
<tr>
<td>G get/site</td>
<td></td>
<td>return interrupt char</td>
<td>91</td>
</tr>
<tr>
<td>G is_reclass</td>
<td>(name, mapset, r_name, r_mapset)</td>
<td>check for legal database file names</td>
<td>72</td>
</tr>
<tr>
<td>G legal_filename</td>
<td>(name)</td>
<td>current location name</td>
<td>66</td>
</tr>
<tr>
<td>G legal_filename</td>
<td>(name)</td>
<td>current location directory</td>
<td>67</td>
</tr>
<tr>
<td>G malloc</td>
<td>(size)</td>
<td>make aspect colors</td>
<td>97</td>
</tr>
<tr>
<td>G malloc</td>
<td>(size)</td>
<td>make color ramp</td>
<td>97</td>
</tr>
<tr>
<td>G malloc</td>
<td>(size)</td>
<td>make color wave</td>
<td>97</td>
</tr>
<tr>
<td>G malloc</td>
<td>(size)</td>
<td>make linear grey scale</td>
<td>97</td>
</tr>
<tr>
<td>G malloc</td>
<td>(size)</td>
<td>make rainbow colors</td>
<td>97</td>
</tr>
<tr>
<td>G malloc</td>
<td>(size)</td>
<td>make random colors</td>
<td>98</td>
</tr>
<tr>
<td>G malloc</td>
<td>(size)</td>
<td>make red, yellow, green colors</td>
<td>98</td>
</tr>
<tr>
<td>G malloc</td>
<td>(size)</td>
<td>memory allocation</td>
<td>76</td>
</tr>
<tr>
<td>G malloc</td>
<td>(size)</td>
<td>current mapset name</td>
<td>66</td>
</tr>
<tr>
<td>G malloc</td>
<td>(size)</td>
<td>location title</td>
<td>66</td>
</tr>
<tr>
<td>G malloc</td>
<td>(size)</td>
<td>open a new cell file (sequential)</td>
<td>84</td>
</tr>
<tr>
<td>G malloc</td>
<td>(size)</td>
<td>open a new cell file (random)</td>
<td>85</td>
</tr>
<tr>
<td>G malloc</td>
<td>(size)</td>
<td>open a new cell file (uncompressed)</td>
<td>85</td>
</tr>
<tr>
<td>G malloc</td>
<td>(size)</td>
<td>open an existing cell file</td>
<td>83</td>
</tr>
<tr>
<td>G malloc</td>
<td>(size)</td>
<td>open a new database file</td>
<td>74</td>
</tr>
<tr>
<td>G malloc</td>
<td>(size)</td>
<td>open a database file for reading</td>
<td>72</td>
</tr>
<tr>
<td>G malloc</td>
<td>(size)</td>
<td>open a database file for update</td>
<td>73</td>
</tr>
<tr>
<td>G malloc</td>
<td>(size)</td>
<td>parse command line</td>
<td>109</td>
</tr>
<tr>
<td>G malloc</td>
<td>(size)</td>
<td>command line usage message</td>
<td>111</td>
</tr>
<tr>
<td>G malloc</td>
<td>(size)</td>
<td>print percent complete messages</td>
<td>117</td>
</tr>
<tr>
<td>G malloc</td>
<td>(size)</td>
<td>return program name</td>
<td>118</td>
</tr>
<tr>
<td>G malloc</td>
<td>(size)</td>
<td>query cartographic projection</td>
<td>80</td>
</tr>
<tr>
<td>G malloc</td>
<td>(size)</td>
<td>query cartographic projection</td>
<td>80</td>
</tr>
<tr>
<td>G malloc</td>
<td>(size)</td>
<td>write the cell header</td>
<td>90</td>
</tr>
<tr>
<td>G malloc</td>
<td>(size)</td>
<td>change cell title</td>
<td>92</td>
</tr>
<tr>
<td>G malloc</td>
<td>(size)</td>
<td>write a cell file (sequential)</td>
<td>88</td>
</tr>
<tr>
<td>G malloc</td>
<td>(size)</td>
<td>write a cell file (random)</td>
<td>88</td>
</tr>
<tr>
<td>G malloc</td>
<td>(size)</td>
<td>write site list file</td>
<td>108</td>
</tr>
<tr>
<td>G malloc</td>
<td>(size)</td>
<td>write the database window</td>
<td>77</td>
</tr>
<tr>
<td>G malloc</td>
<td>(size)</td>
<td>read cell category file</td>
<td>91</td>
</tr>
<tr>
<td>G malloc</td>
<td>(size)</td>
<td>read map layer color table</td>
<td>94</td>
</tr>
</tbody>
</table>

## Index to GIS Library
<table>
<thead>
<tr>
<th>Routine</th>
<th>Parameters</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>G_read_history</td>
<td>(name, mapset, history)</td>
<td>read cell history file</td>
<td>98</td>
</tr>
<tr>
<td>G_read_range</td>
<td>(name, mapset, range)</td>
<td>read cell range</td>
<td>99</td>
</tr>
<tr>
<td>G_read_vector_cats</td>
<td>(name, mapset, cats)</td>
<td>read vector category file</td>
<td>105</td>
</tr>
<tr>
<td>G_realloc</td>
<td>(ptr, size)</td>
<td>memory allocation</td>
<td>76</td>
</tr>
<tr>
<td>G_remove</td>
<td>(element, name)</td>
<td>remove a database file</td>
<td>75</td>
</tr>
<tr>
<td>G_rename</td>
<td>(element, old, new)</td>
<td>rename a database file</td>
<td>75</td>
</tr>
<tr>
<td>G_row_update_range</td>
<td>(cell, n, range)</td>
<td>update range structure</td>
<td>100</td>
</tr>
<tr>
<td>G_set_9t9tntnng</td>
<td>(msg)</td>
<td>set Hit RETURN msg</td>
<td>70</td>
</tr>
<tr>
<td>G_set_cat</td>
<td>(n, label, cats)</td>
<td>set a category label</td>
<td>94</td>
</tr>
<tr>
<td>G_set_cats_title</td>
<td>(title, cats)</td>
<td>set title in category structure</td>
<td>94</td>
</tr>
<tr>
<td>G_set_color</td>
<td>(cat, red, green, blue, colors)</td>
<td>set a category color</td>
<td>96</td>
</tr>
<tr>
<td>G_setenv</td>
<td>(name, value)</td>
<td>set GRASS environment variable</td>
<td>67</td>
</tr>
<tr>
<td>G_setenv</td>
<td>(name, value)</td>
<td>set GRASS environment variable</td>
<td>67</td>
</tr>
<tr>
<td>G_set_error_routine</td>
<td>(handler)</td>
<td>change error handling</td>
<td>65</td>
</tr>
<tr>
<td>G_set_window</td>
<td>(window)</td>
<td>set the active window</td>
<td>79</td>
</tr>
<tr>
<td>G_short_history</td>
<td>(name, type, history)</td>
<td>initialize history structure</td>
<td>99</td>
</tr>
<tr>
<td>G_sleep_on_error</td>
<td>(flag)</td>
<td>sleep on error?</td>
<td>65</td>
</tr>
<tr>
<td>G_squeeze</td>
<td>(s)</td>
<td>remove unnecessary white space</td>
<td>113</td>
</tr>
<tr>
<td>G_store</td>
<td>(s)</td>
<td>copy string to allocated memory</td>
<td>114</td>
</tr>
<tr>
<td>G_strcat</td>
<td>(dst, src)</td>
<td>concatenate strings</td>
<td>113</td>
</tr>
<tr>
<td>G_strcpy</td>
<td>(dst, src)</td>
<td>copy strings</td>
<td>113</td>
</tr>
<tr>
<td>G_strip</td>
<td>(s)</td>
<td>remove leading/trailing white space</td>
<td>114</td>
</tr>
<tr>
<td>G_strncpy</td>
<td>(dst, src, n)</td>
<td>copy strings</td>
<td>114</td>
</tr>
<tr>
<td>GSuppress_warnings</td>
<td>(flag)</td>
<td>suppress warnings?</td>
<td>65</td>
</tr>
<tr>
<td>G_system</td>
<td>(command)</td>
<td>run a shell level command</td>
<td>116</td>
</tr>
<tr>
<td>G_tempfile</td>
<td>()</td>
<td>returns a temporary file name</td>
<td>108</td>
</tr>
<tr>
<td>G_titlecase</td>
<td>(s)</td>
<td>convert string to lower case</td>
<td>114</td>
</tr>
<tr>
<td>G_toupper</td>
<td>(s)</td>
<td>convert string to upper case</td>
<td>114</td>
</tr>
<tr>
<td>G_unctrl</td>
<td>(c)</td>
<td>printable version of control character</td>
<td>114</td>
</tr>
<tr>
<td>G_unopen_cell</td>
<td>(fd)</td>
<td>unopen a cell file</td>
<td>89</td>
</tr>
<tr>
<td>G_unset_error_routine</td>
<td>()</td>
<td>reset normal error handling</td>
<td>66</td>
</tr>
<tr>
<td>G_update_range</td>
<td>(cat, range)</td>
<td>update range structure</td>
<td>100</td>
</tr>
<tr>
<td>G_warning</td>
<td>(message)</td>
<td>print warning message and continue</td>
<td>64</td>
</tr>
<tr>
<td>G_whoami</td>
<td>()</td>
<td>user’s name</td>
<td>118</td>
</tr>
<tr>
<td>G_window_cols</td>
<td>()</td>
<td>number of columns in active window</td>
<td>78</td>
</tr>
<tr>
<td>G_window_rows</td>
<td>()</td>
<td>number of rows in active window</td>
<td>78</td>
</tr>
<tr>
<td>G_write_cats</td>
<td>(name, cats)</td>
<td>write cell category file</td>
<td>92</td>
</tr>
<tr>
<td>G_write_colors</td>
<td>(name, mapset, colors)</td>
<td>write map layer color table</td>
<td>95</td>
</tr>
<tr>
<td>G_write_history</td>
<td>(name, history)</td>
<td>write cell history file</td>
<td>99</td>
</tr>
<tr>
<td>G_write_range</td>
<td>(name, range)</td>
<td>write cell range</td>
<td>100</td>
</tr>
<tr>
<td>G_write_vector_cats</td>
<td>(name, cats)</td>
<td>write vector category file</td>
<td>105</td>
</tr>
<tr>
<td>G_yes</td>
<td>(question, default)</td>
<td>ask a yes/no question</td>
<td>118</td>
</tr>
<tr>
<td>G_zero_cell_buf</td>
<td>(buf)</td>
<td>zero a cell buffer</td>
<td>86</td>
</tr>
<tr>
<td>G_zone</td>
<td>()</td>
<td>query cartographic zone</td>
<td>80</td>
</tr>
</tbody>
</table>
## Appendix D

### Index to Dig Library

Here is an index of Dig Library routines, with calling sequences and short function descriptions.

### Dig Library

<table>
<thead>
<tr>
<th>Routine</th>
<th>Parameters</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>dig_bound_box</code></td>
<td><code>(p, N, S, E, W)</code></td>
<td>get arc bounding box</td>
<td>135</td>
</tr>
<tr>
<td><code>dig_check_dist</code></td>
<td><code>(map, n, x, y, d)</code></td>
<td>distance to arc</td>
<td>132</td>
</tr>
<tr>
<td><code>dig_distance2_point_to_line</code></td>
<td><code>(x, y, x1, y1, x2, y2)</code></td>
<td>distance to line-segment</td>
<td>134</td>
</tr>
<tr>
<td><code>dig_fini</code></td>
<td><code>(fd)</code></td>
<td>end level one vector access</td>
<td>125</td>
</tr>
<tr>
<td><code>dig_init_box</code></td>
<td><code>(N, S, E, W)</code></td>
<td>limit arc search in box</td>
<td>127</td>
</tr>
<tr>
<td><code>dig_init</code></td>
<td><code>(fd)</code></td>
<td>initialize level one vector access</td>
<td>125</td>
</tr>
<tr>
<td><code>dig_P_area_att</code></td>
<td><code>(map, n)</code></td>
<td>get area category attribute</td>
<td>130</td>
</tr>
<tr>
<td><code>dig_P_fill</code></td>
<td><code>(map)</code></td>
<td>end level two vector access</td>
<td>128</td>
</tr>
<tr>
<td><code>dig_P_get_area_bbox</code></td>
<td><code>(map, n, N, S, E, W)</code></td>
<td>get area bounding box</td>
<td>130</td>
</tr>
<tr>
<td><code>dig_P_get_area</code></td>
<td><code>(map, n, pa)</code></td>
<td>get area polygon</td>
<td>129</td>
</tr>
<tr>
<td><code>dig_P_get_area_xxy</code></td>
<td><code>(map, np, x, y)</code></td>
<td>get area polygon</td>
<td>129</td>
</tr>
<tr>
<td><code>dig_P_get_line_bbox</code></td>
<td><code>(map, n, N, S, E, W)</code></td>
<td>get arc bounding box</td>
<td>131</td>
</tr>
<tr>
<td><code>dig_P_init</code></td>
<td><code>(name, mapset, map)</code></td>
<td>initialize level two vector access</td>
<td>128</td>
</tr>
<tr>
<td><code>dig_P_num_areas</code></td>
<td><code>(map)</code></td>
<td>get arc category attribute</td>
<td>131</td>
</tr>
<tr>
<td><code>dig_P_num_lines</code></td>
<td><code>(map)</code></td>
<td>get number of areas</td>
<td>130</td>
</tr>
<tr>
<td><code>dig_point_in_area</code></td>
<td><code>(map, x, y, pa)</code></td>
<td>point in area</td>
<td>132</td>
</tr>
<tr>
<td><code>dig_point_to_area</code></td>
<td><code>(map, x, y)</code></td>
<td>find area with point</td>
<td>132</td>
</tr>
<tr>
<td><code>dig_point_to_line</code></td>
<td><code>(map, x, y, type)</code></td>
<td>find arc with point</td>
<td>132</td>
</tr>
<tr>
<td><code>dig_P_read_line</code></td>
<td><code>(map, n, p)</code></td>
<td>read arc</td>
<td>130</td>
</tr>
<tr>
<td><code>dig_P_read_next_line</code></td>
<td><code>(map, p)</code></td>
<td>read next arc</td>
<td>131</td>
</tr>
<tr>
<td><code>dig_P_rewind</code></td>
<td><code>(map)</code></td>
<td>rewind next-arc pointer</td>
<td>131</td>
</tr>
<tr>
<td><code>dig_print_header</code></td>
<td><code>()</code></td>
<td>display vector header information</td>
<td>125</td>
</tr>
<tr>
<td><code>dig_prune</code></td>
<td><code>(p, threshold)</code></td>
<td>prune a dense arc</td>
<td>136</td>
</tr>
<tr>
<td><code>dig_P_tmp_close</code></td>
<td><code>(map)</code></td>
<td>temporary close vector map</td>
<td>128</td>
</tr>
<tr>
<td><code>dig_P_tmp_open</code></td>
<td><code>(map)</code></td>
<td>reopen closed vector map</td>
<td>128</td>
</tr>
<tr>
<td><code>dig_read_header</code></td>
<td><code>(fd, header)</code></td>
<td>read vector header</td>
<td>134</td>
</tr>
<tr>
<td><code>dig_read_line</code></td>
<td><code>(fd, offset, x, y, np)</code></td>
<td>read arc</td>
<td>133</td>
</tr>
<tr>
<td><code>dig_read_line_in_box</code></td>
<td><code>(fd, np, x, y)</code></td>
<td>read arc in box</td>
<td>127</td>
</tr>
<tr>
<td><code>dig_read_next_line</code></td>
<td><code>(fd, np, x, y)</code></td>
<td>get next arc</td>
<td>126</td>
</tr>
<tr>
<td><code>dig_read_next_line_type</code></td>
<td><code>(fd, np, x, y, type)</code></td>
<td>get next arc by type</td>
<td>126</td>
</tr>
<tr>
<td><code>dig_rewrite</code></td>
<td><code>(fd)</code></td>
<td>rewind vector file</td>
<td>125</td>
</tr>
<tr>
<td><code>dig_write_header</code></td>
<td><code>(fd, header)</code></td>
<td>write vector header</td>
<td>134</td>
</tr>
<tr>
<td><code>dig_write_line</code></td>
<td><code>(fd, type, x, y, np)</code></td>
<td>write arc</td>
<td>133</td>
</tr>
<tr>
<td><code>dig_xy_distance2_point_to_line</code></td>
<td><code>(x, y, x1, y1, x2, y2)</code></td>
<td>distance to line-segment</td>
<td>135</td>
</tr>
</tbody>
</table>
**Appendix E**

**Index to Imagery Library**

Here is an index of Dig Imagery routines, with calling sequences and short function descriptions.

<table>
<thead>
<tr>
<th>Routine</th>
<th>Parameters</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>dig_bound_box</td>
<td>(p, N, S, E, W)</td>
<td>get arc bounding box</td>
<td>135</td>
</tr>
<tr>
<td>dig_check_dist</td>
<td>(map, n, x, y, d)</td>
<td>distance to arc</td>
<td>132</td>
</tr>
<tr>
<td>dig_distance2_point_to_line</td>
<td>(x, y, x1, y1, x2, y2)</td>
<td>distance to line-segment</td>
<td>134</td>
</tr>
<tr>
<td>dig_fin</td>
<td>(fd)</td>
<td>end level one vector access</td>
<td>125</td>
</tr>
<tr>
<td>dig_init_box</td>
<td>(N, S, E, W)</td>
<td>limit arc search in box</td>
<td>127</td>
</tr>
<tr>
<td>dig_init</td>
<td>(fd)</td>
<td>initialize level one vector access</td>
<td>125</td>
</tr>
<tr>
<td>dig_P_area_att</td>
<td>(map, n)</td>
<td>get area category attribute</td>
<td>130</td>
</tr>
<tr>
<td>dig_P_fin</td>
<td>(map)</td>
<td>end level two vector access</td>
<td>128</td>
</tr>
<tr>
<td>dig_P_get_area_bbox</td>
<td>(map, n, N, S, E, W)</td>
<td>get area bounding box</td>
<td>130</td>
</tr>
<tr>
<td>dig_P_get_area_polygon</td>
<td>(map, n, pa)</td>
<td>get area polygon</td>
<td>129</td>
</tr>
<tr>
<td>dig_P_get_area_xy</td>
<td>(map, n, x, y)</td>
<td>get area polygon</td>
<td>129</td>
</tr>
<tr>
<td>dig_P_get_line_bbox</td>
<td>(map, n, N, S, E, W)</td>
<td>get arc bounding box</td>
<td>131</td>
</tr>
<tr>
<td>dig_P_init</td>
<td>(name, mapset, map)</td>
<td>initialize level two vector access</td>
<td>128</td>
</tr>
<tr>
<td>dig_P_line_att</td>
<td>(map, n)</td>
<td>get arc category attribute</td>
<td>131</td>
</tr>
<tr>
<td>dig_P_num_areas</td>
<td>(map)</td>
<td>get number of areas</td>
<td>129</td>
</tr>
<tr>
<td>dig_P_num_lines</td>
<td>(map)</td>
<td>get number of arcs</td>
<td>130</td>
</tr>
<tr>
<td>dig_point_in_area</td>
<td>(map, x, y, pa)</td>
<td>point in area</td>
<td>132</td>
</tr>
<tr>
<td>dig_point_to_area</td>
<td>(map, x, y)</td>
<td>find area with point</td>
<td>132</td>
</tr>
<tr>
<td>dig_point_to_line</td>
<td>(map, x, y, type)</td>
<td>find arc with point</td>
<td>132</td>
</tr>
<tr>
<td>dig_P_read_line</td>
<td>(map, n, p)</td>
<td>read arc</td>
<td>130</td>
</tr>
<tr>
<td>dig_P_read_next_line</td>
<td>(map, p)</td>
<td>read next arc</td>
<td>131</td>
</tr>
<tr>
<td>dig_P_rewind</td>
<td>(map)</td>
<td>rewind next arc pointer</td>
<td>131</td>
</tr>
<tr>
<td>dig_print_header</td>
<td>()</td>
<td>display vector header information</td>
<td>125</td>
</tr>
<tr>
<td>dig_prune</td>
<td>(p, threshold)</td>
<td>prune a dense arc</td>
<td>135</td>
</tr>
<tr>
<td>dig_P_tmp_close</td>
<td>(map)</td>
<td>temporary close vector map</td>
<td>128</td>
</tr>
<tr>
<td>dig_P_tmp_open</td>
<td>(map)</td>
<td>reopen closed vector map</td>
<td>128</td>
</tr>
<tr>
<td>dig_read_header_binary</td>
<td>(fd, header)</td>
<td>read vector header</td>
<td>134</td>
</tr>
<tr>
<td>dig_read_line</td>
<td>(fd, offset, x, y, np)</td>
<td>read arc</td>
<td>133</td>
</tr>
<tr>
<td>dig_read_line_in_box</td>
<td>(fd, np, x, y)</td>
<td>read arc in box</td>
<td>127</td>
</tr>
<tr>
<td>dig_read_next_line</td>
<td>(fd, np, x, y)</td>
<td>get next arc</td>
<td>126</td>
</tr>
<tr>
<td>dig_read_next_line_type</td>
<td>(fd, np, x, y, type)</td>
<td>get next arc by type</td>
<td>126</td>
</tr>
<tr>
<td>dig_rewind</td>
<td>(fd)</td>
<td>rewind vector file</td>
<td>125</td>
</tr>
<tr>
<td>dig_write_header_binary</td>
<td>(fd, header)</td>
<td>write vector header</td>
<td>134</td>
</tr>
<tr>
<td>dig_write_line</td>
<td>(fd, type, x, y, np)</td>
<td>write arc</td>
<td>133</td>
</tr>
<tr>
<td>dig_xy_distance2_point_to_line</td>
<td>(x, y, x1, y1, x2, y2)</td>
<td>distance to line-segment</td>
<td>135</td>
</tr>
</tbody>
</table>
**Appendix F**

**Index to Display Graphics Library**

Here is an index of Display Graphics Library routines, with calling sequences and short function descriptions.

<table>
<thead>
<tr>
<th>Routine</th>
<th>Parameters</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>D_add_to_list</td>
<td>(string)</td>
<td>Add command to window display list</td>
<td>161</td>
</tr>
<tr>
<td>D_a_to_d_col</td>
<td>(column)</td>
<td>Array to screen (column)</td>
<td>183</td>
</tr>
<tr>
<td>D_a_to_d_row</td>
<td>(row)</td>
<td>Array to screen (row)</td>
<td>183</td>
</tr>
<tr>
<td>D_cell_draw_setup</td>
<td>(top, bottom, left, right)</td>
<td>Prepare for raster graphics</td>
<td>165</td>
</tr>
<tr>
<td>D_check_map_window</td>
<td>(window)</td>
<td>Assign/retrieve current map window</td>
<td>160</td>
</tr>
<tr>
<td>D_clear_window</td>
<td>()</td>
<td>Clear window display lists</td>
<td>161</td>
</tr>
<tr>
<td>D_clear_window</td>
<td>()</td>
<td>Clear information about current window</td>
<td>161</td>
</tr>
<tr>
<td>D_clip</td>
<td>(s, n, w, e, x, y, c_x, c_y)</td>
<td>Clip coordinates to window</td>
<td>166</td>
</tr>
<tr>
<td>D_do_conversions</td>
<td>(window, top, bottom, left, right)</td>
<td>Initialize conversions</td>
<td>162</td>
</tr>
<tr>
<td>D_draw_cell_row</td>
<td>(row, raster)</td>
<td>Render a raster row</td>
<td>165</td>
</tr>
<tr>
<td>D_d_to_a_col</td>
<td>(x)</td>
<td>Screen to array (x)</td>
<td>164</td>
</tr>
<tr>
<td>D_d_to_a_row</td>
<td>(y)</td>
<td>Screen to array (y)</td>
<td>164</td>
</tr>
<tr>
<td>D_d_to_u_col</td>
<td>(x)</td>
<td>Screen to earth (x)</td>
<td>164</td>
</tr>
<tr>
<td>D_d_to_u_row</td>
<td>(y)</td>
<td>Screen to earth (y)</td>
<td>164</td>
</tr>
<tr>
<td>D_erase_window</td>
<td>()</td>
<td>Erase current window</td>
<td>161</td>
</tr>
<tr>
<td>D_get_cell_name</td>
<td>(name)</td>
<td>Retrieve cell file name</td>
<td>162</td>
</tr>
<tr>
<td>D_get_cur_window</td>
<td>(name)</td>
<td>Identify current graphics window</td>
<td>160</td>
</tr>
<tr>
<td>D_get_screen_window</td>
<td>(top, bottom, left, right)</td>
<td>Retrieve current window coordinates</td>
<td>160</td>
</tr>
<tr>
<td>D_new_window</td>
<td>(name, top, bottom, left, right)</td>
<td>Create new graphics window</td>
<td>160</td>
</tr>
<tr>
<td>D.overlay_cell_row</td>
<td>(row, raster)</td>
<td>Render a raster row without zeros</td>
<td>165</td>
</tr>
<tr>
<td>D_popup</td>
<td>(bcolor, tcolor, dcolor, top, left, size, options)</td>
<td>Pop-up menu</td>
<td>166</td>
</tr>
<tr>
<td>D_remove_window</td>
<td>()</td>
<td>Remove a window</td>
<td>161</td>
</tr>
<tr>
<td>D_set_colors</td>
<td>(colors)</td>
<td>Set colors in driver</td>
<td>167</td>
</tr>
<tr>
<td>D_set_screen_window</td>
<td>(top, bottom, left, right)</td>
<td>Reset current window position</td>
<td>161</td>
</tr>
<tr>
<td>D_set_cell_name</td>
<td>(name)</td>
<td>Add cell file name to display list</td>
<td>162</td>
</tr>
<tr>
<td>D_set_cur_wind</td>
<td>(name)</td>
<td>Set current graphics window</td>
<td>160</td>
</tr>
<tr>
<td>D_show_window</td>
<td>(color)</td>
<td>Outlines current window</td>
<td>160</td>
</tr>
<tr>
<td>D_timestamp</td>
<td>()</td>
<td>Give current time to window</td>
<td>161</td>
</tr>
<tr>
<td>D_translate_color</td>
<td>(name)</td>
<td>Color name to number</td>
<td>167</td>
</tr>
<tr>
<td>D_u_to_a_col</td>
<td>(east)</td>
<td>Earth to array (east)</td>
<td>163</td>
</tr>
<tr>
<td>D_u_to_a_row</td>
<td>(north)</td>
<td>Earth to array (north)</td>
<td>163</td>
</tr>
<tr>
<td>D_u_to_d_col</td>
<td>(east)</td>
<td>Earth to screen (east)</td>
<td>164</td>
</tr>
<tr>
<td>D_u_to_d_row</td>
<td>(north)</td>
<td>Earth to screen (north)</td>
<td>163</td>
</tr>
</tbody>
</table>
Appendix G

Index to Raster Graphics Library

Here is an index of Raster Graphics Library routines, with calling sequences and short function descriptions.

<table>
<thead>
<tr>
<th>Routine</th>
<th>Parameters</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>R_box_abs</td>
<td>(x1,y1,x2,y2)</td>
<td>fill a box</td>
<td>151</td>
</tr>
<tr>
<td>R_box_rel</td>
<td>(dx,dy)</td>
<td>fill a box</td>
<td>151</td>
</tr>
<tr>
<td>R_close_driver</td>
<td>()</td>
<td>terminate graphics</td>
<td>148</td>
</tr>
<tr>
<td>R_color</td>
<td>(color)</td>
<td>select color</td>
<td>149</td>
</tr>
<tr>
<td>R_color_table_fixed</td>
<td>()</td>
<td>select fixed color table</td>
<td>149</td>
</tr>
<tr>
<td>R_color_table_float</td>
<td>()</td>
<td>select floating color table</td>
<td>149</td>
</tr>
<tr>
<td>R_cont_abs</td>
<td>(x,y)</td>
<td>draw line</td>
<td>151</td>
</tr>
<tr>
<td>R_cont_rel</td>
<td>(dx,dy)</td>
<td>draw line</td>
<td>151</td>
</tr>
<tr>
<td>R_erase</td>
<td>()</td>
<td>erase screen</td>
<td>152</td>
</tr>
<tr>
<td>R_flush</td>
<td>()</td>
<td>flush graphics</td>
<td>152</td>
</tr>
<tr>
<td>R_font</td>
<td>(font)</td>
<td>choose font</td>
<td>155</td>
</tr>
<tr>
<td>R_get_location_with_box</td>
<td>(x,y,rx,ny,button)</td>
<td>get mouse location using a box</td>
<td>157</td>
</tr>
<tr>
<td>R_get_location_with_line</td>
<td>(x,y,rx,ny,button)</td>
<td>get mouse location using a line</td>
<td>156</td>
</tr>
<tr>
<td>R_get_location_with_pointer</td>
<td>(nx,ny,button)</td>
<td>get mouse location using pointer</td>
<td>156</td>
</tr>
<tr>
<td>R_get_text_box</td>
<td>(text, top, bottom, left, right)</td>
<td>get text extents</td>
<td>156</td>
</tr>
<tr>
<td>R_move_abs</td>
<td>(x,y)</td>
<td>move current location</td>
<td>150</td>
</tr>
<tr>
<td>R_move_rel</td>
<td>(dx,dy)</td>
<td>move current location</td>
<td>151</td>
</tr>
<tr>
<td>R_open_driver</td>
<td>()</td>
<td>initialize graphics</td>
<td>148</td>
</tr>
<tr>
<td>R_polydots_abs</td>
<td>(x,y,num)</td>
<td>draw a series of dots</td>
<td>152</td>
</tr>
<tr>
<td>R_polydots_rel</td>
<td>(x,y,num)</td>
<td>draw a series of dots</td>
<td>152</td>
</tr>
<tr>
<td>R_polygon_abs</td>
<td>(x,y,num)</td>
<td>draw a closed polygon</td>
<td>152</td>
</tr>
<tr>
<td>R_polygon_rel</td>
<td>(x,y,num)</td>
<td>draw a closed polygon</td>
<td>153</td>
</tr>
<tr>
<td>R_polyline_abs</td>
<td>(x,y,num)</td>
<td>draw an open polygon</td>
<td>153</td>
</tr>
<tr>
<td>R_polyline_rel</td>
<td>(x,y,num)</td>
<td>draw an open polygon</td>
<td>153</td>
</tr>
<tr>
<td>R_raster</td>
<td>(num,nrows,withzero,raster)</td>
<td>draw a raster</td>
<td>154</td>
</tr>
<tr>
<td>R_reset_color</td>
<td>(red, green, blue, num)</td>
<td>define single color</td>
<td>149</td>
</tr>
<tr>
<td>R_reset_colors</td>
<td>(min,max,red,green,blue)</td>
<td>define multiple colors</td>
<td>149</td>
</tr>
<tr>
<td>R_RGB_color</td>
<td>(red,green,blue)</td>
<td>select color</td>
<td>150</td>
</tr>
<tr>
<td>R_RGB_raster</td>
<td>(num,nrows,red,green,blue,withzero)</td>
<td>draw a raster</td>
<td>154</td>
</tr>
<tr>
<td>R_screen_bot</td>
<td>()</td>
<td>bottom of screen</td>
<td>150</td>
</tr>
<tr>
<td>R_screen_left</td>
<td>()</td>
<td>screen left edge</td>
<td>150</td>
</tr>
<tr>
<td>R_screen_right</td>
<td>()</td>
<td>screen right edge</td>
<td>150</td>
</tr>
<tr>
<td>R_screen_top</td>
<td>()</td>
<td>top of screen</td>
<td>150</td>
</tr>
<tr>
<td>R_set_RGB_color</td>
<td>(red,green,blue)</td>
<td>initialize graphics</td>
<td>154</td>
</tr>
<tr>
<td>R_set_window</td>
<td>(top,bottom,left,right)</td>
<td>set text clipping window</td>
<td>155</td>
</tr>
<tr>
<td>R_standard_color</td>
<td>(color)</td>
<td>select standard color</td>
<td>150</td>
</tr>
<tr>
<td>R_text_size</td>
<td>(width, height)</td>
<td>set text size</td>
<td>155</td>
</tr>
<tr>
<td>R_text</td>
<td>(text)</td>
<td>write text</td>
<td>156</td>
</tr>
</tbody>
</table>
Appendix H

Index to Rowio Library

Here is an index of Rowio Library routines, with calling sequences and short function descriptions.

<table>
<thead>
<tr>
<th>Routine</th>
<th>Parameters</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>rowio_fileno</td>
<td>(r)</td>
<td>get file descriptor</td>
<td>176</td>
</tr>
<tr>
<td>rowio_flush</td>
<td>(r)</td>
<td>force pending updates to disk</td>
<td>176</td>
</tr>
<tr>
<td>rowio_forget</td>
<td>(r, n)</td>
<td>forget a row</td>
<td>175</td>
</tr>
<tr>
<td>rowio_get</td>
<td>(r, n)</td>
<td>read a row</td>
<td>175</td>
</tr>
<tr>
<td>rowio_put</td>
<td>(r, buf, n)</td>
<td>write a row</td>
<td>176</td>
</tr>
<tr>
<td>rowio_release</td>
<td>(r)</td>
<td>free allocated memory</td>
<td>176</td>
</tr>
<tr>
<td>rowio_setup</td>
<td>(r, fd, nrows, len, getrow, putrow)</td>
<td>configure rowio structure</td>
<td>174</td>
</tr>
</tbody>
</table>
Appendix I

Index to Segment Library

Here is an index of Segment Library routines, with calling sequences and short function descriptions.

<table>
<thead>
<tr>
<th>Routine</th>
<th>Parameters</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>segment_flush</td>
<td>(seg)</td>
<td>flush pending updates to disk</td>
<td>182</td>
</tr>
<tr>
<td>segment_format</td>
<td>(fd, rows, ncols, srows, scols, len)</td>
<td>format a segment file</td>
<td>180</td>
</tr>
<tr>
<td>segment_get_row</td>
<td>(seg, buf, row)</td>
<td>read row from segment file</td>
<td>182</td>
</tr>
<tr>
<td>segment_get</td>
<td>(seg, value, row, col)</td>
<td>get value from segment file</td>
<td>181</td>
</tr>
<tr>
<td>segment_init</td>
<td>(seg, fd, nseg)</td>
<td>initialize segment structure</td>
<td>181</td>
</tr>
<tr>
<td>segment_put_row</td>
<td>(seg, buf, row)</td>
<td>write row to segment file</td>
<td>181</td>
</tr>
<tr>
<td>segment_put</td>
<td>(seg, value, row, col)</td>
<td>put value to segment file</td>
<td>182</td>
</tr>
<tr>
<td>segment_release</td>
<td>(seg)</td>
<td>free allocated memory</td>
<td>183</td>
</tr>
</tbody>
</table>
Appendix J

Index to Vask Library

Here is an index of Vask Library routines, with calling sequences and short function descriptions.

<table>
<thead>
<tr>
<th>routine</th>
<th>parameters</th>
<th>description</th>
<th>page</th>
</tr>
</thead>
<tbody>
<tr>
<td>V_call</td>
<td>()</td>
<td>interact with the user</td>
<td>189</td>
</tr>
<tr>
<td>V_clear</td>
<td>()</td>
<td>initialize screen description</td>
<td>188</td>
</tr>
<tr>
<td>V_const</td>
<td>(value, type, row, col, len)</td>
<td>define screen constant</td>
<td>188</td>
</tr>
<tr>
<td>V_float_accuracy</td>
<td>(num)</td>
<td>set number of decimal places</td>
<td>189</td>
</tr>
<tr>
<td>V_intpt_msg</td>
<td>(text)</td>
<td>change ctrl-c message</td>
<td>190</td>
</tr>
<tr>
<td>V_intpt_ok</td>
<td>()</td>
<td>allow ctrl-c</td>
<td>189</td>
</tr>
<tr>
<td>V_line</td>
<td>(num, text)</td>
<td>add line of text to screen</td>
<td>188</td>
</tr>
<tr>
<td>V_ques</td>
<td>(value, type, row, col, len)</td>
<td>define screen question</td>
<td>188</td>
</tr>
</tbody>
</table>
Appendix K

Permuted Index for Library Subroutines

end level one vector access dig_fin() 125
initialize level one vector access dig_init() 125
end level two vector access dig_P_fin() 128
initialize level two vector access dig_P_init() 128
get the active window G_get_set_window() 79
set the active window G_set_window() 79
number of columns in active window G_window_cols() 78
number of rows in active window G_window_rows() 78
add cell file name to display list D_set_cell_name() 162
add command to window display list D_add_to_list() 161
add file name to Ref structure L_add_file_to_group_ref() 141
add line of text to screen V_line() 188
add new control point G_allocate_cell_buf() 86
allocate a cell buffer G_calloc() 76
allocate a memory segment G_malloc() 76
allow ctrl-c V_intert_ok() 189
distance to arc dig_check_dist() 132
read arc dig_P_read_line() 130
read next arc dig_P_read_next_line() 131
prune a dense arc dig_prune() 135
read arc dig_Read_line() 133
get next arc dig_read_next_line() 126
write arc dig_Write_line() 133
get arc bounding box dig_bound_box() 135
get arc bounding box dig_P_get_line_bbox() 131
get next arc by type dig_read_next_line_type() 126
get arc category attribute dig_P_line_att() 131
read arc in box dig_read_line_in_box() 127
limit arc search in box dig_init_box() 127
find arc with point dig_point_to_line() 132
get number of arcs dig_P_num_lines() 130
point in area dig_point_in_area() 132
get area bounding box dig_P_get_area_bbox() 130
get area category attribute dig_P_area_att() 130
get area polygon dig_P_get_area() 129
get area polygon dig_P_get_area_xy() 129
find area with point dig_point_to_area() 132
get number of areas dig_P_num_areas() 129
earth to array (east) D_e_to_a_col() 163
earth to array (north) D_e_to_a_row() 163
array to screen (column) D_a_to_d_col() 163
array to screen (row) D_a_to_d_row() 163
screen to array (x) D_d_to_a_col() 164

Permuted Index for Library Subroutines
Screen to array (y)
make aspect colors
get area category attribute
get arc category attribute
get arc bounding box
get area bounding box
get arc bounding box
limit arc search in box
get area bounding box
get arc bounding box
read arc in box
fill a box
get mouse location using a box
allocate a cell buffer
zero a cell buffer
query cartographic projection
query cartographic projection
query cartographic zone
convert string to lower case
convert string to upper case
get area category attribute
get arc category attribute
get a category color
set a category color
read cell category file
write cell category file
write vector category file
get a category label
set a category label
get title from category structure
initialize category structure
set title in category structure
free category structure memory
allocate a cell buffer
zero a cell buffer
read cell category file
write cell category file
prompt for existing cell file
prompt for new cell file
prompt for existing cell file
close a cell file
find a cell file
find a cell file
read a cell file
open an existing cell file
unopen a cell file
retrieve cell file name
add cell file name to display list
open a new cell file (random) G_open_cell_new_random()
write a cell file (random) G_put_map_row_random()
open a new cell file (sequential) G_open_cell_new()
write a cell file (sequential) G_put_map_row()
open a new cell file (uncompressed) G_open_cell_new_uncompressed()
read a cell file (without masking) G_get_map_row_unmasked()
read the cell header G_get_cellhdr()
write the cell header G_put_cellhdr()
read cell history file G_read_history()
write cell history file G_write_history()
read cell range G_read_range()
write cell range G_write_range()
get cell title G_get_cell_title()
change cell title G_put_cell_title()
change ctrl-c message V_interrupt_msg() 150
change error handling G_set_error_routine()
return interrupt G_intr_char() 117
printable version of control character G_printable_version_of_control()
create a protected child process G_fork() 116
choose font G_font()
set window display lists D_clear_window() 161
clear window display lists D_clear_window() 161
clears information about current window D_clear_window()
set text clipping window R_set_window() 155
close a cell file G_close_cell()
draw a closed polygon R_draw_polygon() 128
draw a closed polygon R_draw_polygon() 152
close vector map G_close_vector_map()
close vector map G_close_vector_map()
reopen vector map R_reopen_vector_map() 128
get a category color G_get_color()
set a category color G_set_color()
select color R_color() 149
define single color R_define_color() 149
select color R_color() 149
select standard color R_standard_color() 150
make aspect colors G_make_aspect_colors() 97
make rainbow colors G_make_rainbow_colors() 97
make random colors G_make_random_colors() 97
make various colors G_make_red_yellow_green() 98
define multiple colors G_define_multiple_colors() 98
set colors in driver D_set_colors() 163
array to screen (columns) D_array_to_screen() 163
number of columns in active window G_window_cols() 78

Permutated Index for Library Subroutines
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>run a shell level command</code></td>
<td>A shell-level command.</td>
</tr>
<tr>
<td><code>print percent</code></td>
<td>Print percent.</td>
</tr>
<tr>
<td><code>define screen print warning message and continue</code></td>
<td>Define screen print warning message and continue.</td>
</tr>
<tr>
<td><code>printable version of read group write group initialize</code></td>
<td>Print a readable version of read group write group initialize.</td>
</tr>
<tr>
<td><code>retrieve current window coordinates clip</code></td>
<td>Retrieve current window coordinates clip.</td>
</tr>
<tr>
<td><code>allow change get a line of input</code></td>
<td>Allow change get a line of input.</td>
</tr>
<tr>
<td><code>identify set move move current location directory</code></td>
<td>Identify set move move current location directory.</td>
</tr>
<tr>
<td><code>assign/retrieve current location name give clear information about</code></td>
<td>Assign/retrieve current location name give clear information about.</td>
</tr>
<tr>
<td><code>erase outlines retrieve resets</code></td>
<td>Erase outlines retrieve resets.</td>
</tr>
<tr>
<td><code>top level database directory</code></td>
<td>Top level database directory.</td>
</tr>
<tr>
<td><code>prompt for existing prompt for new</code></td>
<td>Prompt for existing prompt for new.</td>
</tr>
<tr>
<td><code>find a</code></td>
<td>Find a.</td>
</tr>
<tr>
<td><code>open a new file</code></td>
<td>Open a new file.</td>
</tr>
<tr>
<td><code>remove a</code></td>
<td>Remove a.</td>
</tr>
<tr>
<td><code>rename a</code></td>
<td>Rename a.</td>
</tr>
<tr>
<td><code>open a database file</code></td>
<td>Open a database file.</td>
</tr>
</tbody>
</table>

**Permutated Index for Library Subroutines**
open a database file for reading
open a database file for update
open a database file for update
check for legal read the database file names
read the database window
write the database window
current number of decimal places
set number of read the default window
define multiple colors
prune a dense arc
initialize screen description
get a line of input (detect ctrl-z)
top level program directory
top level database directory
user's home directory
current location directory
force pending updates to disk
flush pending updates to disk
add command to window display list
add cell file name to display list
clear window display lists
display vector header information
distance to arc
distance to line-segment
distance to line-segment
does group exist?
draw a series of dots
draw a series of dots
draw a closed polygon
draw a closed polygon
draw a raster
draw a raster
draw a series of dots
draw a series of dots
draw an open polygon
draw an open polygon
draw line
draw line
set colors in driver
draw arcs
earth to array (east)
draw arcs
earth to array (north)
draw arcs
earth to screen (east)
draw arcs
earth to screen (north)
draw arcs
earth (x)
draw arcs
earth (y)
draw arcs
earth (east)
draw arcs
earth (north)
draw arcs
screen to edge
draw arcs
screen left
draw arcs
screen right
draw arcs

draw arcs

Permutted Index for Library Subroutines
reclass file?
open an existing cell file
open a new database file
write site list file
read cell category file
read cell history file
read vector category file
remove a database file
rename a database file
unopen a cell file
write cell category file
write cell history file
write vector category file
read group REF file
read subgroup REF file
write group REF file
write subgroup REF file
format a segment file
get value from segment file
read row from segment file
put value to segment file
write row to segment file
get file descriptor
open a database file for reading
open a database file for reading
open a database file for update
open a database file for update
retrieve cell file name
prompt for any valid file name
returns a temporary file name
add cell file name to display list
add file name to Ref structure
check for legal database file names
open a new cell file (random)
write a cell file (random)
open a new cell file (sequential)
write a cell file (sequential)
open a new cell file (uncompressed)
read a cell file (without masking)
fill a box
fill a box
find a cell file
find a cell file
find a database file
find a database file
find a vector file
find a vector file
find arc with point
dig_point_to_line()
dig_point_to_area()
select fixed color table
select floating color table
flush graphics
flush pending updates to disk
choose font
G_is_reclass() 91
G_open_cell_old() 83
G_open_new() 74
G_put_site() 108
G_read_cats() 91
G_read_history() 98
G_read_vector_cats() 106
G_remove() 75
G_rename() 75
G_unopen_cell() 89
G_write_cats() 92
G_write_history() 99
G_write_vector_cats() 105
G_get_group_ref() 140
G_get_subgroup_ref() 140
G_put_group_ref() 140
G_put_subgroup_ref() 140
segment_format() 180
segment_get() 181
segment_get_row() 182
segment_put() 182
segment_put_row() 181
rowio_fileno() 176
G_open_old() 73
G_open_old() 73
G_open_append() 73
G_open_update() 73
D_get_cell_name() 162
G_ask_any() 70
G_tempfile() 108
D_set_cell_name() 162
L_add_file_to_group_ref() 141
G_legal_filename() 72
G_open_cell_new_random() 85
G_put_map_row_random() 88
G_open_cell_new() 84
G_put_map_row() 83
G_open_cell_new_uncompressed() 85
G_get_map_row_nonmask() 87
R_box_abs() 151
R_box_rel() 151
G_find_cell2() 82
G_find_cell() 82
G_find_file2() 71
G_find_file() 71
G_find_vector2() 102
G_find_vector() 102
dig_point_to_line() 132
dig_point_to_area() 132
R_color_table_fixed() 149
R_color_table_float() 149
R_flush() 152
segment_flush() 182
R_font() 157

Permuted Index for Library Subroutines
force pending updates to disk rowio_flush() 176
forget a row rowio_forget() 175
format a segment file segment_format() 180
free allocated memory rowio_release() 176
free allocated memory segment_release() 183
free category structure memory G_free_cats() 94
free color structure memory G_free_colors() 96
free Ref structure L_free_group_ref() 142
initialize gis library G_gisinit() 64
initialize graphics D_timestamp() 161
prepare for raster graphics D_cell_draw_setup() 165
terminate graphics R_close_driver() 148
flash graphics R_flush() 152
initialize graphics R_open_driver() 148
initialize graphics R_set_RGB_color() 154
identify current graphics window D_get_cur_win() 160
create new graphics window D_new_window() 160
set current graphics window D_set_cur_win() 160
query GRASS environment variable G_getenv() 67
query GRASS environment variable G_getenv() 67
set GRASS environment variable G_setenv() 67
set GRASS environment variable G_setenv() 67
make linear grey scale G_make_grey_scale() 97
prompt for new group L_group_new() 138
prompt for an existing group L_group_old() 138
read group control points L_get_control_points() 143
write group control points L_put_control_points() 144
read group does group exist? L_find_group() 139
read group any() L_group_any() 139
read group REF file L_get_group_ref() 140
write group REF file L_put_group_ref() 140
change error handling G_set_error_routine() 65
reset normal error handling G_unset_error_routine() 65
read vector header dig_read_head_binary() 134
write vector header dig_write_head_binary() 134
read the cell header G_get_cellhd() 90
write the cell header G_put_cellhd() 90
display vector header information dig_print_header() 125
read cell history file G_read_history() 98
write cell history file G_write_history() 99
initialize history structure G_short_history() 99
get Hit RETURN msg G_get_ask_return_msg() 70
set Hit RETURN msg G_set_ask_return_msg() 70
user's home directory G_home() 117
display vector header information D_get_cur_win() 160
identify current graphics window dig_print_header() 125
read target information L_get_target() 142
write target information L_put_target() 142
clears information about current window D_clear_window() 161
initialize category structure G_init_cats() 93
initialize color structure G_init_colors() 96
initialize conversions D_do_conversions() 162
initialize gis library G_gisinit() 64
initialize graphics R_open_driver() 148

Permutated Index for Library Subroutines
Permuted Index for Library Subroutines
get number of arcs

dig_P_num_lines()

get number of arcs

dig_P_num_areas()

number of columns in active window

G_window_cols()

number of decimal places

V_float_accuracy()

number of rows in active window

G_window_rows()

open a database file for reading

G_open_old()

open a database file for update

G_open_update()

open a database file for update

G_open_append()

open a new cell file (random)

G_open_cell_new_random()

open a new cell file (sequential)

G_open_cell_new()

open a new cell file (uncompressed)

G_open_cell_new_uncompressed()

open a new database file

G_open_new()

open an existing cell file

G_open_cell_old()

open an existing site list file

G_open_sites_new()

open a new vector file

G_open_vector_new()

open an existing cell file

G_open_cell_old()

open an existing site list file

G_open_sites_old()

open an existing vector file

G_open_vector_old()

draw an open polygon

R polyline_abs()

draw an open polygon

R polyline_rel()

outlines current window

D_show_window()

parse command line

G_parse_command()

force pending updates to disk

rowo_flush()

flush pending updates to disk

segment_flush()

print percent complete messages

G_percent()

set number of decimal places

V_float_accuracy()

dig_point_to_area()

dig_point_to_line()

find area with point

dig_point_in_area()

find arc with point

dig_P_rewind()

add new control point

l_new_control_point()

rewind next arc pointer

dig_P_rewind()

get mouse location using printer

R_get_location_with_pointer()

read group control points

l_get_control_points()

write group control points

l_put_control_points()

get area polygon

dig_P_get_area()

dig_P_get_area_xy()

dig_P_get_area_xxy()

dig_P_get_area_xyx()

dig_P_get_area_yxx()

draw a closed polygon

R_polyline_abst()

draw a closed polygon

R_polyline_relt()

draw an open polygon

R_polyline_abst()

draw an open polygon

R_polyline_relt()

pop-up menu

D_popup()

reset current window position

D_reset_screen_window()

prepare for raster graphics

D_cell_draw_setup()

print error message and exit

G_fatal_error()

print percent complete messages

G_percent()

print warning message and continue

G_warning()

create a protected child process

G_fork()

top level program directory

G_gdbase()

return program name

G_program_name()

open cartographic projection

G_projection()

open cartographic projection

G_projection_name()

prompt for a new vector file

G_ask_vector_new()
prompt for an existing group
prompt for an existing vector file
prompt for an existing vector file
prompt for any valid file name
prompt for any valid group name
prompt for existing cell file
prompt for existing cell file
prompt for existing database file
prompt for existing database file
prompt for existing site list file
prompt for existing site list file
prompt for new cell file
prompt for new cell file
prompt for new database file
prompt for new database file
create a protected child process
put value to segment file
put value to segment file
query cartographic projection
query cartographic projection
query cartographic zone
query GRASS environment variable
query GRASS environment variable
ask a yes/no question
define screen
question
make color
ramp
open a new cell file
(open)
write a cell file
(make)
read cell range
write cell range
initialize range structure
update range structure
update range structure
draw a raster
draw a raster
prepare for raster graphics
render a raster
render a raster row without zeros
read a cell file
read a cell file (without masking)
read a row
read arc
read arc
read arc in box
read cell category file
read cell history file
read cell range
read group control points
read group REP file
read map layer color table
read next arc
read row from segment file

L_ask_group_old() 138
G_ask_vector_in_mapset() 101
G_ask_vector_old() 101
G_ask_any() 70
L_ask_group_any() 139
G_ask_cell_in_mapset() 81
G_ask_cell_old() 81
G_ask_in_mapset() 70
G_ask_old() 69
G_ask_sites_in_mapset() 106
G_ask_sites_old() 106
G_ask_cell_new() 81
G_ask_new() 69
L_ask_group_new() 138
G_ask_sites_new() 106
G_fork() 116
dig_prune() 135
segment_put() 182
G_projection() 80
G_projection_name() 80
G_zone() 80
G_getenv() 67
G_getenv() 67
G_yes() 118
V_quest() 188
G_make_rainbow_colors() 97
G_make_color_ramp() 97
G_open_cell_new_random() 85
G_put_map_row_random() 88
G_make_random_colors() 98
G_read_range() 99
G_write_range() 100
G_init_range() 100
G_row_update_range() 100
G_update_range() 100
R_raster() 154
R_RGB_raster() 154
D_cell_draw_setup() 165
D_draw_cell_row() 165
D_overlay_cell_row() 165
G_get_map_row() 87
G_get_map_row_nomask() 87
rowio_get() 175
dig_P_read_line() 130
dig_Read_line() 133
dig_read_line_in_box() 127
G_read_cats() 91
G_read_history() 98
G_read_range() 99
L_get_control_points() 143
L_get_group_ref() 140
G_read_colors() 94
dig_P_read_next_line() 131
segment_get_row() 182
read site list file G_get_site( ) 107
read subgroup REF file L_get_subgroup_ref( ) 140
read target information L_get_target( ) 142
read the cell header G_get_cellhdr( ) 90
read the database window G_get_window( ) 77
read the default window G_get_default_window( ) 78
read vector category file G_read_vector_cat( ) 106
read vector header dig_read_head_binary( ) 134

open a database file for reading G_open_old( ) 73
make red, yellow, green colors G_make_red_yel_gm( ) 98
read subgroup REF file L_getzgroup_ref( ) 140
write group REF file L_putzgroup_ref( ) 140
write subgroup REF file L_putzgroup_ref( ) 140
rewind vector file G_ rewind( ) 131
rewind next arc pointer dig_P_rewind( ) 131
rewind vector file dig_rewind( ) 125
array to screen rowio_get( ) 175
array to screen rowio_forget( ) 175
array to screen rowio_setup( ) 174
conform rowio_structure( ) 174
number of rows in active window G_window_rows( ) 78
number of rows in active window G_system( ) 116
make linear grey scale G_make_grey_scale( ) 97
erase screen R_erase( ) 152
bottom of screen R_screen_bot( ) 150

Permuted Index for Library Subroutines
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>top of screen</td>
<td>270</td>
</tr>
<tr>
<td>add line of text to screen</td>
<td>150</td>
</tr>
<tr>
<td>array to screen</td>
<td>188</td>
</tr>
<tr>
<td>define screen (column)</td>
<td>150</td>
</tr>
<tr>
<td>initialize screen description</td>
<td>163</td>
</tr>
<tr>
<td>set earth to screen (east)</td>
<td>164</td>
</tr>
<tr>
<td>set earth to screen (north)</td>
<td>163</td>
</tr>
<tr>
<td>set earth to screen (row)</td>
<td>188</td>
</tr>
<tr>
<td>screen right edge</td>
<td>150</td>
</tr>
<tr>
<td>screen to array (x)</td>
<td>163</td>
</tr>
<tr>
<td>screen to array (y)</td>
<td>164</td>
</tr>
<tr>
<td>screen to earth (x)</td>
<td>164</td>
</tr>
<tr>
<td>screen to earth (y)</td>
<td>164</td>
</tr>
<tr>
<td>limit arc</td>
<td>127</td>
</tr>
<tr>
<td>format a segment file</td>
<td>180</td>
</tr>
<tr>
<td>get value from segment file</td>
<td>181</td>
</tr>
<tr>
<td>read row from segment file</td>
<td>182</td>
</tr>
<tr>
<td>put value to segment file</td>
<td>181</td>
</tr>
<tr>
<td>write row to segment file</td>
<td>149</td>
</tr>
<tr>
<td>initialize segment structure</td>
<td>149</td>
</tr>
<tr>
<td>select color</td>
<td>149</td>
</tr>
<tr>
<td>select color</td>
<td>150</td>
</tr>
<tr>
<td>select fixed color table</td>
<td>149</td>
</tr>
<tr>
<td>select floating color table</td>
<td>149</td>
</tr>
<tr>
<td>select standard color</td>
<td>150</td>
</tr>
<tr>
<td>open a new cell file (sequential)</td>
<td>84</td>
</tr>
<tr>
<td>write a cell file (sequential)</td>
<td>88</td>
</tr>
<tr>
<td>draw a series of dots</td>
<td>152</td>
</tr>
<tr>
<td>draw a series of dots</td>
<td>152</td>
</tr>
<tr>
<td>set a category color</td>
<td>96</td>
</tr>
<tr>
<td>set a category label</td>
<td>94</td>
</tr>
<tr>
<td>set colors in driver</td>
<td>167</td>
</tr>
<tr>
<td>set current graphics window</td>
<td>160</td>
</tr>
<tr>
<td>set GRASS environment variable</td>
<td>67</td>
</tr>
<tr>
<td>set GRASS environment variable</td>
<td>67</td>
</tr>
<tr>
<td>set Hit RETURN mag</td>
<td>70</td>
</tr>
<tr>
<td>set number of decimal places</td>
<td>189</td>
</tr>
<tr>
<td>set text clipping window</td>
<td>155</td>
</tr>
<tr>
<td>set text size</td>
<td>155</td>
</tr>
<tr>
<td>set the active window</td>
<td>79</td>
</tr>
<tr>
<td>set title in category structure</td>
<td>94</td>
</tr>
<tr>
<td>run a shell level command</td>
<td>116</td>
</tr>
<tr>
<td>define single color</td>
<td>149</td>
</tr>
<tr>
<td>prompt for existing site list</td>
<td>106</td>
</tr>
<tr>
<td>prompt for new site list</td>
<td>106</td>
</tr>
<tr>
<td>prompt for existing site list</td>
<td>106</td>
</tr>
<tr>
<td>open a new site list</td>
<td>107</td>
</tr>
<tr>
<td>open an existing site list</td>
<td>107</td>
</tr>
<tr>
<td>read site list</td>
<td>108</td>
</tr>
<tr>
<td>write site list</td>
<td>155</td>
</tr>
<tr>
<td>set text size</td>
<td>65</td>
</tr>
<tr>
<td>sleep on error?</td>
<td></td>
</tr>
</tbody>
</table>

**Permutated Index for Library Subroutines**
remove unnecessary white space
remove leading/trailing white space
select standard color
copy string to allocated memory
convert string to lower case
convert string to upper case
concatenate strings
get title from category
initialize category structure
initialize color structure
initialize range structure
set title in category
initialize history structure
update range structure
add file name to Ref structure
free Ref structure
initialize Ref structure
configure rowio structure
initialize segment structure
free category structure
free color structure
read subgroup REF file
write subgroup REF file
read map layer color table
write map layer color table
select fixed color table
select floating color table
read target information
write target information
temporary close vector map
terminate graphics
write text
set text clipping window
get text extents
set text size
add line of text to screen
get cell title
location title
change cell title
get title from category structure
set title in category structure
top level database directory
top level program directory
top of screen
get next arc by type
open a new cell file
remove unnecessary white space
unopen a cell file
open a database file for update

Permutated Index for Library Subroutines
open a database file for

force pending
flush pending
convert string to
command line
interact with the

get mouse location
get mouse location
prompt for any
prompt for any
get
put

query GRASS environment
query GRASS environment
set GRASS environment
set GRASS environment

end level one vector access
initialize level one
end level two
initialize level two
read
write
rewind
prompt for an existing
prompt for a new
prompt for an existing
find a
find a
open a new
open an existing
read
write
display
temporary close
reopen closed
printable

print
suppress
make color
remove unnecessary
remove leading/trailing
assign/retreive current map
clears information about current
clip coordinates to
erase current
identify current graphics
create new graphics
remove a
set current graphics

G_open_update() 73
G_row_update_range() 100
G_update_range() 100
rowio_flush() 176
segment_flush() 182
G_toucase() 114
G_parse_command_usage() 111
V_call() 189
G_home() 117
G_whoami() 118
R_get_location_with_box() 157
R_get_location_with_line() 156
R_get_location_with_pointer() 156
G_ask_any() 70
L_ask_group_any() 139
segment_get() 181
segment_put() 182
G_getenv() 67
G_getenvv() 67
G_setenv() 67
G_setenvv() 67
dig_fin() 125
dig_init() 125
dig_P_fin() 128
dig_P_init() 128
G_read_vector_cats() 105
G_write_vector_cats() 105
dig_terminate() 125
G_ask_vector_in_mapset() 101
G_ask_vector_new() 101
G_ask_vector_old() 101
G_find_vector2() 102
G_find_vector() 102
G_open_vector_new() 104
G_open_vector_old() 103
dig_read_head_binary() 134
dig_write_head_binary() 134
dig_print_header() 125
dig_P_tmp_close() 128
dig_P_tmp_open() 128
G_uncut() 114
G_warning() 64
G_suppress_warnings() 65
G_make_color_wave() 97
G_spacez() 113
G_strip() 114
D_check_map_window() 160
D_clear_window() 161
D_clip() 166
D_erase_window() 161
D_get_cur_wind() 160
D_new_window() 160
D_remove_window() 161
D_set_cur_wind() 160

Permutated Index for Library Subroutines
outlines current window
give current time to window
give current time to window
give current time to window
give current time to window
give current time to window
give current time to window
give current time to window
read the default window
get the active window
read the database window
write the database window
set the active window
number of columns in active window
number of rows in active window
set text clipping window
retrieve current window coordinates
add command to window display list
clear window display lists
reset current window position
read a cell file
read a cell file (random)
write a cell file (random)
write a cell file (sequential)
write a row
write any
write cell category file
write cell history file
write cell range
write cell range
write group control points
write cell range
write group control points
write group control points
write group control points
write map layer color table
write row to segment file
write site list file
write subgroup REF file
write target information
write text
write the cell header
write the cell header
write the database window
write vector category file
write vector category file
write vector category file
write vector category file
write vector category file
write vector category file
write vector category file
write vector category file
write vector category file
write vector category file
write vector category file
write vector category file
data to array (x)
data to array (x)
data to array (x)
data to array (x)
data to array (y)
data to array (y)
data to array (y)
data to array (y)
screen to array (x)
screen to array (x)
screen to array (y)
screen to array (y)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
screen to array (x)
read a master row without zeros
query cartographic zone

Permuted Index for Library Subroutines
Index

finding 82
history file 29, 98
opening (new) 84
opening (read) 83
programming interface 24, 80
prompting for 85
range file 29, 99
reading 86
reclass format 27
resolution 27
writing 87

cell header:
  see: cell files
color table:
  see: cell files
colors:
  see: cell files
  see: Display Graphics Library
  see: Raster Graphics Library
color.h 160
compiling:
  Gmake 55, 122, 135, 144, 157, 167, 170, 177, 185, 191
curses:
  Gmake 57
  Vask Library 187, 191, 192

database:
  access permissions 20
  programming interface 15, 68
  search path 20
title 19, 66
database structure 15
  $GISDBASE 16
  location 16
  $LOCATION_NAME 16
  $MAPSET 16
  mapset 16
date:
  G_date() 117
DEFAULT_WIND 19
diagnostics:
  see: error messages
Dig Library 12, 123
  arc types 124
  include files 123
  INDEX of routines 243
  INDEX, permuted 257
level one access 124
level two access 127

A

access permissions:
  GRASS 20
  UNIX 20
Approach 1

B

Background 1
band files 41
Bourne Shell:
  shell scripts 229

C

category file:
  see: cell files, vector files
category number:
  cell 23
  vector 31
CELL 79, 80, 86, 87, 88, 93, 94, 95, 96, 97, 98, 100,
  105, 237
cell files 23
  see also: GIS Library
  allocate CELL buffer 88
  and the database 24
category file 28, 91
category number 23
cell file format 24
cell header 26, 89
closing 89
color file 28, 94

D

database:
  access permissions 20
  programming interface 15, 68
  search path 20
title 19, 66
database structure 15
  $GISDBASE 16
  location 16
  $LOCATION_NAME 16
  $MAPSET 16
  mapset 16
date:
  G_date() 117
DEFAULT_WIND 19
diagnostics:
  see: error messages
Dig Library 12, 123
  arc types 124
  include files 123
  INDEX of routines 243
  INDEX, permuted 257
level one access 124
level two access 127
levels of access 124
LOADING the library 135
writing vector files 133
digit files:
see: vector files
dig:
index of dig_ routines (Dig Library) 243
dig_defines.h 124, 126
dig_structs.h 124
Display Graphics Library 12, 57, 159
colors 167
coordinate transformation 162
INDEX of routines 247
INDEX, permuted 257
LOADING the library 167
popup menus 166
raster graphics 164
window clipping 165
window contents 161
windows 159
drivers:
writing a digitizer driver 156
writing a graphics driver 207
writing a paint driver 215
D_:
index of D_ routines (Display Graphics
Library) 247
E
elements 17, 18
environment 15, 51, 66
G_getenv() 67
G_getenv() 67
G_gisbase() 66
G_gisbase() 67
$GISBASE 51
gisbase 52
$GIS_LOCK 51
$GISRC 51
G_location() 66
G_mapset() 66
GRASS 52
grass 52
G_getenv() 67
G_getenv() 67
location 52
mapset 52
UNIX 51
error messages 64
GIS_ERROR_LOG 64
F
fork() 84
G_fork() 115
G
Gisk:
and shell scripts 230
gets():
G_gets() 117
Gfindfile:
and shell scripts 231
GIS Library 11, 56, 63
allocate CELL buffer 85
and UNIX 115
cell category file 91
cell color table 94
cell file support 89
cell files 80
cell header 89
cell history file 98
cell range 99
closing cell files 89
command line parsing 109
data structures 118
database access 68
database information 66
database management 75
environment information 66
error messages 64
finding cell files 82
finding database files 70
finding vector files 102
fork() 115
gets() 117
INDEX of routines 239
INDEX, permuted 257
initialization 64
legal file names 72
LOADING the library 122
memory allocation 75
open cell file (new) 84
open cell file (read) 83
opening a vector file (read) 103
opening database files (read) 72
opening database files (update) 73
opening database files (write) 74
opening site files 107
opening vector files (new) 104
projection 79
prompting for cell files 81
prompting for database files 68
prompting for site files 106
prompting for vector files 101
reading and writing site files 107
reading cell files 86
sites 105
string routines 113
struct Categories 119
struct Cell_head 119
struct Colors 120
struct History 121
struct Range 122
system() 116
tempfiles 108
vector category file 104
vector files 100
window 76
writing cell files 87
gis.h 56, 63, 76, 80, 118, 137, 237
gisbase 52
.grassrc 52
GIS_ERROR_LOG 64
Gmake 11, 55, 202, 216, 225
variables 56, 233
Gmakefile 11, 55, 202, 216, 225, 226
and Dig library 135
and Display Graphics Library 167
and GIS Library 122
and Imagery Library 144
and Lock Library 170
and Raster Graphics Library 157
and Rowio Library 177
and Segment Library 185
and Vask Library 191
construction of 58
graphics:
see: Display Graphics Library
see: Raster Graphics Library
GRASS:
Information Center 2, 3, 4
Inter-Agency Steering Committee 3
User Group Meeting 3
GRASSClirnings 3
GRASSNET 3
grid cell:
see: cell files
group 41, 117, 138
see also: Imagery Library
finding 1,9
POINTS file 44
POINTS file routines 143
programming interface 46
prompting for 138
REF file 43
REF file routines 139
structure of a group 43
subgroup 44
TARGET file 44
TARGET file routines 142
Guidelines 5
G:
index of G_routines (GIS Library) 239
H
history file:
see: cell files
home directory:
G_home() 117
I
imagery:
band files 41
image classification 42
image rectification 42
image registration 42
programs 45
x,y projection 26, 42, 48, 80
Imagery Library 137
data structures 144
finding groups 139
group 138
group POINTS files 143
group REF file 139
group TARGET file 142
INDEX of routines 245
INDEX, permuted 257
LOADING the library 144
prompting for a group 138
struct Control_Points 145
struct REF 144
imagery.h 137, 139, 143, 144
import:
vector files 38
include files:
colom.h 160
dig defines.h 124, 126
dig_structs.h 124
gis.h 56, 63, 76, 80, 118, 137, 237
imagery.h 137, 139, 143, 144
rowio.h 174
segment.h 180
index:
Dig Library 243
Display Graphics Library 247
GIS Library 239
Imagery Library 245
permuted 257
Raster Graphics Library 249
Rowio Library 251
Segment Library 253
Vask Library 255
L
library:
  see: Dig Library
  see: Display Graphics Library
  see: GIS Library
  see: Imagery Library
  see: Lock Library
  see: Raster Graphics Library
  see: Rowio Library
  see: Segment Library
  see: Vask Library
  how to build 60
  permuted index 257
location 52
  G_location() 66
  .grassc 52
Lock Library 169
  INDEX, permuted 257
  LOADING the library 170
login name:
  G_whoami() 118
longitude/latitude:
  see: projection
M
map:
  see: cell files, vector files
map layer:
  see: cell files, vector files
mapset 16, 24, 32, 52
  access permissions 20
  cell files 24
  current mapset 16, 20, 47, 48, 68, 69, 70, 71, 73,
  74, 75, 77, 81, 82, 83, 84, 86, 88, 90, 100, 101,
  102, 103, 104, 105, 106, 107, 108, 109
  elements 17, 18
  files 17
  G_mapset() 66
  .grassc 52
  $MAPSET 16
  mask 48
  PERMANENT 17, 19
  search path 17, 20, 68, 71, 82, 83, 102
  structure of a mapset 17
  subdirectories 17
  vector files 32
  window 17, 47
  mask 18
mathlib:
  G_make 57
  MYNAME 19
  G_myname() 66
O
Objective 1
P
parsing:
  G_parse_commands() 109, 111
PERMANENT 17, 19
  access permissions 21
  default window 19
  DEFAULT_WIND 19
  MYNAME 19, 66
  permuted index 257
point data:
  see: site files
Programmer:
  drivers 12
  GRASS 10
  system designer 13
Programming:
  compiling 55
  interface to cell files 24, 80
  interface to groups 46
  interface to site files 40, 105
  interface to the database 18, 68
  interface to vector files 32, 100, 123
programming:
  standards 6
  projection 26, 48
  GIS Library 79
  imagery (x,y) 26, 42, 48, 80
  longitude/latitude 5
  State Plane 26, 48, 80
  UTM 5, 26, 27, 42, 44, 48, 80, 161
  zone 26, 48
R
range file:
  see: cell files
raster files:
  see: cell files
Raster Graphics Library 12, 56, 147
  basic graphics 150
  colors 148
  connecting to the driver 148
  INDEX of routines 249
  INDEX, permuted 257
  LOADING the library 157
mouse 156
poly calls 152
raster calls 153
text 154
reclass files:
  see: cell files
resolution:
  cell file 27
  window 48
Rowio Library 173
  INDEX of routines 251
  INDEX, permuted 257
  LOADING the library 177
rowio.h 174
rowio.r:
  index of rowio_routines (Rowio Library) 251
R:
  index of R_routines (Raster Graphics Library) 249

S
Scope 2
search path 17, 20, 69, 71, 82, 83, 102
Segment Library 11, 56, 179
  INDEX of routines 253
  INDEX, permuted 257
  LOADING the library 186
segment.h 180
segment.r:
  index of segment_routines (Segment Library) 253
shell scripts 15, 229
  Bourne Shell 229
  Gsksh 230
  Gfindfile 231
site files 39
  file format 39
  opening 107
  programming interface 40, 105
  prompting for 106
  reading and writing 105
Standards:
  documentation 7
  programming 6
State Plane:
  see: projection
string routines:
  GIS Library 113
structures:
  P_AREA 129, 132
  struct Categories 91, 92, 93, 94, 105, 119
  struct Cell_head 76, 77, 78, 79, 80, 119, 160, 162
  struct Colors 94, 95, 96, 97, 98, 120, 167
  struct Command_keys 109, 110, 111, 113
  struct Control_Points 143, 144, 145
  struct dig_head 134
  struct History 98, 99, 121
  struct line_points 130, 131, 133, 135
  struct Map_info 128, 129, 130, 131, 132, 133
  struct Range 98, 100, 122
  struct Ref 139, 140, 141, 142, 144
subgroup 44
system():
  G_system() 116
T
target:
  see: group
Technology Transfer 2
termcap,termlib:
  Gmake 57
  Vask Library 187, 191
U
UNIX:
  access permissions 20
  environment 51
  fork() 84, 115
  gets() 117
  system() 116
User:
  general 9
UTM:
  see: projection
V
Vask Library 56, 187
  INDEX of routines 255
  INDEX, permuted 257
  LOADING the library 191
vector files 31
  see also: Dig Library
ascii format 32
  attribute file 35
  category file 36, 104
  category number 31
  digitizer registration 37
  finding 102
  import 38
  index and pointer file 37
  opening (new) 104
  opening (read) 103
  programming interface 32, 100, 123
  prompting for 101
  reading and writing 104
  topology rules 37