INTERAGENCY/INTERGOVERNMENTAL COORDINATION FOR ENVIRONMENTAL PLANNING (IICEP): SYSTEMS CONSIDERATIONS

by
R.D. Webster
D.E. Putnam
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.
DISCLAIMER NOTICE

THIS DOCUMENT IS BEST QUALITY PRACTICABLE. THE COPY FURNISHED TO DTIC CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.
The primary purpose of this report is to document the organization and command structure of a computerized system for providing access to information necessary for the Intergency/Intergovernmental Coordination for Environmental Planning (IICEP) requirements as set forth in Air Force Environmental Planning Bulletin 14. A secondary objective is to identify problems associated with the IICEP system's implementation and to recommend pertinent solutions. Preliminary data acquired...
Block 20 continued.

Air Force contractors were obtained and used as a basis for developing the software structure necessary to handle these data. This report describes IICEP and explains the development of the organization, structure, and software of the pilot computerized system. It will form the basis for evaluating the system and further clarifying the need for data base refinement and update.
FOREWORD

This project was performed for the Department of the Air Force Engineering and Services Center (AFESC), Tyndall AFB, FL, under Project Order Number S-79-26 dated 19 March 1979. CPT R. Hawkins was the project monitor.

The work was performed by the Environmental Division (EN), U.S. Army Construction Engineering Research Laboratory (CERL), Champaign, IL.

This research was made possible through the efforts of Air Force personnel and the scientists and engineers of CERL. Administrative support and counsel were provided by Dr. E.W. Novak, Acting Chief of EN.

COL L.J. Circeo is Commander and Director of CERL, and Dr. L.R. Shaffer is Technical Director.
## CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>DD FORM 1473</td>
<td>1</td>
</tr>
<tr>
<td>FOREWORD</td>
<td>3</td>
</tr>
<tr>
<td>1 INTRODUCTION</td>
<td>5</td>
</tr>
<tr>
<td>Background</td>
<td></td>
</tr>
<tr>
<td>Objective</td>
<td></td>
</tr>
<tr>
<td>Approach</td>
<td></td>
</tr>
<tr>
<td>2 THE IICEP PROGRAM</td>
<td>5</td>
</tr>
<tr>
<td>3 THE PILOT SYSTEM: ORGANIZATION AND STRUCTURE</td>
<td>6</td>
</tr>
<tr>
<td>4 COMMAND STRUCTURE</td>
<td>8</td>
</tr>
<tr>
<td>Selection Commands</td>
<td>2</td>
</tr>
<tr>
<td>Save and Restore Commands</td>
<td></td>
</tr>
<tr>
<td>List and Peek Commands</td>
<td></td>
</tr>
<tr>
<td>Help and Quit Commands</td>
<td></td>
</tr>
<tr>
<td>5 SUMMARY AND RECOMMENDATIONS</td>
<td>11</td>
</tr>
<tr>
<td>REFERENCES</td>
<td>12</td>
</tr>
<tr>
<td>APPENDIX A: Sample Data From IICEP Directory.</td>
<td>13</td>
</tr>
<tr>
<td>APPENDIX B: Software Descriptions</td>
<td>17</td>
</tr>
<tr>
<td>APPENDIX C: Source Code</td>
<td>21</td>
</tr>
<tr>
<td>DISTRIBUTION</td>
<td></td>
</tr>
</tbody>
</table>
INTERAGENCY/INTERGOVERNMENTAL COORDINATION FOR ENVIRONMENTAL PLANNING (IICEP): SYSTEMS CONSIDERATIONS

1 INTRODUCTION

Background

The U.S. Army Construction Engineering Research Laboratory (CERL) has maintained an extensive systems development program for Department of Defense (DOD) personnel to use in environmental assessment, planning, and management. These systems include the Environmental Technical Information System (ETIS) and its subsystems— the Environmental Impact Computer System (EICS), the Economic Impact Forecast System (EIFS), and the Computer-Aided Environmental Legislative Data System (CEILDS). These systems have been used extensively by both the Army and the Air Force. As a result of this cooperative effort, CERL has been tasked with analyzing new areas for assisting users and producing other systems which respond to these additional requirements and also function in the same interactive mode as ETIS. This mode is extremely beneficial from both developmental and operational standpoints. For instance, one new application for the ETIS type of system has been the review and systemization of the Air Force’s three-volume directory—Interagency/Intergovernmental Coordination for Environmental Planning (IICEP)—developed to insure adequate coordination of Air Force activities with state and local agencies responsible for environmental planning issues as required by Air Force Interim Planning Bulletin 14. Updating the information in the current directory is a problem. Responsibilities of the listed agencies change constantly; furthermore, the directory—filling three large binders—is physically awkward and inconvenient to update because changes must be mailed to all users. A computerized system could help remedy these difficulties. Implementation of IICEP as a new subsystem of ETIS will encourage maintenance of current directories by simplifying retrieval of the contacts.

Objective

The primary purpose of this research was to develop a pilot IICEP computerized system operating in interactive mode on the same host computer as ETIS and exhibiting the same user-oriented characteristics as the other ETIS subsystems. A secondary objective was to identify any problems associated with the IICEP system’s implementation under ETIS and to recommend solutions to these problems.

Approach

The documentation for IICEP was obtained from AFESC. The database was designed and developed, and an interactive retrieval program was designed and implemented.

2 THE IICEP PROGRAM

IICEP includes a three-volume directory of state environmental planning agencies designed for use by the three Air Force Regional Civil Engineers. Agencies located in all 50 states, Guam, and Puerto Rico are included. The listed agencies deal with issues from the following environmental categories:

1. General
2. Air Resources
3. Energy
4. Health and Safety
5. Land Use
6. Natural Resources
7. Noise
8. Socioeconomics
9. Solid Waste
10. Transportation
11. Water

Table 1 gives the subdivisions of the 11 major environmental categories.
Table 1
Categorical Breakout of IICEP

<table>
<thead>
<tr>
<th>1. General</th>
<th>6. Natural Resources</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coordination</td>
<td>Land Management and Grounds Maintenance</td>
</tr>
<tr>
<td>Environmental Quality</td>
<td>Fish and Wildlife</td>
</tr>
<tr>
<td>Environmental Impact Statements</td>
<td>Recreation</td>
</tr>
<tr>
<td>A-95 Clearinghouse</td>
<td>Forestry</td>
</tr>
<tr>
<td>Transportation</td>
<td>Archaeology and Historic Preservation</td>
</tr>
<tr>
<td></td>
<td>Flood Control</td>
</tr>
<tr>
<td></td>
<td>Oil and Gas</td>
</tr>
<tr>
<td>2. Air Resources</td>
<td></td>
</tr>
<tr>
<td>General</td>
<td></td>
</tr>
<tr>
<td>3. Energy</td>
<td></td>
</tr>
<tr>
<td>General</td>
<td></td>
</tr>
<tr>
<td>Facility Siting</td>
<td></td>
</tr>
<tr>
<td>4. Health and Safety</td>
<td></td>
</tr>
<tr>
<td>General</td>
<td></td>
</tr>
<tr>
<td>Civil Defense</td>
<td></td>
</tr>
<tr>
<td>Occupational Health</td>
<td></td>
</tr>
<tr>
<td>Pesticides</td>
<td></td>
</tr>
<tr>
<td>Radiation</td>
<td></td>
</tr>
<tr>
<td>Building Codes</td>
<td></td>
</tr>
<tr>
<td>Safety</td>
<td></td>
</tr>
<tr>
<td>5. Land Use</td>
<td></td>
</tr>
<tr>
<td>Planning</td>
<td></td>
</tr>
<tr>
<td>Agriculture</td>
<td></td>
</tr>
<tr>
<td>Coastal Zone Management</td>
<td></td>
</tr>
<tr>
<td>Minerals and Geology</td>
<td></td>
</tr>
<tr>
<td>7. Noise</td>
<td></td>
</tr>
<tr>
<td>General</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>8. Socioeconomics</td>
<td></td>
</tr>
<tr>
<td>Economic Development</td>
<td></td>
</tr>
<tr>
<td>Education</td>
<td></td>
</tr>
<tr>
<td>Housing</td>
<td></td>
</tr>
<tr>
<td>Local Government</td>
<td></td>
</tr>
<tr>
<td>Social Services</td>
<td></td>
</tr>
<tr>
<td>9. Solid Waste</td>
<td></td>
</tr>
<tr>
<td>General</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>10. Transportation</td>
<td></td>
</tr>
<tr>
<td>General</td>
<td></td>
</tr>
<tr>
<td>Aeronautics</td>
<td></td>
</tr>
<tr>
<td>Highways</td>
<td></td>
</tr>
<tr>
<td>11. Water</td>
<td></td>
</tr>
<tr>
<td>General</td>
<td></td>
</tr>
<tr>
<td>Water Resources Management</td>
<td></td>
</tr>
</tbody>
</table>

The IICEP directory contains information allowing the user to decide whether a particular environmental planning issue falls under the responsibility of an agency, and lists the point of contact at each agency. IICEP listings provide the agency's name and function, address, telephone number, and contact person, as shown by the examples in Appendix A. Interim Air Force Environmental Planning Bulletins 14 and 15 have clarified the general concepts of IICEP use and hierarchically organized the data originally contained in the directory. However, CERL's research on IICEP has indicated that updating the information is the most serious problem with the directory and the computerized system, primarily because the jurisdictions and duties of agencies identified in IICEP are vague and change frequently.

Interim Air Force Environmental Planning Bulletin 15, Volume II, lists environmental contacts for Federal agencies. The IICEP pilot program used these contacts as the basis for developing the computer-based retrieval system. The listings contained in the three volumes of contacts for the state environmental planning agencies will be incorporated in the IICEP computer program at a later date.

3 THE PILOT SYSTEM: ORGANIZATION AND STRUCTURE

For the three-volume directory of state environmental planning agencies, Volume II of Interim Air Force Environmental Planning Bulletin 15, and the computerized system, the information in IICEP is organized around a unit of data called a "contact." A given contact consists of information about some individual in the Government, and thus generally lists a name, title, address, and phone number. In addition, a contact includes keyword data which enable an IICEP user to locate the contacts of interest. The keywords currently fall into the following seven categories.
1. “agency”

This category consists of the abbreviated name of the 30 major agencies of the executive branch of the Federal Government. For example, “doc” and “doa” are agency keywords corresponding to the Departments of Commerce and Agriculture. All contacts belonging to the Department of Commerce include “doc” as one of their keywords. A complete list of the agencies and their corresponding keywords is given below:

- Advisory Council on Historic Preservation: ach
- Civil Aeronautics Board: cab
- Community Services Administration: csa
- Department of Agriculture: doa
- Department of Commerce: doc
- Department of Defense: dod
- Department of the Interior: doi
- Department of Justice: DOJ
- Department of Labor: dol
- Department of State: dos
- Department of Transportation: dot
- Environmental Protection Agency: epa
- Energy Research and Development Administration: erd
- Executive Office of the President: exo
- Farm Credit Administration: fca
- Federal Energy Administration: fea
- Federal Maritime Commission: fmc
- Federal Power Commission: fpc
- General Services Administration: gsa
- Department of Housing and Urban Development: hud
- National Aeronautics and Space Administration: nas
- Nuclear Regulatory Commission: nrc
- National Science Foundation: NSF
- National Transportation Safety Board: nts
- Small Business Administration: sba
- Smithsonian Institution: si
- Treasury Department: td
- Veterans Administration: va
- Water Resources Council: wrcc

2. “sub-agency”

This category corresponds to the next level below “agency” in the Government hierarchy. For example, “bureau of the census” is a subagency keyword occurring in some of the contacts belonging to the Department of Commerce.

3. “region”

This category consists of the names of the Federal regions. Unfortunately, many Federal agencies have adopted nonstandard regional divisions. Therefore, these keywords must be taken in the context of the appropriate Federal agency. The 10 standard Federal regions—“region 1” through “region 10”—are included in this category, as well as regions like the “atlanta region” of the Department of Commerce.

4. “state”

This category consists of the 50 state names and the term “us,” which refers to the whole United States. Users who seek contacts concerning some issue in an individual state should retrieve the contacts having that state as a keyword (such as Ohio, “oh,” or Alabama, “al”), as well as those having “us” as a keyword.

5. “topic”

This category consists of the 11 general topics listed in the following section.

6. “sub-topic”

This is a subcategory of the “topic” category. The following list gives various topics; the subtopics under each are indented.

- a. general
  - coordination
  - environmental quality
  - environmental impact statements/A-95 clearinghouse
  - transportation
- b. air resources
  - general air
- c. energy
  - general energy
  - facility siting
- d. health and safety
  - general health and safety
  - civil defense
  - occupational health
  - pesticides
  - radiation
  - building codes
  - safety
e. land use
   planning
   agricultural
   coastal zone management
   minerals and geology

f. natural resources
   land management and ground maintenance
   fish and wildlife
   recreation
   forestry
   archaeology and historic preservation
   flood control

g. noise
   general noise

h. socioeconomics
   economic development
   education
   housing
   local government
   social services

i. solid waste
   general solid

j. transportation
   aeronautics
   highways

k. water
   general
   water resources management

7. “program”

This category contains the names of the 26 Air Force programs.

air installation compatible use zone       aicu
air pollution                           ap
airfield and airspace criteria           aac
coastal zone management                 czm
compliance with pollution controls       cwpc
comprehensive plan                       cp
energy conservation                      ec
environmental impact assessments and    eias
   statements                            esc
explosive safety criteria                esc
fish and wildlife and endangered species fws
forest management                        fm
grazing and agricultural outleasing      gao
interagency/intergovernmental
   coordination (a-95)                    a-95
joint use of military airfields          juma
land management and landscape development lmld
military construction program (programs) mcpp
   (construction)                         mcpp
military family housing                   mfh
noise pollution                          np
outdoor recreation and cultural resources orcr
pesticide use and control                 puc
real property and acquisition             rpa
real property disposal                    rpd
reducing flight disturbances             rfd
solid waste                              sw
water pollution                           wp

4 COMMAND STRUCTURE

This chapter discusses the commands available to the IICEP user. Appendices B and C provide the software description and source code for IICEP, respectively, if further clarification is necessary. Table 2 lists and briefly describes IICEP commands.

Selection Commands

The IICEP system maintains a list of all contacts in the data base. By using keywords with the selection commands described below, a user can narrow this list to those contacts of interest.

1. “find”

The “find” command sets the list of contacts equal to those associated with a given keyword. For example, “find doc” sets the current list to contain all the contacts in the Department of Commerce. The find command can be used to retrieve a specific contact. For example, “find #162” brings the contact number 162 to the current list.

2. “and”

The “and” command limits the current list to contacts already in the list and associated with a given keyword. For example, suppose a user types

“find hew”

“and radiation”
Table 2
Pocket IICEP: Reference For Using IICEP Information Retrieval Program

<table>
<thead>
<tr>
<th>Command Format</th>
<th>Description</th>
<th>Categories</th>
<th>Keyword</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>find &lt;keyword&gt;</code></td>
<td>sets current subset of contacts equal to those associated with given keyword.</td>
<td>use with &quot;list&quot; and &quot;peek&quot; commands</td>
<td>use with &quot;find,&quot; &quot;and,&quot; &quot;or&quot; and &quot;except&quot; commands</td>
</tr>
<tr>
<td><code>and &lt;keyword&gt;</code></td>
<td>limits current subset to those associated with the given keyword.</td>
<td>agency</td>
<td>e.g., epa</td>
</tr>
<tr>
<td><code>or &lt;keyword&gt;</code></td>
<td>augments current subset with those associated with the given keyword.</td>
<td>sub-agency</td>
<td>e.g., enforcement</td>
</tr>
<tr>
<td><code>except &lt;keyword&gt;</code></td>
<td>removes from current subset those contacts associated with the given keyword.</td>
<td>region</td>
<td>e.g., region 6</td>
</tr>
<tr>
<td><code>save &lt;filename&gt;</code></td>
<td>saves current list in the specified file.</td>
<td>state</td>
<td>e.g., texas</td>
</tr>
<tr>
<td><code>restore &lt;filename&gt;</code></td>
<td>replaces current list with list of contacts stored in specified file.</td>
<td>topic</td>
<td>e.g., land use</td>
</tr>
<tr>
<td><code>restore</code></td>
<td>replaces current list with previous list.</td>
<td>sub-topic</td>
<td>e.g., planning</td>
</tr>
<tr>
<td><code>list &lt;category 1&gt; &lt;category 2&gt;</code></td>
<td>displays keywords associated with contacts in current list for the given category or categories. (IMPORTANT: if more than one category is to be specified, they should be ordered as follows: &lt;narrower&gt; &lt;broader&gt; e.g., list agency sub-agency)</td>
<td>program</td>
<td>e.g., acuz</td>
</tr>
<tr>
<td><code>peek &lt;category&gt;</code></td>
<td>invokes the editor on a copy of system file which contains keywords for given category. &quot;q&quot; returns to IICEP.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>show</code></td>
<td>displays contact number, keywords, name, title, address, phone number, and possibly comments for each contact in the current list.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>help &lt;term&gt;</code></td>
<td>prints message about the given term.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>help</code></td>
<td>prints summary of commands and references to more specific topics.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>quit</code></td>
<td>exits IICEP program.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The first command sets the current list to all the contacts in the Department of Health, Education, and Welfare. The second command limits that list to those concerned with radiation. The "and" command can be used to retrieve a specific contact. For example, "find # 234" "and # 678" bring the contact numbers 234 and 678 to the current list.

3. "or"

The "or" command augments the current list to include the contacts associated with a given keyword, and can be used to retrieve a specific contact. For example, suppose a user types "find us"
“or alabama”

The first command sets the current list to include all contacts having national jurisdiction. The second command expands this list to include contacts with jurisdiction only in Alabama and the Federal government. At this point, the user might wish to further modify the list, for example, by typing

“and radiation”

The three commands create a list of all contacts associated with radiation in Alabama.

4. “except”

The “except” command, which modifies the current list by excluding contacts associated with the given keyword, can be used to exclude specific contacts. For example,

“find radiation”

“except hew”

establishes a list of contacts—other than those in “hew”—associated with radiation. As the selection commands narrow the current list of contacts, the IICEP program reports the number of contacts in the list. When this number is small enough, the user may use the “show” command to see the actual contact data. This command is invoked by simply typing “show” at the keyboard. For each contact in the list, the contact number, keywords, name, title, address, phone number, and any comments are displayed on the terminal. The contact numbers displayed by the “show” command can be used as keywords with any of the selection commands. For example,

“find alabama”

“show”

“except #435”

“except #932”

might be used to eliminate contacts that are not of interest.

Save and Restore Commands

Once the user has narrowed contacts down to those of interest, he/she may wish to save this list for future reference. This can be done with the “save” command. For example, if the user types

“save testfile”

the current list is written to a file named “testfile.” The file name can be any character string up to 14 characters long. Lists saved in this way can be recovered later by typing

“restore <filename>”

For example,

“restore testfile”

would recover the list saved by the “save testfile” command. The “restore” command can also be used to recover from errors made during the selection process. The “restore” command used without any filename causes the previous list of contacts to be restored as the current list. Suppose a user types

“find texas”

“and alabama”

“restore”

The result of this series of commands is a current list of all contacts associated with “texas” and “alabama.” Since these resulted in a null set, the “restore” command reestablished only the set of “texas” contacts without reestablishing the entire search.

List and Peek Commands

The selection commands described earlier are useful only if the user knows which keywords to use. For example, “Bureau of Census” is a keyword, but “Census Bureau” is not. Therefore, two additional commands have been provided to furnish information on keywords. The “list” command displays all the keywords from a given category that apply to the current list of contacts. The number of contacts in the current list which corresponds to a given keyword is displayed alongside that keyword. For example,

“find radiation”

“list agency”
displays all the agencies which have contacts concerned with radiation.

DOC (1)  
EPA (5)  
ERD (3)  
HEW (1)  
NRC (21)

Each of these agencies is a "keyword" associated with the subtopic "radiation." If the selection commands have been used to modify the current list of contacts,

"find all"

can be used to set the current list to contain all the contacts in the data base. Thus,

"find all"

produces a list of all agencies in the data base. The "list" command can be invoked with more than one category name. If the user types

"list sub-agency agency"

the program responds with a list of subagencies and the agencies to which they belong. In general, this feature should be used only when the first category is a subcategory of each subsequent category. The program will respond in any case, but the information generated may be misleading. Suppose the user types

"list state agency"

In this case, the first category is not a subcategory of the second, and when the program responds with

"alabama"

"agency: doa"

it only means that "doa" is one of the agencies having a contact where Alabama is a keyword.

The other command designed to provide information on keywords is the "peek" command. This command invokes the operating system's editor on a copy of the IICEP system keyword file for a given category. As an example,

"peek sub-topic"

invokes the editor on the file of "sub-topic" keywords. Then,

"g/waste/p"

prints a list of all keyword terms which contain the word "waste." Finally,

"q"

quits the editor session and returns the user to the IICEP program.

Help and Quit Commands

A "help" command has been provided to help acclimate the user to using the system. If the user simply types

"help"

the system responds with a message that briefly summarizes the IICEP commands. For further information, the user can type

"help <term>"

and the system will respond with a message providing information about the given term. For example,

"help list"

gives a brief message concerning the use of the list command. Many help messages refer to other terms that can be used with the help command. By following these chains of reference with the "help" command, many questions can be answered without the aid of a manual. The last command that a user must know is the "quit" command. When the user types

"quit"

the IICEP session is ended.

5 SUMMARY AND RECOMMENDATIONS

This report has documented the organization and command structure of a pilot IICEP computerized system operating in interactive mode as a subsystem
of ETIS. The study also identified difficulties in implementing IICEP. The most serious problem with both the IICEP directory and system is the task of updating the information. The jurisdictions and duties of the various identified agencies are nebulous and change constantly.

Nonetheless, the IICEP program could be a valuable source of information to Air Force planners. If the information were maintained in a central database accessible by remote terminal and capable of supporting interactive usage, the system could be updated constantly with minimal effort, and users could access it from the central source (the interactive system). Incorporating IICEP into ETIS would allow the user to access IICEP's information without having to learn to operate a new system.

It is recommended that selected potential users (the Air Force Regional Civil Engineering offices, for example) use excerpts from this document to analyze the usefulness of the software produced under this research and development effort. The suggestions resulting from such a review could form the basis for modifying and improving the system.

Furthermore, an update procedure could be set up as part of an effort already contemplated for the ETIS operational component now being established for Army users. The additional update of the IICEP data could be integrated into existing procedures for CELDS and EIFS with little increase in long-term operational costs.

REFERENCES


APPENDIX A:
SAMPLE DATA FROM IICEP DIRECTORY

STANDARD FEDERAL REGION V

A. Illinois
   1. General

   a. Agency

      Environmental Protection Agency
      2200 Churchill Road
      Springfield 62706

      (217) 782-3397

      Richard H. Briceland, Director

      Function—The Agency coordinates programs for air quality, noise, solid waste and water quality.

      State Laws—The Agency is established by S.L., Chapter III 1/2, Section 1004.

      Federal Laws—(See functional headings.)

   b. Agency

      Bureau of the Budget
      103 State House
      Springfield 62706

      (217) 782-4520

      Leonard Schaeffer, Director

      Function—The Bureau is responsible for reviewing federally financed projects in accordance with
      A-95 procedures.

      State Laws—None identified.

      Federal Laws—The Bureau coordinates state review of federally assisted projects pursuant to OMB
      Circular No. A-95.
c. Agency

Department of Transportation
2300 S. Dirksen Parkway
Springfield 62706

(217) 782-5597

Langhorne Bond, Secretary

Function—The Department plans and develops state transportation systems. It develops and implements mass transit programs, plans airports, promotes transportation safety and constructs and maintains highways.

State Laws—The Department is established by the Civil Administrative Code of 1917.

Federal Laws—(See functional headings.)

2. Air Resources

a. Agency

Division of Air Pollution Control
Environmental Protection Agency
2200 Churchill Road
Springfield 62706

(217) 782-6514

John Moore, Division Director

Function—The Division administers and enforces state air pollution laws and reviews applications for permits.

State Laws—The Division is established under the Environmental Protection Act of 1970, as amended. The Division operates under the following laws and regulations: Stationary Sources Standards, 1972, as amended; Air Quality Standards, 1973; Episodes Regulations, 1976; Open Burning Regulations, 1971, as amended; and Odors Regulations, 1972.

Function—The Division administers state responsibilities under the Clean Air Act.
b. Agency

Pollution Control Board
309 West Washington Street
Chicago 60606

(312) 793-3620

Jacob D. Dumelle, Board Chairman

Function—The Board establishes air quality standards and regulations.

State Laws—The Board is established under the Environmental Protection Act of 1970, as amended. The Board operates under the following laws and regulations: General Air Pollution Regulations, 1972, as amended; Stationary Sources Standards, 1972, as amended; Air Quality Standards, 1973; Episodes Regulations, 1976; Open Burning Regulations, 1971, as amended; and Odors Regulations, 1972.

Federal Laws—The Board administers state responsibilities under the Clean Air Act.

3. Energy

a. Agency

Division of Energy
Department of Business and Economic Development
222 South College Avenue
Springfield 62702

(217) 782-5784

Sidney M. Marder, Director

Function—The Division conducts energy conservation programs and coordinates energy research within the state. The Division administers fuel allocation programs.

State Laws—The Division is organized under 78-1125, S.L. 1974.

b. Agency

Interstate Oil Compact Commission
(See Interstate Agency Appendix for details.)
APPENDIX B:

SOFTWARE DESCRIPTIONS

Hashing Subsystem

The hashing subsystem provides a means of looking up character strings in files of keywords. If a string is present in these titles, the lookup mechanism returns identifying data, specifying:

1. The number of keyword file in which the string is found
2. The keyword's number within each file
3. The byte offset of the keyword within each file.

The keyword files are specially formatted text files which are named with some fixed prefix such as "key," followed by a numeric string. Generally, it is best to organize keywords into coherent groups according to file numbers associated with each keyword type. For example, in the IICEP system, one category consists of state names, while another consists of agency names, so these categories should have different file numbers. In the files themselves, keywords are marked by a "#" character in column 1, followed by the keyword string, followed by a terminating ":" character. Characters following the "" character and characters on subsequent lines are not part of the keyword string. This provides space for comments about the keyword. The next "#" character found in column 1 marks the end of the comments and the beginning of a new keyword.

Thus, the "hash" program sets up a hashtable which allows keyword data to be retrieved, but in order to use the "hash" program, another file must be prepared which itself names the keyword files. This file simply lists one keyword file name per line; the following is a current list of IICEP keyword files.

| key.0 | key.5 |
| key.1 | key.6 |
| key.2 | key.7 |
| key.3 |
| key.4 |

Then the "hash" program is invoked by

```
"hash <file list>"
```

where "<file list>" is the name of the file discussed above. The keyword files in the <file list> file are opened and read in order; each is scanned for keywords, and identifying data on each keyword are written to a temporary file. This temporary file is an array of struct elements defined as follows:

```
struct marker /* word marker structure layout */
int file;    /* keyword file number */
int idnum;   /* rel word # within file */
long beginbyte; /* byte offset of work in file */
int hashv[3]; /* hash value */
```

Next, the "hash" program calls a subroutine named "maketable," which rearranges the contents of the temporary file into a hashtable. The "hashtable" file is also an array of struct elements defined as above; it is about half-empty at this point in the "hash" program, with the empty slots marked by setting the "file" field equal to -1. Slots occupied by struct elements corresponding to keywords from the keyword files are positioned as follows:

1. The total number of slots in the "hashtable" file minus a maximum overflow allowance defines a modulus.
2. The hash value included in a word marker struct is used to define a long integer.
3. The remainder of the long integer divided by the modulus yields a trial position in the hashtable.
4. The marker struct element is inserted into the first empty slot following the trial position.

The reader should consult the "maketable" subroutine source code (Appendix C) to see the actual mechanics of the temporary file of word markers reorganization into the "hashtable" file.

Once the "hashtable" file has been created, the "lookup" subroutine can be called from a "C" program to retrieve identifying data on any character string. The "lookup" routine computes a trial
position in the hashtable just as in steps 1, 2, and 3 above. Then the hashtable is searched until the first empty slot is encountered. The marker struct elements matching the given keyword are passed back to the calling procedure.

Setup Subsystem

The “setup” program scans files of IICEP data on contacts to prepare for retrieval of this data by the IICEP information retrieval program. The “setup” program is invoked by typing

```
"setup <file list>"
```

The argument “<file list>” is a file which names the IICEP files containing contact data. These names should be listed in the “<file list>” file, one per line: for example,

- pc.0
- pc.62
- pc.125
- pc.181
- pc.241
- pc.309
- pc.377
- pc.444
- pc.518
- pc.595
- pc.678
- pc.764
- pc.841
- pc.937

The files of contact data are named “pc.x,” where “x” stands for the contact number of the first contact in the file. It is important that the files listed in “pc.list” be ordered so that contacts are encountered in strictly increasing order. Each file contains data on one or more contacts, and each contact has the format given below:

```
# <contact number>
<category number> : <keyword string>
<category number> : <keyword string>

&
<text data, including name, title, address, phone, comments>
```

In the format description above, the <contact number> field is a numeric string giving the number of the particular contact. Contacts are numbered beginning with zero and must be arranged in increasing order. Gaps are permissible, but tend to slow down the retrieval of data.

In the next section, each line gives a keyword string and the category (e.g., “agency,” “region”) to which it belongs. Presumably, the given string will be found in the keyword file numbered with the given category number. For example, the string for category two will be found in the keyword file for category two.

The latter section must be terminated by a line consisting of a single “&” character.

Succeeding lines contain text data about the contact; the next line containing a “#” character in the first column marks the beginning of a new contact.

The following is an example of data for a specific contact:

```
# 61
0: doc
```
1: office of the secretary

4: socioeconomics

5: economic development

3: us

&

Jerry Jasinowski, Assistant Secretary for Policy
(8-77)

14th Street, N.W.

Washington, DC 20203

(202) 377-2113

The "setup" program opens and reads the data files in the order they are listed in the "pclist" file. As "setup" scans the data, messages are printed, if

1. The numbering of the contacts is not consecutive.

2. A keyword is not present in the alleged keyword file.

3. The "&" line ending the keyword section is missing.

Under any of these conditions, the line number in the file is printed along with an appropriate message.

As the "setup" program scans the data files, the keyword data are digested and written to special files that will later be employed by the retrieval program. For each keyword category, a file named "pekey.X" is created. The "X" stands for the number of the corresponding keyword file. Each file lists the id numbers of the keywords pertaining to the contacts in the data files. A "pekey" file can be thought of as an array of integers. If the keyword id numbers for a given contact have no keywords from a given category, or if there is a gap in the contact numbers, then the -1 entry is still present to signify an empty list of keywords.

The "setup" program also creates an index file as it scans the contact data. Each entry in the index file contains the location of a given contact. This location consists of

1. The number of the "pc" file in which the contact occurs

2. The byte offset of the beginning of the contact

3. The byte offset of the text data for the contact.

If there is a gap in the numbering of the contacts, the missing entries in the index are marked with a -1 in each of the above three fields.

The Retrieval Program

When the "hash" and "setup" programs have been successfully run, the retrieval program "iicep" can be used. A complete description of the retrieval commands can be found in Chapter 4. The following discussion focuses both on the files required by the "iicep" program and on their functions. Five families of data files are used by the "iicep" program:

1. The "key." files containing keywords and comments

2. The "hashtable" of pointers to the "key." files

3. The "pc." files of textual contact data

4. The "pcndx" file indexing the "pc." files

5. The "pckey" files of keyword id numbers.

When the "iicep" program in invoked, a subroutine named "initlist" is called to construct a list of all the contacts in the data base. This is done by reading the "pcndx" file and noting those entries not marked as being empty. Thus, gaps in the sequence of contacts are detected and left out of the list of contact numbers. The list of contact numbers is represented as an array of integer entries and written to a disk file. An entry of -1 marks the end of the list.

The selection commands "find," "and," "or," and "except" modify this list. Each of these commands takes a keyword as an argument. The hashing lookup mechanism converts the keyword string into data specifying the appropriate keyword category and id number within that category. Next, the appropriate "pekey." file is scanned by the "keypcs"
routine to list those contact numbers in which the given keyword appears. Finally, the "bool" subroutine is called to perform the appropriate logical operation on this list of contact numbers and the previous list of contact numbers.

The "list" command scans the current list of contact numbers and the appropriate "pkey." file to determine which keyword id numbers from a given category are associated with the contacts in the current list. The result is a list of keyword id numbers and the number of contacts in which they appeared. Also listed is the number of a specific contact and the location within that contact where the keyword appeared. When this list is completed, the "pcndx" file is used to locate the contacts where the keywords are listed. The "pc." files are then opened and read in order to retrieve the actual keyword strings so that they can be printed to the user's terminal.

The "show" command runs through the current list of contact numbers and displays the data for each contact. This is done by finding the location of the contact data in the "pcndx" file and then reading the data from the appropriate "pc." file.

The "help" command uses the hashing lookup mechanism to convert a character string into data specifying the category number and byte offset of the string within the given keyword file. The keyword file is then read, and any comments following the keyword string in that file are displayed on the terminal.
APPENDIX C:
SOURCE CODE

```c
/* this file is included by both ice.c and select.c */
/* it defines command numbers for switch statements */
#define FIND 0
#define AND 1
#define JR 2
#define EXCEPT 3
#define SAVE 4
#define RESTORE 5
#define SHOW 6
#define QUIT 7
#define PEEK 8
#define HELP 9
#define LIST 10
```
char *keynames[]
{
    "agency",
    "sub-agency",
    "region",
    "state",
    "topic",
    "sub-topic",
    "program",
    0,
};


1  #define  KEYNAME  "/usr/tmp/iicep/data/key."
2  #define  HASHTBL  "/usr/tmp/iicep/reference/hashtable"
3  #define  PCKEY  "/usr/tmp/iicep/reference/pckey."
4  #define  PCNDX  "/usr/tmp/iicep/reference/pcndx"
5  #define  PC  "/usr/tmp/iicep/data/pc."
6
7  #define  NUMTYPES  7
8
9  #define  MESSAGES  7
10
11  #define  ALL  "all"
#define BUFRSZ 256

struct keybufr
{
    int descrp; /* file descriptor */
    int *next; /* next empty slot in bufr */
    int *endbufr; /* marks end of bufr */
    int bufr[BUFRSZ]; /* buffer for keyword id numbers */
};

struct marker /* word marker structure layout */
{
    int file;   /* keyword file number */
    int icnum; /* rel word # within file */
    long lenbyte; /* byte offset of word in file */
    int hashv[3]; /* hash value */
};

struct qetbuf /* buffered input by line */
{
    int filedes; /* file descriptor of the given file */
    int nleft; /* # of chars left in buffer */
    char *nextp; /* pointer to next char in buffer */
    char buff[512]; /* for buffered reads */
};

#define HDXSIZ 256

struct ndx
{
    int filenum; /* number of file where entry occurs */
    long keylines; /* byte offset of keyline section */
    long datalines; /* byte offset of data section */
};

struct ndxbufr
{
    int fndxds; /* descriptor of ndx file */
    struct ndx *nextndx; /* next open slot in buffer */
    struct ndx *endndx; /* marks end of buffer */
    struct ndx bufnds[HDXSIZ]; /* buffer for index entries */
};

struct keycheck
{
    int keycount; /* number of occurrences */
    int pcnum; /* id of pc where found */
    int keynum; /* number of key in cc keylines */
}
struct keymarker
{
    int keytype;
    char *keystring;
};

#define MAXKEYS 50
#define MAXCHARS 1224

struct keydata
{
    int totkeys;
    struct keymarker keyptr[MAXKEYS];
    char keybuf[MAXCHARS];
};
NAME:
SICEP (main program)

FUNCTION:
Implement the commands of the IICEP system.

ALGORITHM:
The program begins by performing certain system initialization tasks. In particular, "iam()" is called to select a unique name for the process and scratch files are created. The scratch files are used for listing the current and previous lists of contacts and they are initialized to list all the contacts in the database.

When the above operations are concluded, the program enters the main command loop where the user is prompted to enter a command and control is transferred to the sub-routine appropriate to executing that command.

CALLS:
iam()
corcat()
initlist()
resp()
copy()
execute()
table()
select()
save()
restore()
show()
peek()
help()
keyword()
list()

Also, Unix routines:
printf()
exit()
signal()
creat()
error()
open()
creat()
setexit()
unlink()

HISTORY:
written by Dan Putnam - spring 1979.
#include "structdefs.i" /* defines getbuf */
#include "commanddefs.i" /* defines commands */
#include "keynames.i" /* needed for KEYNAME only!! -Dan Putnam */
char *commtbl[3] /* command names, must be consistent with commandefs.i */
{
    "find", 
    "find", 
    "except", 
    "show", 
    "quit", 
    "peek", 
    "help", 
    "list", 
    0,
};

struct getbuf bufin; /* for line oriented input */
char linet[256]; /* used with bufin */
int count; /* character count returned from getl */
int fidscratch[2]; /* file descriptors */
int phase C; /* used to alternate between files */
char *keyprefix KEYNAME; /* name of the keyword files */

main(argc,argv)

int argc;
char **argv;
{
    int reset(); /* used with setexit to handle breaks */
    char buffer[256]; /* buffer for user response */
    char comm[256]; /* buffer for command string */
    char *src; /* utility pointer used with copy() */
    char *dst; /* utility pointer used with copy() */
    char me[10]; /* buffer for my unique name */
    char scratch[2][256]; /* names of scratch files */
    int opcode; /* command number */
    int quitflag; /* loop controls main command loop */
    int num; /* number of pcs returned from initlist */
    struct marker *keyword(); /* returns pointer to keyword data */
    signal(2, 1); /* ignore interrupts */
printf("Welcome to the LICEP information retrieval program\n");
printf("For help, type 'help licep commands'\n");

/**************** create scratch files ********/
/**************** we will need to read and write on them, so close and reopen */

if(iam(me) < 0)
{
    printf("can't create unique name, aborting\n");
    exit();
}
concat(me, "@scratch", scratch[0]);
concat(me, "@scratch", scratch[1]);

fidscratch[0] = creat(scratch[0], 0666);
fidscratch[1] = creat(scratch[1], 0666);

if( fidscratch[0] < 0 || fidscratch[1] < 0 )
{
    perror("can't creat, aborting\n");
    exit();
}

close( fidscratch[0] );
close( fidscratch[1] );

fidscratch[0] = open( scratch[0], 2);
fidscratch[1] = open( scratch[1], 2);

if( fidscratch[0] < 0 || fidscratch[1] < 0 )
{
    perror("can't open, aborting\n");
    exit();
}

/**************** initialize scratch files to list all pcs */
num = initlist( fidscratch[0] );
printf("%d contacts in current list\n", num);
copyfile( fidscratch[0], fidscratch[1] );

/**************** this is the main command loop */
for(quitflag = 0; quitflag == 0;)
{
    setexit();
    signal(2, reset);
printf("What next?\n");
resp(buffer);  /* get user response */
src = buffer;
dst = buffer;
copy(&src, &dst, 0, sizeof(buffer));  /* omit extra blanks */

/**** see if user wants to execute a Unix command */
if (*buffer == '!') {
    execute(buffer + 1);
    continue;  /* go back to top of command loop */
}

/**** copy characters into command string */
src = buffer;
dst = comm;
copy(&src, &dst, buffer, sizeof(comm));

if (opcode = table(comm, commtbl) == -1) {
    printf("*s' not a command
", comm);
    continue;
}

/**** copy() has left src pointing at command argument string */
switch(opcode)

    /**** these commands select the current pc list */
    case FIND:
    case AND:
    case OR:
    case EXCEPT:
        signal(2, 1);  /* ignore interrupts here */
        select(opcode, src);
        break;

    case SAVE:
        signal(2, 1);  /* ignore interrupts here */
        save(src);
        break;
case RESTORE:
    signal( 2, 1);       /* ignore interrupts here */
    restore( src );
    break;

case SHOW:
    /* don't ignore interrupts here */
    show(fidscratch[ phase ], src);
    break;

case QUIT:
    quitflag = 1;       /* this will get us out of loop */
    break;

case PEEK:
    peek( src );
    break;

case HELP:
    help(O, keyword( src ));       /* L = standard output */
    break;

case LIST:
    list( fidscratch[ phase ], src);
    break;

default:
    printf("'s is not yet implemented", buffer);
    break;

        }

    unlink( me );
    unlink( scratch[0] );
    unlink( scratch[1] );
    }
NAME: eetdata()

FUNCTION:
Read the keyword lines for a contact into a "keydata" struct
so that they can be more easily referenced.

CALLING SEQUENCE:

int pcid
int fidpc
long offset
struct keydata *pcdata

PARAMETERS:

pcid Accession number of the point of contact whose
data is being read.

fidpc File descriptor of the contact data file where the
data for the given pc resides.

offset Byte offset of the data in the given file.

pcdata Points to the structure which gets the key data
to be read from the file.

RETURNS:
nothing.

ALGORITHM:
The routine seeks into the file and reads the header.
If these operations are successful, the routine enters
a loop, and reads the keyword lines into the "keydata"
struct indicated by "pcdata".

CALLS:
seekl()
getl()
copy()
Also Unix routines:
printl()

CALLED BY:
HISTORY:
written by Dan Putnam - spring 1979.

#include "structdefs.i" /* define getlbuf struct */

struct getlbuf buf;

getdata( pcid, fidpc, offset, pcdata)
int pcid;      /* number of source permit */
int fidpc;     /* file descriptor of pc data file */
long offset;   /* byte offset of data for given pc */
struct keydata *pcdata; /* gets lines of keyword data */
{
  int keynumber;    /* counts number of keys in pc */
  struct keymarker *markptr; /* points thru keyptrs of pc data */
  char *bufptr;     /* points thru keybuf of pcdata */
  char *endptr;     /* points off end of keybuf */
  char *src;        /* utility pointer used with copy() */
  char line[256];  /* gets line lines from getl() */
  int nlytes;      /* returned from getl */
  struct getlbuf bufr; /* used by getl() */

  bufr fiddes = fidpc;
  bufr nleft = 0;

  if( offset < 0 || seekl( fidpc, offset) < 0 )
    {
      printf("can't seek to data on pc %d\n", pcid);
      return;
    }

  if( (nbytes = getl( line, &bufr)) <= 0 )
    {
      printf("can't find data on pc %d\n", pcid);
      return;
    }

  keynumber = 0;
  bufptr = pcdata -> keybuf;

  ...
markptr = pcoata -> keyptr;
while(( nbytes = get(( line, &bufr)) > 0 )
    ( line[ nbytes - 1 ] = 0; )/* replace '\n' by null */
)
markptr = pcoata -> keyptr;
while(( nbytes = get(( line, &bufr)) > 0 )
    ( line[ nbytes - 1 ] = 0; )/* replace '\n' by null */
)
markptr = pcoata -> keyptr;
while(( nbytes = get(( line, &bufr)) > 0 )
    ( line[ nbytes - 1 ] = 0; )/* replace '\n' by null */
)
markptr = pcoata -> keyptr;
while(( nbytes = get(( line, &bufr)) > 0 )
    ( line[ nbytes - 1 ] = 0; )/* replace '\n' by null */
)
markptr = pcoata -> keyptr;
NAME:
getndx()

FUNCTION:
Find the location of the data for a given contact, and return a file descriptor for the data file.

CALLING SEQUENCE:

int pcid
struct ndx *pc_ptr

PARAETERS:

pcid The accession number of the contact of interest.

pc_ptr Points to the index struct to be filled in with the data giving the location of the given contact.

RETURNS:

Returns a file descriptor of the contact data file containing the given contact.

ALGORITHM:

This routine may be interrupted if the user hits the "rub-out" key. If this happens, the index file won't get closed. To handle this problem, the descriptor is stored in a static variable. The routine begins by examining this variable to see if it is non-zero. If so, the file is closed and the descriptor is set to zero to mark the file as being closed.

The routine next opens the index file to read the index struct giving the location data for the given contact. Then, the data file containing the given contact is opened and the file descriptor is returned.

CALLS:

Various Unix routines.

CALLED BY:

show()
57 list();
58
59 HIST0R¥:
60
61 Adapted from the "getdøx()" routine of the PAMS system - spring 1975.
62
63 **********************************************
64
65 #include "structdefs.i"    /* defines pc index structure */
66 #include "params.i"        /* defines PCDX */
67
68 getndx(pcid, pc_ptr)
69 int pcid;
70 struct ndx *pc_ptr;
71 {
72     char pcfile[10];   /* name of pc file */
73     int file;         /* pc file number where pc is */
74     long offset;     /* byte offset into a file */
75     int fidpc;        /* descriptor of pc file */
76     static int fidndx;  /* descriptor for pc index file */
77
78     /******** make sure we close olc files before using ********/
79
80     if( fidndx != -1 )
81         {              /* and mark it as closed */
82             close( fidndx );
83             fidndx = -1;
84         }
85
86     if( (fidndx = open("PCDX", O)) < 0 )
87         {              /* mark it as closed */
88             perror("getndx can't open pcdx file");
89             return(-1);
90         }
91
92     offset = pcid;
93     offset += sizeof( *pc_ptr );
94     if( seek( fidndx, offset, 0 ) < 0 )
95         /* mark it as closed */
96         {              /* return(-1) */
97             perror("getndx can't seek into pc index");
98             return(-1);
99         }
100     if( (read( fidndx, pc_ptr, sizeof(*pc_ptr) ) < sizeof(*pc_ptr) )
101         {              /* mark it as closed */
102             printf("can't read pc index file\n");
103             return(-1);
104         }
105     close( fidndx );
106     fidndx = -1;
107
108     return(0);
109
110}
```c
113    file = pc_ptr -> filenum; /* this is pc file number */
114    concat(PC, locv(0, file), pcfile);
116
117    if( (fidpc = open(pcfile, O)) < 0 )
118    {
119        perror("getndx can't open pc file");
120    }
121
122    return( fidpc );
123
124)
```
NAME: initlist()

FUNCTION: Initialize a file to list all the contacts in the database.

CALLING SEQUENCE:

int fidpc
int initlist()

PARAMETERS:

fidpc file descriptor of the output list of accession numbers.

RETURNS:

Returns the number of accession numbers in the output list.

ALGORITHM:
The routine opens the index file and reads from it in a loop. Empty index structs are marked by having their "filename" fields set to -1. Whenever a struct is encountered that is not empty, the corresponding accession number is inserted into the output buffer.

CALLS:

Unix routines:

seek()
open()
perror()
exit()
read()
write()
close()

CALLED BY:

liceq (main program)
select()

HISTORY:

written by Dan Putnam - spring 1979.
```c
#include "structdefs.i"
#include "params.i"

#define PCSIZ 256
#define NDXSIZ 256

int initlist( fidpc )
    /* descriptor of output list of pc ids */
    { 
        int fidpc;
        /* descriptor of output list of pc ids */
        int *pcbuf[PCSIZ];
        /* output buffer for list of pc ids */
        int *pcptr;
        /* points thru pcbuf */
        int fidndx;
        /* descriptor of input index file */
        struct ndx ndxbuf[NDXSIZ];
        /* input buffer for index file */
        struct ndx *ndxptr;
        /* points thru ndxbuf */
        int pcid;
        /* id of current pc in index list */
        int countpc;
        /* counts number of pc's in index */
        /* be sure to start at beginning */
        seek( fidpc, 0, 0 );
        if( (fidndx = open( "PCNdxt", C )) < 0 )
            { 
                perror( "initlist can't open pcndx" );
                exit();
            }
        pcid = -1;
        /* pc ids begin with zero */
        countpc = 0;
        /* no pc's so far */
        pcptr = pcbuf;
        while( ( n = read( fidndx, ndxbuf, sizeof( ndxbuf ) ) ) > 0 )
            { 
                n = sizeof( *ndxbuf );
                /* n = # of ndx entries */
                n*ndxptr = ndxbuf;
                do
                    { 
                        if( ndxptr++ -> filenum != -1 )
                            { 
                                /******* got one! */
                                countpc++;
                                *pcptr++ = pcid;
                                if( pcptr >= pcbuf[PCSIZ] )
                                    { 
                                        if( write( fidpc, pcbuf, sizeof(pcbuf)) < sizeof(pcbuf) )
                                            { 
                                                perror( "initlist can't write pc id's" );
                                                exit();
                                            } 
                                        pcptr = pcbuf;
                                    } 
                    } 
                } 
```
pcid++;            /* bump pcid to id of next incex entry */
}

while(--n);        /* count down on number of entries */

/******** flush remaining pc id's in pcbuf */

*pcptr++ = -1;      /* null terminate list */

n = (pcptr - pcbuf) * 2;

if (write(fidpc, pcbuf, n) < n) {
    perror("inittist can't flush pc list");
    exit();
}

return(countpc);

}
NAME:  
keypcs()

FUNCTION:  
Make a list of the contacts associated with a given keyword.

CALLING SEQUENCE:

int type  
int fidin  
int fidout  
int keypcs()

PARAMETERS:

type     Id number of the given keyword.
fidin    File descriptor of the "pkey." file for the category  
of the given keyword.
fidout   File descriptor for the output file which will list  
the accession numbers of the contacts associated  
with the given keyword.

RETURNS:

Returns the number of contacts associated with the given keyword.

ALGORITHM:

The routine reads through the "pkey." file given by the "fidin"  
file descriptor. Each entry in the file bumps the current  
PC number by one in order to keep track of which PC accession  
number is current. When an entry matches the "type" argument,  
the current PC accession number is inserted into the output  
buffer. The "previous" variable keeps track of the last accession  
number to be put into the output list, and the routine checks to  
be sure that no accession number is inserted twice. This step  
is necessary in case a keyword has been entered twice in the same  
contact in the database.

CALLS:

Unix routines:

seek()  
read()  
write()  

CALLED BY:

40
```c
select()

HISTORY:
written by Dan Putnam - spring 1979.

#define INSIZ 1224 /* size of input buffer */
#define OUTSIZ 256 /* size of output buffer */
kpcs(type, tidin, fidout)  
    int type;        /* locate pcs with this key */
    int fidin;       /* descriptor for pc key file */
    int fidout;      /* descriptor for qualified pc file */
{
    int count;       /* counts number of qualified pcs */
    int inbuf[INSIZ];    /* input buffer for pc key file */
    int outbuf[OUTSIZ]; /* output buffer for qualified pcs */
    int *outptr;       /* points to next open slot in cutbuf */
    int pcid;          /* current pc id number */
    int inprevious;    /* id number of last pc put in cutbuf */
    register int n;    /* for loop counting thru pc key list */
    register int *idptr; /* grabs id numbers out of list */
    register int idkey; /* equals id of current key in list */

    count = 0;
    pcid = 0;
    previous = -1;
    outptr = outbuf;
    seek(fidin, 0, 0);        /* start at beginning */
    seek(fidout, 0, 0);       /* start at beginning */
    while((n = read(fidin, inbuf, INSIZ * 2)) > 0)
    {
        n /= 2;        /* n = # of entries in buffer */
        idptr = inbuf;
        do
        {
            if( (idkey = *idptr++) == -1 )
                pcid++;
            else
            {
                if( idkey == type && pcid > previous )
                { /****** put one! */
                    count++;
                }
            }
        } while(n--);
    }
}
```
previous = pcid;  /* to avoid repetition */
*outptr++ = pcid;

if(outptr >= outfbuf[OUTSIZE])
{
    write(fidout,outbuf,(outptr - outfbuf) * 2);
    outptr = outfbuf;
}

while(--n);

/*** terminate list and write out */
*outptr++ = -1;
write(fidout,outbuf,(outptr - outfbuf) * 2);
return(count);
NAME:
list()

FUNCTION:
Implement the "list" command of IICEF.

CALLING SEQUENCE:
int fidpclist
char *arg

PARAMETERS:
fidpclist: file descriptor of the current list of contact accession numbers.
arg: Points to string containing keyword category names that are to be listed.

RETURNS:
nothing.

ALGORITHM:
The List command can be interrupted by the user by hitting the "rubout" key. This operation can leave open files. To deal with this problem, file descriptors are stored in static variables. The routine begins by examining these variables to see if they are non-zero. If so, the files are closed and the descriptors are set to zero to mark the files as being closed.

The next operation that is performed is to parse the argument string, given by "arg". The string is broken down into sub-strings delimited by blanks. The "keynames" array is searched to see if these sub-strings are indeed valid keyword category names. If so, the index in the array which matches a sub-string is saved to identify the category.

The first category named in the argument string drives the operation of the list command.

The routine loops to pick up the keys from the first argument category that occur in the current list of contacts. This is accomplished through the call to "listcheck()" which drives the loop. This sub-routine fills out the "checklist" array which keeps track of:

1. The number of contacts in the current list which contain a given keyword.
2. The accession number of one of the contacts that contains a given keyword.

3. The number of keywords from the given category which precede the keyword in the contact named by item 2 above.

Items 2 and 3 provide a way of recovering a keyword in order to print it. Number 2 gives a contact where it occurs and number 3 indicates which of the keys it is. Since the "listcheck()" array has limited length, it covers just a subrange of the possible keyword id numbers on each loop iteration. On each call, "listcheck()" returns the smallest id number of a keyword occurring in the current list of contacts which has not yet been considered in the "checklist" array. This provides a lower bound for the next iteration.

Once the "checklist array has been filled out for an iteration, the routine prints out the keyword data for the checked keys. If the "checklist" struct for a key has not been checked, then nothing is printed. Otherwise, the contact data is read and the given keyword string is printed as it appears in the contact cata file. If any other categories were named in the argument list, then the keywords from those categories which occur in the contact data are also printed.

CALLS:

copy()
table()
concat()

tischeck()

g:tnax()
e:tdate()

Also, Unix routines:
close()
printf()
locv()
open()
perror()

CALLED BY:

show()

HISTORY:

written by Dan Putnam - spring 1979.

*******************************

#include "structdefs.i"

#include "params.i"
#define CHECKSIZ 256

list(fidnclist, arg) /* file descriptor of current pc list */
char argv[]; /* contains arguments of list command */
{
    struct keycheck checklist[CHECKSIZ]; /* marks found keys */
    struct keycheck *checkptr; /* points thru checklist */
    int type; /* number of chosen category */
    int argtype; /* type of other arguments */
    int i; /* number of key among keys of a pc */
    int accurs; /* counts keys of a given type */
    extern char *keynames[]; /* names of keyword categories */
    struct keydata *pndata; /* picks up keyword lines for pcs */
    struct keymarker *markptr; /* points thru keymarkers in pndata */
    int argc[20]; /* argument numbers of show */
    int invalid; /* flag = 1 if an argument is invalid */
    int nargs; /* loop control: counts arguments */
    int num; /* id number of an argument */
    char reply[256]; /* gets user response to prompt */
    int hi; /* high id in range passed to listcheck */
    int lu; /* low id in range passed to listcheck */
    char *string; /* points to individual arg strings */
    char *src; /* utility pointer used with copy() */
    char *dst; /* utility pointer used with copy() */
    char *key; /* points to keyword string in pndata */
    register int pcid; /* id number of contact in lists */
    struct nda pcindex; /* offsets of data in pc file */
    char *pckeyfile[100]; /* for building pckey filename */
    static int fidpckey; /* file descriptor for pc data file */
    static int fidc; /* file descriptor for pckey data file */
}

/******** make sure file descriptors are closed before using again */

if ( fidpckey != 0 )
{
    close( fidpckey );
    fidpckey = 0;
}

if ( fidpdc != 0 )
{
    close( fidpdc );
    fidpdc = 0;
}

if ( fidpc != 0 )
{
    close( fidpc );
    fidpc = 0;
**** parse argument string */
invalid = 0; /* assume all arguments ok */
argnum = 0;
src = arg;
string = src;
dst = src;
while(copy(&src, &dst, '"', 100) > 1)
{
  if( (num = table(string, keynames)) < 0 )
    print("%s is not a valid argument\n", string);
  invalid = 1;
}
arglist[ argnum++ ] = num;
/* save start of string */
string = src;
if( invalid )
  return; /* try again */
arglist[ argnum ] = -1; /* terminate list of argument codes */
if( argnum == 0 )
{
  return;
}
type = arglist[0];
concat(PEKEY, locv(0, type), pckeyfile);
if( (fidpckey = open(pckeyfile, 0)) < 0 )
  perror("List can't open pckey file");
  return;
lo = 0;
hi = CHECKSIZ - 1;
do
{
lo = listcheck( fidpckey , fiopckey , checklist , lo , hi);  
hi = lo + CHECKSIZ - 1;

for( checkptr = checklist ; checkptr < &checklist[CHECKSIZ] ; checkptr++)
{
    if( checkptr -> keycount == 0 )
        continue;

    pcid = checkptr -> pnum;
    keynumber = checkptr -> keynum;
    occurs = checkptr -> keycount;
    if( ( fidpckey =getmax( pcid , pckey->pckkey ) ) < 0 )
        continue;

    putdata( pcid , fidpckey , pckey->pckkey->keylines , &pcdata);

    close( fidpckey );
    fidpckey = 0; /* mark it as being closed */
i = 0;

    for( markptr = pcdata , keyptr ; markptr++ )
    {
        if( markptr -> keytype == &type )
            if( i++ == keynumber )
                break;
    }

    printf( "%s \ %d \n", markptr -> keystore , occurs );

    for( argtype = arglist[ argnum ] ; argtype != -1 ; argnum++)
    {
        for( keynumber = ; keynumber < ; cdata . totkeys ; keynumber++ )
        {
            if( pcdata . keyptr[ keynumber ]; keytype == argtype )
            {
                key = pcdata . keyptr[ keynumber ]; keystring;
                printf( " %s: %s\n", keynames[ argtype ]; key );
                break;
            }
        }
    }

}  

while( lo <= 0 );

close( fidpckey ); /* mark it as being closed */
NAME: listcheck()

FUNCTION:
Fill in a checklist indicating the presence of keywords in a list of contacts.

CALLING SEQUENCE:
int fidpclist
int fidpckey
struct keycheck *checklist
int lu
int hi

PARAMETERS:

fidpclist file descriptor for the list of current contact accession numbers.

fidpckey File descriptor for the "pckey," file listing keyword id numbers of keywords occurring in contacts.

max Maximum number of keywords per contact from the given category. Effectively gives the length of the "rows" of the pckey file.

checklist The structs in this array give information about the occurrence of keywords in the current contact list:

  1. The number of permits in the current list which contain a given keyword.
  2. The accession number of one of the contacts that contains a given keyword.
  3. The number of keywords from the given category which precede the keyword in the contact name ordered by item 2 above.

flag If this flag is set, only want to get keywords whose high bits are set to denote non-compliance.

lo Defines the low end of the range covered by the checklist array.

hi Defines the high end of the range covered by the checklist array.
RE TUR4$S:  S
Rpturrs the smallest id number of the keywords occurring in the current
e-3 List of contacts, but not yet checked in the "checklist" array.
61 Presumably, this value will be used for the "lo" parameter of the
62 next call to this routine. If there is no such smallest id number,
63 the routine returns -1 to signify that all the keywords have been
64 covered.
65
ALGORITHM:
66 A pass is made through the "checklist" array to initialize it to
67 empty. Then, the input buffer for the current contact list
68 is filled to prepare for the main loop. In the main loop, each
69 iteration considers an accession number of a contact in
70 the current list of contacts. The routine moves through
71 the "pkey." file to locate the "row" corresponding to the given
72 contact number. The keys listed in this row are checked
73 in the "checklist" provided that they fall into the range
74 defined by "lo" and "hi", and they match the "flag" parameter.
75
CALLS:
76 seekL()
77 Also, Unix calls:
78 seek()
79 read()
80 perror()
81
CALLED BY:
82 list()
83
HISTORY:
84 written by Dan Putnam - spring 1979.
85
**********************************************************************
86
#include "structdefs.i" /* define keycheck struct */
87 #define PCLISTSIZ 25 /* buf size for current pc list */
88 #define PCKEYSIZ 108 /* buf size for pkey file */
89
100 listcheck (fiopcList, fidpckeys, checklist, lo, hi)
101 int fidpclist; /* descriptor for current list of pcs */
102 int fidpckeys; /* descriptor for pkey file */
103 struct keycheck checklist[]; /* used to keep track of found keys */
104 int lo; /* key id # of base entry in checklist */
/* key id # of last entry in checklist */
int hi;

/* smallest key id > hi */
int newlo;

/* buffer for current pc list */
int *pclist[PCLISTSIZ];

/* points thru pc list */
int *endpctist;

/* points off end of pc list */
int *pkeylist[PCKEYSIZ];

/* buffer for pc key file */
int *rckeyptr;

/* points thru pc key entries */
int *endpckey;

/* marks end of pc key buffer */
int pcids;

/* id number of pcs in pclist */
int keyids;

/* id of pc of pc key entries */
int keyval;

/* keyword value in pc key file */
int keynumber;

/* number of keyval in list */
int nbytes;

/* returned from reaos */
struct keycheck *checkptr;

/* run control: checklist */
int 1;

newlo = 00777777;

/* largest pos integer */
keypc = 0;

/* pc of first keys */

/***** init checklist to none found */
checkptr = checklist;
for( i = lo; i <= hi; i++)
{
    checkptr->pnum = -1;
    checkptr++->keycount = -1;
}

seek( fidpctist, 0, C);

/* be sure to start at the beginning */
seek( fidpckey, 0, W);

/* be sure to start at the beginning */

/***** fill up pc list buffer to get started */

if( (nbytes = read(fidpctist, pclist, PCLISTSIZ * 2)) < C )
{
    perror("list can't read current pc list");

    return;
}

pclistptr = pclist;

endpctist = pclist + (nbytes / 2);

/* pts off end of list */

/***** fill up pc key buffer to get started */
if ( nbytes = read(fidpckey, pckeylist, PCKEYSIZ * 2)) < C )
    {
      perror("list can't read pckey file");
      return;
    }
  pkeyptr = pckeylist;
  endpkey = pckeylist + (nbytes / 2);  /* pts off end of list */

  /******** run thru current pck list to get keys for each one */
  while( (pcid = *plistptr++) != -1 )  /* null terminate */
  {
    /******** first check if we have used up pck buffer */
    if( plistptr >= endplist )
    {
      /***** refill buffer and reset plistptr */
      if ( nbytes = read(fidpckey, pclist, PCLISTSIZ * 2)) < C )
        {
          perror("list can't read current pck list");
          return;
        }
      pclistptr = pclist;
      endplist = pclist + (nbytes / 2);  /* pts off end of list */
    }
  }

  /******** next, read up to proper section of pckey file */
  while( keypc < pcid )
  {
    if( *pckeyptr++ == -1 )
      keypc++;
    /* run thru unwanted key ids */
  }
  if( pckeyptr >= endpkey )
    {
      if ( nbytes = read(fidpckey, pckeylist, PCKEYSIZ * 2)) < C )
        {
          perror("list can't read pckey file");
          return;
        }
      pckeyptr = pckeylist;
      endpkey = pckeylist + (nbytes / 2);
    }

  /******** run thru keys for pcid and put in checklist */
for (keynumber = 0; (keyval = *pckeyptr++) != -1; keynumber++)
{
    if (pckeyptr >= endpckey)
    {
        if (nbytes = read(fd, pckeylist, PCKEYSIZE * 2)) < 0)
            perror("list can't read pckey file");
        return;
    }

    pckeyptr = pckeylist;
    endpckey = pckeylist + (nbytes / 2);
}

if (keyval < lo)
   continue;

if (keyval > hi)
{
    if (keyval < newlo)
        newlo = keyval;
    continue;
}

checkptr = checklist + (keyval - lo);

****** don't bump count if duplicate keyword in contact

if (checkptr->pcnum != pcid)
{
    checkptr->keycount++;
    checkptr->pcnum = pcid;
    checkptr->keynum = keynumber;
}

keypc = pcid + 1;  // above loop uses up keys for pcid

****** if newlo has its original value return -1 = all done

if (newlo == 077777)
    return(-1);
else
    return(newlo);
#include "structdefs.h"
#include "commandefs.h"
#include "params.h"

select( opcode, term)
int opcode; /* is number of command */
char term[]; /* null terminated string, argument of command */

{ extern int phase; /* for switching between scratch files */
  extern int fidscratch[2]; /* file descriptors for scratch files */
  int fidpckey; /* descriptor for pckey file */
  char pckeyfile[256]; /* used to build pckey file name */
  int old; /* file descriptor for scratch file */
  int new; /* file descriptor for scratch file */
  char buffer[256]; /* buffer for user response */
  char *src; /* utility pointer used with copy() */
  char *dst; /* utility pointer used with copy() */
  struct marker *termptr; /* points to struct describing term */
  int num; /* number of pcs from boot or keypcs */
  int filenum; /* category of keyword */
  int inumber; /* number of keyword in category */
  int onepc[2]; /* buffer for writing list of one pc */

  /******** if term is null use current and old lists */
  if( term[0] == 0 )
  {
    if( opcode != FIND )
      { /* note that we don't change phase on this one */
        new = fidscratch[ phase ]; /* new pc list will be in phase */
        old = fidscratch[ 1 - phase ]; /* out of phase */
        num = boot( old, new, new, opcode );
        printf("%d in current List\n", num);
      }
    return;
  }

  /******** look at term[0] to see if user wants just one pc */
  if( term[0] == *"#" )
  { /* maps 1 to 1 and maps 1 to */
    onepc[0] = atoi( term + 1 );
    onepc[1] = -1;
    phase = 1 - phase; /* maps 1 to 1 and maps 1 to */
    new = fidscratch[ phase ]; /* new pc list will be in phase */
  }

  /******** look at term[0] to see if user wants just one pc */
  if( term[0] == '#')
  { /* maps 1 to 1 and maps 1 to */
    onepc[0] = atoi( term + 1 );
    onepc[1] = -1;
    phase = 1 - phase; /* maps 1 to 1 and maps 1 to */
    new = fidscratch[ phase ]; /* new pc list will be in phase */
  }

  }
old = filescratch[1 - phase];  /* old will be out of phase */
58
59 seek(new, C, 0);
60 if(write(new, onepc, 4)  < 4 )
61 {
62    perror("select can't write to pc list");
63    exit();
64 }
65
66 printf("pc %d selected\n", onepc[0]);
67 if(opcode != FIND )
68 {
69    num = bool(old, new, new, opcode);
70    printf("%d in current list\n", num);
71 }
72 return;
73
74 /****** special case: user wants list of all pc's */
75 if(compar(term, ALL) == 0 )
76 {
77    phase = 1 - phase;  /* maps 0 to 1 and maps 1 to 0 */
78    new = filescratch[ phase ];  /* new pc list will be in phase */
79    old = filescratch[ 1 - phase ];  /* old will be out of phase */
80    num = initlist(new);
81 printf("%d in current list\n", num);
82 if(opcode != FIND )
83 {
84    num = bool(old, new, new, opcode);
85    printf("%d in current list\n", num);
86 }
87 return;
88
89 /****** here is where we handle ordinary keywords */
90 if( (term = keyword(term))  != 0 )
91 {
92    filenum = term -> file;
93    idnum = term -> idnum;
94    if( filenum >= MESSAGES )
95    {
96        help(0, term);  /* print cut message for user */
97 54
return;
}

/****** reset phase to switch new and old files ******/

phase = 1 - phases;          /* maps C to 1 and maps 1 to 0 */
new = fiidscratch[phase];    /* new pc list will be in phase */
old = fiidscratch[1 - phase]; /* old will be out of phase */

concat( PCKEY, lccv(0, filenum), pckeyfile);
if( (fidpckey = open(pckeyfile, C)) < 0 )
{
    perror( "select, pckey open" );
    return;
}

/****** use keyst to get list of pcs for non-event keyword ******/

num = keypcs(idnumber, fidpckey, new);
printf("%d found\n", num);
close( fidpckey );

if( opcode != FIND )
{
    num = bool(old,new,opcode);
    printf("%d in current list\n", num);
}
return;
#include "structdefs.h"
#include "params.h"

#define LSTSIZ 256

int show(fid)

struct getbuf bufin; /* buffer for get() routine */
char line[256];  /* gets lines from get() */
int nbytes;  /* char count from get() */
char *key;  /* points to keyword string */
int *index to categories */
int printflag; /* flags printing first key of a type */
struct keymarker *markptr; /* points thru keyptr array of pcdata */
extern char *keynames[]; /* names of keyword categories */
int keynumber; /* counts filled in keyptr entries */
struct keydata pcdata; /* picks up keyword lines for pcs */
char *src; /* utility pointer used with copy() */
char *dst; /* utility pointer used with copy() */
int pclist[LSTSIZ]; /* buffer for input and output lists */
register int j; /* fast loop counter */
register int pcid; /* no number of source permit in lists */
int *pclistptr; /* points thru list buffer */
int *endlist; /* marks end of pclist array */
struct ndx pcindex; /* offsets of data in pc file */
long *offset; /* temp copy of pcindex offsets */
static int fidpc; /* file descriptor for pc data file */

/***** make sure fidpc is closed before using again *****/
if( fidpc != 0 )
{
    close( fidpc );
    fidpc = C;  /* mark it as being closed */
}

seek(fid, 0);  /* make sure we get whole file */
while( (j = read(fid, pclist, LSTSIZ * sizeof(*pclist))) > C)
{
    j = 2; /* j = number of ints read */
    pclistptr = pclist;
    while((j--) && (pcid = *pclistptr++) != -1)
if ( fidpc = getndx( pcid, &pindex) ) < C )
    continue;

printf("pc %lu\n", pcid);

putdata( pcid, fidpc, pindex, keylines, %cdata);

for( type = C; type < NUMTYPES; type++ )
{
    printfflag = 0;
    markptr = pcdata + keyptr;
    for( keynum = C; keynum < %cdata, totkeys; keynum++ )
    {
        if ( markptr -> keytype == type )
            if ( printfflag++ == 0 )
                printf(" %s:\r", keynames[ type ]);
        printf(" %s\n", markptr -> keystring);
        printfflag++;
    }
    /****** now print text data */

printf("\n\n");

offset = pindex, datalines;

if( offset < C || seek( fidpc, offset) < C )
    printf("can't seek to data\n");
    continue;

bufin.fildes = fidpc;
bufin.nleft = 0;

while( ( nbytes = getl( line, &bufin) ) > 0 )
{
    line[ nbytes ] = C;
    if ( line[0] == 'S' )
        break;
    printf("%s", line);
}

close( fidpc ); /* all done with this one */
fidpc = 0; /* mark it as being closed */
```c
#define ****

if(pcid == -1)
    break;
```
#

/* ******************************************

NAME:
lookup()

FUNCTION:
Look in the hash table file for the word marker structs corresponding to a given string.

CALLING SEQUENCE:
char *word
int fileid
int checkflag
struct marker *findptr
int max

PARAMETERS:

word Points to the string to be looked up in the hashtable.
fileid The number of the keyword file in which the word should be located. If this flag is -1, then all the keyword files are searched.
checkflag If this flag is G, then a struct whose hash values match those of the given word is assumed to match the word. If this flag is non-zero, then the keyword corresponding to such a struct is read from its keyword file, and compared to the given word.
findptr Points to an array of structs which is filled in by "lookup()" with the structs which match the given word.
max Gives the size of the above array so that "lookup()" can avoid overwriting that array.

RETURNIS:
The number of struct elements matching the given word.

If an error condition is encountered on an "open()", "seek()" or a "read()", then "exit()" is called to terminate the program.

ALGORITHM:
On the first call, the hashtable file is opened and the file descriptor is saved in a static variable to save time on subsequent calls. At this time, "fstat()" is called to determine the length of the hashtable file measured in marker structs. The COUNTERFLOW parameter is subtracted from this length to determine the "modulus". Obviously, this parameter must agree with its counterpart in the
To look up the given word in the hash table, the "hashfn()" routine is called to compute the 3 hash values of the word. The index into the hash table is computed from the hash values and the "moculus". A calculation is performed to determine the number of structs that can be read beginning with the index, that will not cross a 512 byte boundary in the file. This makes the initial read from the hash table about twice as fast as if it crossed the boundary, and the first read almost always encompasses the collision list.

The structs in the collision list are scrutinized to see if they match the input word and those that do are copied into the array of structs indicated by "findptr". If this array runs out of room, the structs are no longer copied, but the count of matching structs still continues.

**CALLS:**

- **hashfn()** To compute hash values of the input word.
- **seekto()** To perform seeks at long offsets.
- **concat()** To concatenate strings. (borrowed from CELOS)
- **getl()** To read keywords from the keyword files.
- **copy()** To extract the keywords from the line on which they are declared.
- **compar()** To compare strings. (borrowed from CELOS)

Also, the following Unix calls.

- **open()**
- **fstat()** To get the size of the hash table file.
- **read()**
- **exit()**
- **signal()**
- **printf()**

**CALLED BY:**

Various programs that need to look up keywords.

**HISTORY:**

written by Dan Putnam - fall 1978

This routine is essentially identical to the "lookup()" used in the PAMS system. The "include" files are the only major difference, and this change was needed only to redefine the "KEYNAME" parameter. Adaptations were made, spring 1978, by Dan Putnam.
#include "structdefs.h"
#include "params.h"

#define HASHBUFSIZE 32 /* fits in one block */
#define OVERFLOW 100

lookup(word, fileid, checkflag, findptr, max)
char *word; /* points to word we are looking for */
int fileid; /* if -1, any files; if >= 0, specific */
int checkflag; /* if 1, then check characters */
struct marker *findptr; /* for markers of found words */
int max; /* length of findptr array */

{
  struct filestruct /* used for getting length of hashtable */
  {
    char jnk[93]; /* don't need this stuff */
    char size0; /* high byte of file size */
    int size1; /* low word of file size */
    char jnk2[24]; /* don't need this stuff either */
  } filedata;

  struct /* used to load size0 and size1 into a long */
  {
    char hi_byte; /* corresponds to size0 of filestruct */
    char highest; /* high order byte of a long */
    int low_word; /* corresponds to size1 of filestruct */
  };

  struct /* used to access high and low words of a long */
  {
    int hibits;
    int lobits;
  };

  int num; /* for returning number of finds */

  register struct marker *srcptr; /* points thru hashtable */
  register struct marker *astptr; /* for moving found markers */

  struct getbuf buf; /* struct used by getl() routine */
  int nbyte; /* char count returned from getl */
  char filename[256]; /* for building keyword file name */
  char keyline[256]; /* for reading line from keyword file */
  char keystring[256]; /* gets keyword string out of keyline */
  char *src; /* used with copy routine */
  char *dst; /* used with copy routine */
  long index; /* index into hashtable */
  long boundary; /* 312 byte boundary after index */
  register int readbytes; /* bytes in markers up to boundary */
  int hashval[3]; /* hash values */

  struct marker hashbuf[HASHBUFSIZE]; /* buffer for hashtable */
struct marker *endbuf;  /* end of markers in hashbuf */
int evenword;  /* number of bytes in hashbuf */

static int fidhash;  /* descriptor of hashtable */
static long modulus;  /* modulus for hash algorithm */

****** first call initialization ******
if( fidhash == 0 )
{
  if( (fidhash = open(table, O_RDONLY)) < 0 )
    perror("lookup: can't open hashtable");
  exit();
}

****** get size of hashtable to compute modulus ******
fstat( fidhash, &filedata);
modulus = 0;
modulus hi byte = filedata. size64; modulus low word = filedata. size1;  /* size of hashtable */
modulus = filedata. size of ( *hashbuf );  /* number of keyword markers */
modulus -= OVERFLOW;

****** compute hash values of word and look into hashtable ******
hashfn(word, hashval);
index. lobits = hashval[0];
index. hibits = hashval[1] & 00000000;
index = index % modulus;

****** compute number of bytes from index to 512 byte boundary ******
readbytes = 512 - ( index. low word & 077777);
readbytes = (readbytes / sizeof( *hashbuf ) ) * sizeof( *hashbuf );
if( readbytes > sizeof( hashbuf ) || readbytes == 0 )
  readbytes = sizeof( hashbuf );
if (seekl (fidhash, index) < 0) {
    printf("failed on seek into hashtable\n");
    exit();
}

// look at hashtable entries until an empty slot is found
num = 0; /* none found so far */
dstptr = finoptr; /* copy to register pointer for extra speed */
while ((nbytes = read (fidhash, hashbuf, readbytes)) > 0) {
    readbytes = sizeof (hashbuf); /* next time fill buffer */
    endbuf = hashbuf + (nbytes / sizeof (*hashbuf));
    for (srcptr = hashbuf; srcptr < endbuf; srcptr++) {
        if (srcptr -> file == -1) {
            return (num); /* thats all folks */
        }
        if (srcptr -> hashv[0] != hashval[0]) continue; /* not found */
        if (srcptr -> hashv[1] != hashval[1]) continue; /* not found */
        if (srcptr -> hashv[2] != hashval[2]) continue; /* not found */
        if (fileid >= 0 && srcptr -> file != fileid) continue; /* not in the right file */
    }
    if (checkflag) {
        /* check strings to be absolutely sure */
        concat(KEYNAME, locv(0, srcptr -> file), filenane);
        if (bufr.filles = open (filenane, 0)) < 0 ) {
            perror("lookup: can't open keyword file");
            exit();
        }
        bufr.nleft = 0;
    }
}
if ( seekl ( bufr.fildes, srcptr->beginbyte ) < 0 )
{
    perror( "Lookup can't seek to key~dord" );
    exit( );
}

if ( nbytes = getl ( keyline, &bufr ) < 0 )
{
    printf ( "Lookup can't read keyword file\n" );
    exit ( );
}

clsoe ( bufr.fildes );
keyline [ nbytes - 1 ] = 0;
src = keyline + 1;
start = keystring;
copy ( 8src, &dst, ':', 256 );

if ( comparl ( keystring, word ) != 0 )
    continue; /* no match */

if ( num++ < max )
{
    dstptr->hashv[0] = srcptr->hashv[0];
    dstptr->hashv[1] = srcptr->hashv[1];
    dstptr->file = srcptr->file;
    dstptr->linum = srcptr->linum;
    dstptr->beginbyte = srcptr->beginbyte;
    dstptr++;
}

}
NAME:
  help()

FUNCTION:
  Print any lines following the line which declares a keyword in
  a keyword file.

CALLING SEQUENCE:
  int help(int fid, struct marker *termpt)

PARAMETERS:
  fid  File descriptor for output messages. Set to 1
       for output to the user's terminal.
  termpt Points to a word marker struct identifying a given
           keyword.

RETURNS:
  nothing.

ALGORITHM:
  The routine examines "termpt" and returns immediately if it is
  a null pointer. Otherwise, the category number is appended to
  the keyword file prefix and the keyword file is opened. The offset
  stored in the marker is used to seek into the keyword file.
  Note that 1 is added to the offset to skip over the "#" character
  which marks the keyword. This line is not printed, but subsequent
  lines are printed until a line beginning with '#' is found or
  until the end of file.

CALLS:
  concat()
  s-ekl()
  g-tll()

Also, Unix calls:
  open()
  perror()
  write()
  close()

CALLED BY:
  ficep()
HISTORY:
written by Dan Putnam - fall 1978 - for PAMS system.
Adapted for use by the IICEP system - spring 1979 - by changing the
"include" files to define the "KEYNAME" parameter differently.

*****************************************************************************

#include "structdefs.i"
#include "params.i"

help(fid, temptr)
int fid;
struct marker *temptr; /* descriptor of output file */
struct dvfs *struct marker; /* describes keyword */
char filerame[300];
char line[80]; /* input line from file */
struct getbuf buffer; /* used by getl routine */
int nchars; /* number of chars in line */
int linecount; /* number of lines printed */

if ( temptr == 0 )
   {
      /* nothing to print */
      return;
   }
concat(KERNAME, locv(0, temptr=file), filename);
if ( buffer.fildes = open(filenamg, O) ) < 0)
   {
      perror("help can't open");
      return;
   }

buffer.nleft = 0;
if ( seekl(buffer.fildes, temptr->beginbyte + 1) ) < 0 )
   {
      perror("help can't seek to keyword");
      close(buffer.fildes);
      return;
   }

linecount = 0;
while ( (nchar = getl (line, &buffer)) > 0 && line[0] != 'n')
   {
      if ( linecount++ == 0 )

continue;       /* skip over first line */
line[nchars] = 0;  /* insert null after the end-of-line */
write(fid, line, nchars);
}
close(buffer.fildes);
}
NAME:

iam()

FUNCTION:

Create a unique name which can be concatenated with scratch file
names to prevent multiple instances of a program from overwriting
each other's scratch files.

CALLING SEQUENCE:

char *me
int iam()

PARAMETERS:

me - Points to a character buffer of at least 3 characters.
This buffer receives the unique name, which consists of a lower
case letter, followed by a "#" character and a null character.

RETURNS:

positive integer if name creation was successful.
negative integer otherwise.

ALGORITHM:

The routine uses the "creat" system call to attempt to create
a file named with the string "me". The "creat" fails if a file
already exists with this name and does not have write access.
If this occurs, then the name is altered and the process continues
until a unique name is found or else the lower case prefixes have
been exhausted. In the latter case, -1 is returned to signify
failure in creating the unique name. When the procedure succeeds
in creating a unique name, the file opened by iam() is closed
before returning. It is not expected that this file will be
used for anything except to mark its name as already being in use.

CALLS:

creat() Unix system call to create files.

CALLED BY:

usually a main program.

HISTORY:

written by Dan Putnam - fall 1978
char *me;
{
    int i;
    int fid;
    me[0] = 'a';
    me[1] = 'b';
    me[2] = 0;
    for (i = 0; i < 26; i++)
    {
        if (fid = creat(me + i)) < 0)
            me[i]++;
        else
        {
            close(fid);
            return fid;
        }
    }
    return fid;
}
```c
/*
 * CONCAT (Note: Borrowed from CELDS, Thans!)
 * Concatenate two strings into one string. Concat returns a
 * pointer to the end of the resultant string so that successive calls
 * to concat may be made easily.
 * 
 * Arguments: first  pointer to first string
 *            second pointer to second string
 *            result pointer to end of resulting string
 * 
 * Returns: pointer to end of result string
 * 
 * Calls: none
 */

char *concat (first, second, result) char *first, *second, *result;
{
    while (*result++ = *first++); /* Copy first string to result */
    --result; /* Back up over null */
    while (*result++ = *second++); /* Copy second string to result */
    --result; /* Back up over null */
    return (result);
}
```
#
/*****************************/

NAME:
copyfile()

FUNCTION:
Copy the contents of one open file to another.

CALLING SEQUENCE:
int fidin
int fidout

PARAMETERS:

fidin  file descriptor of source file opened for reading.

fidout  file descriptor of destination file opened for writing.

RETURNS:
nothing.

ALGORITHM:
The routine first seeks to the start of both files in case other procedures have used the file descriptors. Then the routine reads from the source file in a loop and writes the same number of bytes to the destination as it reads.

CALLS:

Unix calls:
seek()
read()
write()
errort()

CALLED BY:
pam (main program )
restore()
save()

HISTORY:
written by Dan Putnam - fall 1978

******************************************************************************
57 copyfile( fidin, fidout)
58 int fidin;  /* descriptor of source file */
59 int fidout;  /* descriptor of destination file */
60 {
61   int nbytes;
62   char buffer[512];
63   seek( fidin, 0, 0);  /* from beginning */
64   seek( fidout, 0, 0);  /* from beginning */
65   while( (nbytes = read( fidin, buffer, 512)) > 0 )
66     {
67       if( write( fidout, buffer, nbytes) < nbytes )
68         {
69           perror("write write error");
70           close( fidout );
71           return;
72       }
73     }
74   }
75
76   if( nbytes < 0 )
77     {
78     perror("copyfile, read error");
79   }
NAME: resp()

FUNCTION:
Get a line of user response from the terminal.

CALLING SEQUENCE:
char *bfr
int resp()

PARAMETERS:
bfr Points to buffer for user response.

RETURNS:
Returns the number of characters in the response exclusive of '\r',
or returns -1 on end-of-file.

ALGORITHM:
The routine works with a limit of 80 characters per response.
Characters are read from the terminal until either 8C are read or
an end-of-line or end-of-file is encountered. If the last character
is an end-of-line, then it is overwritten with a null.

CALLS:
nothing.

CALLED BY:

HISTORY:
written by van Putnam - fall 1978

#define MAXCHR 80 /* maximum response length*/

resp(bfr)
char *bfr; /* character buffer for user response */

register int countdown;
register int chr;
register char *ptr;

countdown = MAXCHR;
ptr = bfr;
do {
    *ptr++ = chr = getchar();
    if (chr == '0')
        return (-1);
} while (--countdown && chr != 'n');

if (chr != 'n')
    while (getchar() != 'n'); /* flush input */
else
    (*(--ptr) = 'c; /* replace CR by null */

return (ptr - bfr);
NAME: copy()

FUNCTION: Move characters from one string to another and update pointers to source and destination for subsequent calls.

CALLING SEQUENCE:

```c
char **source
char **dest
char delimiter
int maxchars
```

PARAMETERS:

- source is the address of a pointer to the source character string. This pointer is updated to point past the last character moved.
- dest is the address of a pointer to the destination string. This pointer is left pointing past the null character terminating the string that was moved.
- delimiter is the character signalling the end of the source string. If this character is not encountered, a null character will halt the transfer of characters.
- maxchars is the size of the destination string. If there are more characters to be moved than maxchars, -1 is returned and copy does not overwrite the end of the buffer.

RETURNS:

-1 if the size limitation given by maxchars can not be met.
otherwise copy returns the number of characters moved including the null character terminating the destination string.

ALGORITHM:

The copy routine skips over leading blank or tab characters. Embedded substrings of blanks or tabs in the source string are condensed to one blank. The transfer of characters stops when the delimiter character or a null character is encountered or when the size limitation given by maxchars is met. The source pointer is never moved past a null character. In this case, subsequent calls to copy move an empty string. If the delimiter is not null and it is encountered before a null, then the source pointer is moved past the delimiter. Thus, successive calls can move substrings separated by the delimiter.

The destination string is null terminated and the destination
pointer is left pointing past the null. Thus, repeated calls
to copy can move strings into a shared buffer.

CALLS:
nothing

CALLED BY:
all kinds of procedures that move strings around.

COMMENTS:
copy() can be used for several different purposes:
1. cleaning a string to eliminate extra blanks or tabs.
2. parsing a line into fields.
3. counting the number of fields on a line.

HISTORY:
written by Dan Putnam - fall 1978

******************************************************************************
copy(source, dest, delimiter, maxchars)
char **source; /* points to a pointer to source string */
char **dest; /* points to pointer to destination */
char delimiter; /* stop copying when this char is found */
int maxchars; /* size of destination */

{  register char *src; /* copy of source for speed, esthetics */
   register char chr; /* temp for *src to save indirection */
   register int slack; /* room left in destination */
   char *dst; /* points to destination */
   int ret; /* return value */

   src = *source;
   dst = *dest;
   slack = maxchars; /* available room */

   if(slack <= 0)
      {  return(-1);
      }

   while(*src == ' ' || *src == '\t')
      src++;
/***** now run through the rest of the string ******/

do
{
    if (chr = *src) == 0)
        break;
    src++;  /* not null, so move on */
    if (chr == delimiter)
        break;

    if(chr == "" || chr == \t")  /* if blank or tab */
        while ( (chr = *src) == "" || chr == \t")
            src++;

    if (chr == 0)
        break;

    if (chr == delimiter)
        {  
            src++;  /* move past delimiter */  
            break;
        }

    chr = ";
    }  /*st++ = chr;  
    while (--slash );  */

    if (slack > 0)
    {  
        ret = maxchars - slack + 1;
    }
    else
    {  
        /* looks like we didn't find the end but ran out of room */
        --dst;
        ret = -1;
        /* move src past delimiter or up to null byte */
        while ( (chr = *src) != delimiter && chr != 0 )
            src++;
169       if(chr != 0)
170           src++;
171
172       }
173
174       *dst++ = C2;  /* leave dst pointing past null byte */
175
176       *source = src;
177       *oest = dst;
178
179       return( ret );
180     )
execute - send a string to sh to be executed
execute (command) char *command;

if ((child = fork()) < 0) /* Set up the fork */
    return(-1);

/* The child does the exec using the argument string */
if (child == 0) {
    signal(2, 0);
    exec( "/bin/sh", "sh", "-c", command, 0);
}

signalstatus = signal(2, 1);
while (wait(&waitstatus) != child); /* wait for child */
signal(2, signalstatus);
return(0);
NAME:

`table()`

FUNCTION:
To look up a character string in an array of string pointers.

CALLING SEQUENCE:

```c
char *string
char **ptrarray
int table()
```

PARAMETERS:

- `string` - points to a null terminated string of characters.
- `ptrarray` - points to a null terminated array of character pointers.

RETURNS:
- `-1` if the string is not found in the array of pointers.
- otherwise, `table()` returns the index of the first pointer in the array pointing to an identical string.

EXAMPLE:

Define "name" and "nametable" as follows:

```c
char *name = "jody";
char *nametable[] = { "fred", "jody", "pat" };
```

Then the call `table(name, nametable)` returns 1 to indicate that `nametable[1]` points to the same string as "name".

However, `table("joe", nametable)` returns `-1`, since "joe" is not listed in "nametable".

ALGORITHM:

The "ptrarray" is searched sequentially, and if a pointer in the array points to a string agreeing with that indicated by the "string" argument, then "table" returns the index of that element in the array. If a null pointer is found in the array, then `-1` is returned.
CALLS:
compar() A routine borrowed from CELDS to test whether string pointers point to identical strings.

CALLED BY:
usually routines that need to parse command strings or check for "legal" values of string variables from among those in a small, pre-defined list.

HISTORY:
written by Dan Putnam - fall 1978

*******************************************************************************

table(string,**ptrarray)
char *string; /* pts at null terminated string */
char **ptrarray; /* pts at null term array of char ptrs */
{
    register char **ptrptr; /* copy of ptrarray for speed */
    register char *ptr; /* copy of *ptrptr for speed */
    register int i; /* fast loop counter */
    ptrptr = ptrarray;
    for(i = 0; (*ptr = *ptrptr++); ++i) {
        if(compar(string,ptr) == 0)
            return(i);
    }
    return(-1);
}
NAME: save()

FUNCTION: Save the contents of the current scratch file of id numbers in
a file named by the input character string.

CALLING SEQUENCE:
char *filename

PARAMETERS:
filename Points to the string naming the output file.

RETURNS:
nothing.

ALGORITHM:
The routine attempts to create a file named by the "filename"
argument. If this attempt fails, the routine prints a message
to that effect and returns. If it succeeds, then the "copyfile()"
routine is used to copy the contents of the current scratch file
to the file which has been created.

CALLS:
copyfile()

Also, Unix calls:
creat()
seek()
perror()

CALLED BY:
pams (main program)

HISTORY:
written by Dan Putnam - fall 1978

save( filename )
char filename[]; /* string naming output file */
extern int phase;
extern int fidscratch[23];
int fidin;
int fidout;
int fidlin;
int fidout;

fidin = fidscratch[phase];
seek( fidin, 0, 0);

if( (fidout = creat( filename, 0666)) < 0 )
{
    perror("save can't create file");
    return;
}

copyfile( fidin, fidout );
}
NAME:

restore()

FUNCTION:

Restore a list of data accession numbers to current status.

CALLING SEQUENCE:

char *filename

PARAMETERS:

filename Points to string naming the file of accession numbers. If filename points to a null string, then the previous list is restored to current status.

RETURNS:

nothing.

ALGORITHM:

The "filename" parameter is checked to see if it points at a null string. If so, then the global "phase" variable is reset to switch the scratch files. If the "filename" parameter points at a non-null string, then the routine attempts to open the file. If the open is successful, then "phase" is reset and "copyfile()" is called to copy the contents of the input file into the current file.

CALLS:

copyfile()

Also, Unix calls:

open()
perror()

CALLED BY:

pams (main program)

HISTORY:

written by Dan Putnam - fall 1978

restore( filename )
char *filename;

/* names file to be read in. */

extern int phase; /* for switching scratch files */
extern int fidscratch[2]; /* scratch file descriptors */
int fidin; /* descriptor for restored file */
int fidout; /* copy of scratch descriptor */

if (*filename == 0)
{
    /* phase switch effectively restores old list */
    phase = 1 - phase;
    return; /* that's all there is to it */
}

if (fidin = open(filename, C)) < 0)
{
    perror("restore can't open file");
    return;
}

phase = 1 - phase; /* switch scratch files */

fidout = fidscratch[phase]; /* write to in-phase file */
copyfile(fidin, fidout);
NAME: peek()

FUNCTION: Invoke the editor on the keyword file corresponding to the category name given by the input argument string.

CALLING SEQUENCE:
char *category

PARAMETERS:
category String naming the category that the user wants to inspect.

RETURNS:
nothing.

ALGORITHM:
The routine begins by checking to see that "category" matches an entry in the "keynames" array. The index of a matching string in that array is the keyword file number of the corresponding file of keywords. This number is appended to the keyword file prefix and the editor is invoked on this file using "execute()".

CALLS: table() concat() execute() Also, Unix routines:
locv()

CALLED BY:
pams ( main program )

HISTORY:
written by Dan Putnam - fall 1978

peek( category )
char *category; /* name of a keyword category */
char command[NH]; /* argument string for execute routine */
extern char *keyprefix; /* keyword file name prefix */
extern char *keynames[]; /* names of categories */
int num; /* category number */

if (num = table(category, keynames)) < 0)
{
    printf(""s is not a keyword category
", category);
    return;
}
concat("ed ", keyprefix, command);
concat(command, locv(0, num), command);
execute(command);
NAME:

keyword()

FUNCTION:

Look up a string and prompt for correct category in case of duplicates.

CALLING SEQUENCE:

char *term
struct marker *keyword().

PARAMETERS:

term Points to the string to be looked up.

RETURNS:

keyword() Points to a marker struct which contains data on the string which has been looked up.

ALGORITHM:

Most of the work is done by the "lookup()" routine; this routine is primarily just a user interface to "lookup()". A call to "lookup()" is performed with the parameters set to find all occurrences of the string in the database and check the spelling character for character. If no instances are found, then a message to that effect is printed and a zero pointer is returned. If more than one instance is found, the user is prompted to name which category he wants. A pointer to the appropriate marker struct is returned.

CALLS:

lookup()
resp()

CALLED BY:

letter()

PMS (main program)

select()

HISTORY:

written by Dan Putnam - fall 1978
#include "structdefs.i" /* marker decl */
#define MAXFIND 50 /* found array size */

keyword(term)
char *term;
{
    extern char *keynames[]; /* names of keyword categories */
    char buffer[256]; /* for getting user response */
    int index; /* loop control: found words */
    int filenum; /* file number of a found word */
    static struct marker copyList[MAXFIND]; /* identical copies */
    int copies; /* counts number of copies */

    /****** Lookup: any category, check strings */
    copies = Lookup(term, -1, 1, copylist, MAXFIND);

    /****** if copies > 1, prompt for correct category */
    if( copies <= 0 )
    {
        printf("can't find '\%s'\n", term);
        return(0);
    }
    else
    {
        /****** prompt for the correct category */
        printf("Which category?\n");
        for(;;)
        {
            for(index = 0; index < copies; index++)
            {
                filenum = copyList[index].file;
                printf("%s: %s\n", index + 1, keynames[filenum]);
            }
            resp(buffer);
            index = atoi(buffer) - 1;
            if( index >= 0 ) && index < copies )
                break; /* a valid response */
        }
    }
}
return &copyylist(index);
NAME:

hashfn()

FUNCTION:

Convert a null-terminated character string to a 3-word array of integer hash values.

CALLING SEQUENCE:

char *string
int *hashout

PARAMETERS:

string - points to null-terminated string to be hashec.
hashout - points to output array of 3 hash values.

RETURNS:

nothing.

ALGORITHM:

One pass is made through the string for each of the 3 output hash values. On a given pass, hashfn() treats the characters in the input string as 5, 6, or 7-bit strings, respectively.

This is accomplished by masking off the appropriate number of high order bits in each character (i.e., 3, 2, or 1). The algorithm effectively treats the input string as a bit string, which is "wraps around" the output hash value integer.

The routine initializes the hash values to zero and processes the input characters until the null terminator is encountered. As each input character is considered, the low order bits are exclusive-ored into the hash value after being shifted past the bits from the previous character. If this results in losing bits off the end of the integer, the lost bits are exclusive-ored onto the low order bits.

For example, when the first hash value is computed, the first three characters of a string contribute their low order 5 bits to give the low order 15 bits of the integer hash value. The low order 5 bits of the next character are exclusive-ored into the output integer as follows. The low order bit is exclusive-ored onto the remaining high bit of the integer, and the next 4 bits are exclusive-ored onto the first 4 bits of the integer. The fifth character is shifted into place beginning with the fifth bit of the output integer.

CALLS:
nothing.

CALLED BY:

hash The program which creates the hashtable file.

lookup() The routine used to look up terms in the keyword files.

HISTORY:

written by Dan Putnam - fall 1978

COMMENTS:

The three hash values generated by this routine virtually identify strings uniquely. The three hash values are essentially orthogonal in the sense that if two terms collide under one of the hash functions, there is no increased likelihood that they will collide under either of the other two.

In a file of about 13,000 English words, no two words were found that collided under both of the first two hash functions. When the third function is also considered, it seems virtually assured that if two terms agree in all three hash values, then the two words are identical. If it is assumed that the bit patterns of the hash values are random, it may be computed that the chances of finding a collision in a collection of 10,000 words is less than one in a million.

**************************************************************************

int maskarray[]

| 357 |
| 077 |
| 0177 |

int nbitarray[]

| 5 |
| 6 |
| 7 |

hashfn(string, hashout)

/* string to be hashed */

int *hashout; /* 3 word output array */

register int numbites; /* number of bits used in hash */
int mask; /* masks low numbits */
char *cpt; /* points thru string */
register int chr; /* temp copy of *cpt */
register int shift; /* shift chr by this many bits */
int hashv; /* gets hash value */
int i; /* loop control: 3 hash values */

for(i = C; i < 3; i++)
{
  mask = maskarray[i];
  numbits = nbitarray[i];
  hashv = C;
  shift = C;
  cpt = string;
  while(chr = *cpt++)
  {
    chr &= mask; /* remove unwanted bits */
    hashv = " (chr << shift); /* shift into place */
    shift = (shift + numbits) & G17; /* == numbits mod 16 */
    if(numbits > shift) /* if we wrap around word */
    {
      hashv = " (chr >> (numbits - shift));
    }
    hashout[i] = hashv;
  }
}
NAME:
seekl()

FUNCTION:
Perform seeks into files with long offsets.

CALLING SEQUENCE:
int fid
long offset

PARAMETERS:
fid is the file descriptor of an open file.
offset is the offset from the beginning of the file to which seekl() will seek.

RETURNS:
returns the same value as the seek() system call returns to seekl().
-1 signals an error condition.

ALGORITHM:
seekl() tests offset to see if the seek can be performed as an ordinary short integer seek. If not, then seekl() first seeks by blocks (512 bytes) and then seeks the rest of the way by bytes.

CALLS:
seek() - Unix system call.

CALLED BY:
all kinds of routines that read from random locations in large files.

HISTORY:
written by Dan Putnam - fall 1978

seekl(fid, offset)
int fid;
long offset;
{
  struct /* for accessing hi and lo words of offset */
register int code;  // return code from seek
register int i;

if(offset.hi != 0)
{
    if (code = seek(fid, (i = offset, / 512), 3)) < 0)
    {
        return(code);
    }
    else
        return(seek(fid, (i = offset % 512), 1));
}
else
    return(seek(fid, offset.lo, 0));
/**
 * compar two null-terminated strings
 * The characters at "s1" and "s2" are compared until one terminates.
 * If the last characters compared are equal, zero is returned;
 * if the char from "s1" is > "s2", a positive value is returned;
 * otherwise a negative value.
 *
 * Calls: none
 * Globals: none
 * Last modification: 31 mar 77
 *
 * int compar (s1, s2) char *s1, *s2;
 {
     register char *p;
     register int greater;
     p = s1;
     q = s2;
     while ((greater = *p - *q++) == 0 && *p++ != 0);
     return (greater);
 }
NAME: bool()

FUNCTION: Perform boolean operations on files.

CALLING SEQUENCE:
int fida
int fidb
int fidc
int opcode

PARAMETERS:

  fida  File descriptor of the first argument file.
  fidb  File descriptor of the second argument file.
  fidc  File descriptor of the output file.
  opcode  Specifies the operation to be performed:
          1 - file(a) AND file(b)
          2 - file(a) OR file(b)
          3 - file(a) EXCEPT file(b)

RETURNS: Returns the number of items listed in the output file.

ALGORITHM:
The input files are read and their contents are used as indices into the "check" array. Bits are set in the "check" array elements to indicate whether a given entry is present in either or both of the input files.

When the above step is completed, a pass is made through the check array. The index of a "check" array element is written to the output buffer depending on its membership in the input files and the value of the "opcode".

  AND           belongs to file(a) and to file(b).
  OR            belongs to file(a) or to file(b) or both.
  EXCEPT        belongs to file(a) but not to file(b).

Note: The "check" array is an array of SPMAX characters, where SPMAX is currently defined at 5120. This parameter should
be large enough for some time to come, and could be set still
higher without exceeding core limitations. However, somebody
probably ought to rewrite this routine so that it loops to
write the output file in segments. That is, the routine would
make a complete pass through both input files in each iteration.
Only those values in the current segment range would be marked
in the "check" array.

CALLS:
Unix calls:
seek()
read()
write()

CALLED BY:
select()

HISTORY:
written by Dan Putnam - fall 1978

********************

#define AND 1
#define OR 2
#define EXCEPT 3
#define MASKA 01
#define MASKB 010
#define SMPAX 5000
#define LSTSIZ 256

bool(fida, fidb, fidc, opcode)
int fida;
int fidb;
int fidc;
int opcode;
int j;
register char *checkptr;
register int spid;
int *listptr;
int *endlist;
int maxa;
int maxb;
int maxc;
int count;

(/******** first, clear check array */

/***********

#define ALL 1
#define OR 2
#define EXCEPT 3
#define MASKA 01
#define MASKB 010
#define SMPAX 5000
#define LSTSIZ 256

bool(fida, fidb, fidc, opcode)
int fida;
int fidb;
int fidc;
int opcode;
int j;
register char *checkptr;
register int spid;
int *listptr;
int *endlist;
int maxa;
int maxb;
int maxc;
int count;

(/******** first, clear check array */

/***********
checkptr = check;
j = SMAX;
do
  checkptr++ = 0;
  while (--j);

/***** run thru file a checking sp's found in list */

seek(fida, 0, 0);  /* make sure we get whole file */
maxa = -1;  /* init to find max in file a */
while ( ( j = read(fida, list, LSTSZ * sizeof(*list))) > 0 )
  j /= 2;  /* j = number of ints read */
lisptr = list;
while(j-- && (spid = *lisptr++) != -1)
  maxa = maxa > spid ? maxa : spid;
  check[spid] |= MASKA;
if(spid == -1)
  break;

/***** run thru file b checking sp's found in list */

seek(fidb, 0, 0);  /* make sure we get whole file */
maxb = -1;  /* init to find max in file b */
while ( ( j = read(fidb, list, LSTSZ * sizeof(*list))) > 0 )
  j /= 2;  /* j = number of ints read */
lisptr = list;
while(j-- && (spid = *lisptr++) != -1)
  maxb = maxb > spid ? maxb : spid;
  check[spid] |= MASKB;
if(spid == -1)
  break;

/***** now run thru the check array to get output file */
listptr = list;
endlist = list + LSTSZ;
checkptr = check;
seek(fidco(0,0));  /* start at the beginning of the file */
count = 0;  /* init count to zero */

switch(opcode)
{
  case AND:
    maxc = maxa < maxb ? maxa : maxb;
    for(j = 0; j <= maxc; j++)
      {  
        if(*checkptr++ == (MASKA | MASKB))
          {
            *listptr++ = j;
            count++;
            if(listptr >= endlist)
              write(fidco(listptr = list),(endlist - list) * 2);
            }
      break;
    /* end case AND */
  case OR:
    maxc = maxa > maxb ? maxa : maxb;
    for(j = 0; j <= maxc; j++)
      {  
        if(*checkptr++ != 0)
          {
            *listptr++ = j;
            count++;
            if(listptr >= endlist)
              write(fidco(listptr = list),(endlist - list) * 2);
            }
      break;
    /* end case OR */
  case EXCEPT:
    maxc = maxa;
    for(j = 0; j <= maxc; j++)
      {  
        /* if a and not b */
        if((*checkptr & MASKA) & !(*checkptr & MASKB))
          {
            *listptr++ = j;
            count++;
          }
  }
if (listptr >= endlist)
    write(fidc, (listptr - list), (endlist - list) * 2);
}
checkptr++;
break; /* end case EXCEPT */

/******* terminate list and write out the remainder */
*listptr++ = -1;
write(fidc, list, (listptr - list) * 2);
return(count);
Chief of Engineers
ATIN: Tech Monitor
ATIN: DAEN-HD
ATIN: DAEN-HH
ATIN: DAEN-JC
ATIN: DAEN-JM
ATIN: DAEN-CP
ATIN: DAEN-AS-L (2)

US Army Engineer Districts
ATIN: Library
Alaska
Albany
Albuquerque
Baltimore
Buffalo
Charleston
Chicago
Cincinnati
Dallas
Dayton
Detroit
Far East
Fort Worth
Galveston
Huntington
Jacksonville
Japan
Kansas City
Little Rock
Los Angeles
Louisville
Memphis
Mobile
Nashville
New Orleans
New York
Norfolk
Omaha
Philadelphia
Pittsburgh
Portland
Ripon
Rock Island
Sacramento
San Francisco
Savannah
Seattle
St. Louis
St. Paul
Tulsa
Victoria
Vicksburg
Wallace
Wallingford

US Army Engineer Divisions
ATIN: Library
Europe
Huntsville
Lower Mississippi Valley
Middle East
Middle East (Reef)
Mississippi River
New England
North Atlantic
North Central
North Pacific
Ohio River
Pacific Ocean
South Atlantic
South Pacific
Southwestern

Waterways Experiment Station
ATIN: Library

Cold Regions Research Engineering Lab
ATIN: Tech Monitor

US Government Printing Office
Receiving Service/Depositary Copies (2)

Defense Technical Information Center
ATIN: DA (12)

Engineering Societies Library
New York, NY

FESA, ATIN: Library

ETL, ATIN: Library

Engr. Studies Center, ATIN: Library

Inst. for Water Res., ATIN: Library

Army Instl. and Major Activities (COMUS)
GSA/COM - Dir., Inst., & Svcs.
ATIN: Facilities Engineer

AHROCSM
Aberdeen Proving Ground
Army Natls. and Mechanics Res. Ctr.
Corpus Christi Army Depot
Harry Diamond Laboratories
Dugway Proving Ground
Jefferson Proving Ground
Fort McPherson
Letterkenny Army Depot
MacClellan Research and Dev. Ctr.
New Cumberland Army Depot
Pueblo Army Depot
Red River Army Depot
Redstone Arsenal
Rock Island Arsenal
Savannah Army Depot
Sharpe Army Depot
Seneca Army Depot
Shippan Army Depot
Toole Army Depot
Trinity Army Depot
White Sands Missile Range

FORSCOM
FORSCOM Engineer, ATIN: AFEN-FE
ATIN: Facilities Engineers
Fort Buchanan
Fort Bragg
Fort Campbell
Fort Carson
Fort Devens
Fort Drum
Fort Hood
Fort Indiantown Gap
Fort Irwin
Fort Jackson
Fort Leonard Wood
Fort Lewis
Fort McPherson
Fort George G. Meade
Fort Irwin
Fort Polk
Fort Richardson
Fort Riley
Presidio of San Francisco
Fort Scott
Fort Stewart
Fort Wainwright
Fort Wainwright
Fort Wainwright

CSER

Chief of Engineers
ATIN: Tech Monitor
ATIN: DAEN-HD
ATIN: DAEN-HH
ATIN: DAEN-JC
ATIN: DAEN-JM
ATIN: DAEN-CP
ATIN: DAEN-AS-L (2)
Webster, Ronald Dwight
Interagency/Intergovernmental Coordination for Environmental Planning (IICEP): systems considerations / by R. D. Webster, D. E. Putnam. -- Champaign, IL : Construction Engineering Research Laboratory ; Springfield, VA : available from NTIS. 1980. 101 p.; 27 cm. (Technical report ; N-87)
