ENHANCEMENT OF CATALOG PROCESSING SYSTEM
FOR MIT SCIENCE FICTION SOCIETY
BY
CHERYL ANN WHEELER

Submitted in Partial Fulfillment of the Requirements
for the
DEGREE OF BACHELOR OF SCIENCE
IN
ELECTRICAL ENGINEERING AND COMPUTER SCIENCE
AT THE
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
June 1, 1982

Signature of Author:

Certified by: Mr. Richard Marcus, Thesis Supervisor

Accepted by: Chairman of Departmental Committee on Undergraduate Theses

Archives
MASSACHUSETTS INSTITUTE
OF TECHNOLOGY
SEP 27 1982
LIBRARIES
ABSTRACT

The aim of this thesis is the design and implementation of a data entry and manipulation system to be used to maintain a library book catalog. The system is to replace one currently used by the MIT Science Fiction Society. The MITSFS catalog, known as Pinkdex for historical reasons, has been kept in the form of computer punched cards. It has three sections of 10,000 to 15,000 cards each: an author index, a title index, and a shelf index. To obtain a hard copy of Pinkdex, it has been necessary to transport the cards to a card reader and lister; sorting has always been done by hand. The problem addressed and solved by this thesis was to create a storage and retrieval system for Pinkdex that is easy to use and more efficient than the current system.
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>2</td>
</tr>
<tr>
<td>I. The Problem</td>
<td>5</td>
</tr>
<tr>
<td>II. Background</td>
<td>6</td>
</tr>
<tr>
<td>III. Previous Attempts</td>
<td>8</td>
</tr>
<tr>
<td>IV. Research</td>
<td>9</td>
</tr>
<tr>
<td>V. Specifications, Design, and Implementation</td>
<td>11</td>
</tr>
<tr>
<td>VI. Suggestions For Future Work</td>
<td>22</td>
</tr>
<tr>
<td>VII. Conclusion</td>
<td>24</td>
</tr>
<tr>
<td>Appendix A: Operator's Guide to the</td>
<td></td>
</tr>
<tr>
<td>Computerized Piktex</td>
<td>25</td>
</tr>
<tr>
<td>Appendix B: PL/1 Code for Piktex</td>
<td>30</td>
</tr>
<tr>
<td>References</td>
<td>50</td>
</tr>
<tr>
<td>Acknowledgements</td>
<td>51</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

Figure 1: Standard Card Format 6
Figure 2: Continuation Card Format 7
Figure 3: Reformat, Data-Entry,
       Output, and Edit Operations 13
Figure 4: Sort and Merge Operations 14
I. The Problem

The MIT Science Fiction Society is, for the purposes of this discussion, a library. Although it is the largest open library of science fiction in the world (approximately thirty thousand items), it is small as libraries go. Most "real" libraries provide card catalogs to users to index the libraries' holdings. The MITSFS uses something known as Pinkdex. (1) Pinkdex has two forms. It is a listing on paper of the books owned by the MITSFS, and it is the card "deck" from which this listing is produced. The cards are ordinary computer punched cards; I say "deck" because it consists of twenty-one boxes of cards.

That is part of the problem. Twenty-one boxes of cards are inconvenient to update, edit, or list. No one wants to interfile new cards into the old ones or carry them all to a card lister to print the information on paper. An obvious solution is to put all this information on magnetic tape or in a computer memory somewhere; this would, of course, require software to manipulate the data. There are commercial systems which do this sort of thing, written especially to be used by libraries. However, there are a few more constraints involved. The main one is money. The MITSFS is a student activity and doesn't have the funds to

(1) Pinkdex was named for the person who originally put it together on cards. She was called Fuzzy Pink (because she always wore fuzzy pink sweaters). Today she is better known to the science fiction community as Fuzzy Pink Niven.
buy such a system. Also, Pinkdex is not organized according to any standard classification system, so that either commercially available library catalog software would have to be extensively customized, or the classification system currently used in the MITSFS would have to be completely changed.

II. Background

Pinkdex on paper has a column format. In order to achieve this, the cards have been punched in fields (see Figure 1):

Author field: columns 1 to 24
Title field: columns 26 to 70
Filing letter field: column 72
Shelf code field: columns 75 to 80.

|___________|b|_______________________|b|_|bb|____|

author  title  f  shelf

FIGURE 1 : STANDARD CARD FORMAT

Should an author or a title be longer than the allotted number of characters, it is broken between words and continued in the same field of the next card, but indented four spaces (see Figure 2). The filing letter is used in
case a book has more than one author. It is the first letter of the last name of the principal author. By looking at this field, a user supposedly can tell which author such a book is shelved under. Of course, the system fails when two or more of the authors have last names that begin with the same letter. The shelf code is the coding system that indicates where in the MITSFS a book is shelved. Books are "classified" according to whether they are hardcovers, paperbacks, special reserve books, and so on. The filing letter and shelf code fields of continuation cards are always blank.

| bbbb_________ | bbbb_________________________ | b | b | bb | bbbb |

  author       title

FIGURE 2 : CONTINUATION CARD FORMAT

The other important feature of Pinkdex is that it has three parts: an author index, a title index, and a shelf index. These all contain essentially the same information, but organized in a different way in each case. In particular, Authordex contains many duplicate entries. For every book that has more than one author, there is a listing for each author in Authordex. Therefore, every book that has more than one author has more than one entry in Authordex. Titledex and Shelfdex contain entries for the principal authors only. Authordex is ordered first by author, then by title, and then by shelf field; Titledex by title, author, and shelf; and Shelfdex by shelf, author, and
title. Organization by title in Shelfdex differs a little from that in Authordex and Titledex, since it is supposed to reflect the way that the books are actually shelved. Differences arise mainly in the case of books in a series, where the books may be shelved in the order they appear in the series rather than alphabetically by title.

Since Pinkdex has always been sorted by hand, many strange sorting orders have been possible. For instance, a book entitled "1979 World's Best Science Fiction" would be sorted under N for "Nineteen." However, such titles are then grouped together in yearly order rather than alphabetical order. Titles that might alphabetically fall among them (such as "Nineteen Eighty-Four") are put either before or after them. Another problem, also involving numerals, is that many titles have been abbreviated in order to fit onto one card. "Fourth Fontana Book of Great Ghost Stories" has been entered as "4TH Fontana Book of Great Ghost Stories," but it is still sorted under F for "Fourth." This phenomenon has been the biggest deterrent to people who have tried to computerize Pinkdex in the past.

III. Previous Attempts

Several people have tried to computerize Pinkdex in one way or another. Three of the most recent attempts have been made by J. Spencer Love, Charles Frankston, and Otis Bricker.
Mr. Love's project was (or perhaps I should say is) significantly more ambitious than mine, which is why it hasn't been finished. He plans to use the data base management system RDMS III, because it will be able to do many useful things, such as ignore leading articles in titles, and so on. The reason that his project has not been completed is that RDMS III does not exist as yet. When he completes his implementation of Pinkdex, it should be a significant step up. However, though he doesn't mind waiting for a perfect Pinkdex, everyone else does. As I said before, the twenty-one boxes of computer cards are very unpopular in the MITSFS.

Mr. Frankston's idea was to preserve the inconsistent sorting order in Pinkdex and design his program so that new data would be "manually" inserted. This, he says, was the only design decision he made before he ran out of time in which to do the project.

Mr. Bricker was working on a more complicated program. It was to be display-oriented to facilitate searching, and the information was to be stored in a linked-list structure. But he, too, ran out of time and inclination for the project.

IV. Research

I decided to make my version of Pinkdex an offline Pinkdex. By "offline" I mean that the stored data would be
accessed only when Pinkdex was updated. A hard copy listing would still be used as the catalog at the MIT SFS. With an "online" Pinkdex the data would be the catalog, and a terminal would be used to access it. An online Pinkdex would require a terminal or personal computer in the MIT SFS, and, quite apart from the cost, there would be some trouble in keeping random people from playing with the hardware. Pinkdex is big; the 10,000 to 15,000 entries in each of the three indexes would require a lot of memory and efficient search techniques to be really useful online. More likely to be a problem in the case of a personal computer, Pinkdex would be completely unusable whenever there was hardware trouble. Finally, at least two hardcopy listings of Sh elfdex must be produced each year for inventory, which would require access to a lister.

I also briefly considered a personal computer for an offline Pinkdex, but the hardware would see so little use that it would hardly be worth the trouble. I looked into MULTICS and CMS, since they are on-campus and easily accessible. I finally chose MULTICS because it was cheaper to use than CMS, and because it has many potentially useful software packages available. I decided to implement any necessary additional software in PL/1 because it is highly compatible with the MULTICS system and to MULTICS software. I expected to be using a database management system (MRDS) and the MULTICS Sort/Merge.

I took a radical approach to the problem of the strange
sorting orders within Pinkdex. I decided that the best thing in the long run would be to edit Pinkdex such that the abbreviations were spelled out, and to force titles that begin with numerals to sort before those that begin with letters. In this way, Pinkdex would be more internally consistent and easier for random members to deal with, besides the fact that it would be a lot easier to implement. It would also make the new Pinkdex compatible with other computer filing systems when further enhancements are sought.

V. Specifications, Design, and Implementation

I decided that I would need the following operations for my version of Pinkdex:

1) data-entry: to enter new book entries into Pinkdex

2) sort: to sort the entries

3) merge: to merge two sets of entries together

4) output: to format the entries into columns to be printed on paper

5) edit: to change entries already in Pinkdex

6) reformat: to parse the card images of Pinkdex into pieces suitable for storing in a file.
Initially, I thought I would write the new Pinkdex around the EMACS editor using LISP, because I was still thinking of the information as text rather than as data. Next I considered storing the data in a MRDS database, under the assumption that MRDS would make it easier to access and manipulate them. But it soon became clear that MRDS would be more trouble than it was worth. Essentially, MRDS is too secure for my applications. A typical call to MRDS requires about six arguments, which is rather painful. Also, I wouldn't be able to modify a MRDS record, because all the fields except filing letter would be required as key fields to specify a unique key, and key fields cannot be modified. Finally, MRDS has no sorting or merging features, and a MRDS database cannot be sorted or merged directly by Sort/Merge, since Sort/Merge operates on MULTICS record files only.

MULTICS record files appeared to be an appropriate alternative to MRDS. PL/1 provides basic features for reading, writing, and modifying records in record files in a much simpler fashion than MRDS could offer. I ended this phase of the project with a design stated in terms of specifications for sixteen modules needed to perform the required functions. At the end of the specification for each module is the name in parentheses of the operation that the module is a part of. See Figures 3 and 4 for a hierarchical diagram of these modules grouped according to the six operations.
FIGURE 3: REFORMAT, DATA-ENTRY, OUTPUT, AND EDIT OPERATIONS
FIGURE 4: SORT AND MERGE OPERATIONS
data_entry(newbookname: string): opens a record file named newbookname. It calls on the subroutine get_data to get the new data from the user's terminal. When it receives a "quit" signal from get_data, it verifies that the user wants to end the current input session, and then either continues the session or closes the file and ends.
(data-entry)

get_data(instream, outstream: file) returns (record): prompts the user via the output stream for author, title, filing letter, and shelf code. These are read from the input stream. The user is allowed to edit the current entry. The user may decide to quit the session at any time by typing a line with just a "q" on it, in which case a quit condition is signalled and the current record is discarded. Otherwise, the completed record is returned.
(data-entry)

sort_file(sort_by, infilename, outfilename: string): calls on Sort/Merge to sort file infilename according to the order given by sort_by, and write it to file outfilename. Sort_by determines which of three user-defined compare functions (author_sort, title_sort, or shelf_sort) will be used
by Sort/Merge. (sort)

merge_file(merge_by, infile1, infile2, outfile: string): calls on Sort/Merge to merge infile1 and infile2 according to the order given by merge_by, and write it to outfile. Merge_by determines which of three user-defined compare functions (authorsort, titlesort, or shelfsort) will be used by Sort/Merge. (merge)

authorsort(rec_ptr1, rec_ptr2: ptr) returns (fixed bin(1)): compares the records pointed to by rec_ptr1 and rec_ptr2, first by author using compare_authors, then by title using compare_titles, then by shelf_code. 1 is returned if the second record comes before the first, -1 is returned if the first record comes before the second, and 0 is returned if the two records are equal. (sort and merge)

titlesort(rec_ptr1, rec_ptr2: ptr) returns (fixed bin(1)): compares the records pointed to by rec_ptr1 and rec_ptr2, first by title using compare_titles, then by author using compare_authors, then by shelf_code. 1 is returned if the second record comes before the first, -1 is returned if the first record comes before the second, and 0 is returned if the two records are
equal. (sort and merge)

shelfsort(rec_ptr1, rec_ptr2: ptr) returns (fixed bin(1)): compares the records pointed to by rec_ptr1 and rec_ptr2, first by shelf code, then by author using compare_authors, then by title using compare_titles. 1 is returned if the second record comes before the first, -1 is returned if the first record comes before the second, and 0 is returned if the two records are equal. (sort and merge)

compare_authors(author1, author2: string) returns (fixed bin(1)): parses the last names and first names of the first authors of author1 and author2. Last names are compared, and then first names. 1 is returned if the second record comes before the first, -1 is returned if the first record comes before the second, and 0 is returned if the two records are equal. (sort and merge)

compare_titles(title1, title2: string) returns (fixed bin(1)): removes all punctuation except for (,),# from title1 and title2. Also expands certain abbreviations and number strings and then compares title1 and title2. 1 is returned if the second record comes before the first, -1 is returned if the first record comes before the second, and 0 is
returned if the two records are equal. (sort and merge)

make_paper_file(infilename, outfilename: string): reads a record from infilename, parses it into 81-column lines using make_out_lines, and writes them to an unstructured ASCII file. (output)

make_out_lines(out_rec: record) returns (array): using break_string, breaks author and title fields into pieces not longer than 24 characters and 45 characters. It then puts the pieces together in an array of 81-character lines, which is returned. (output)

break_string(string: string, max_length: fixed dec(2)) returns (array): breaks string between words into pieces not longer than max_length. These pieces are returned in the array. (output)

reformat(cardname, newname: string): opens file of card images cardname and output file newname. Reads one line from file cardname, and checks to see whether there is a continuation card. Author and title are parsed out of the card images, and filing letter and shelf code are parsed from the last 9 characters of the first card by parse_cols. Each "field" is delimited by a "<" and the entire entry is read as one line into file newname.
parse_cols(cols: string) returns (record): parses cols into a filing letter and a shelf code. Then checks the shelf code via check for certain codes to be changed. Returns the filing letter and shelf code in the record. (reformat)

check(shelf: string) returns (string): checks to see whether shelf is one of several shelf codes which are inverted. If so, the correct version of that code is returned. Otherwise, shelf is returned. (reformat)

dump_cards(cardfile, recfile: string): opens file cardfile of card images parsed by reformat, and output file recfile. Reads the lines from cardfile one by one into a record, which is then written into recfile. (reformat)

Notes

In addition to the above modules, I have written a user-defined input_record module for the sort, which removes secondary author entries. I have not included it here, because it depends for its operation upon an editing change in Pinkdex. I plan to put an asterisk after the last name of the principal author, and eliminate the use the filing letter. A secondary author entry will have no asterisk.
after the first author's last name, and all such entries will be removed from files being sorted by title or shelf.

One may wonder why I have chosen to do the reformatting in two steps rather than one, first parsing card images into delimited lines, and then reading those lines into a record file. At first it was merely to facilitate testing, but later, when I decided that Pinkdex would need extensive editing, it seemed that an already-parsed file would be easier to work with. Since reformat depends on column position for parsing, its operation could easily be upset if column positions were accidently altered in an editing session.

Note that I have not included the design specification for the edit module. That module is at this time incomplete. However, a preliminary design for this module follows.

Edit displays a chosen record file by calling on make_out_lines to break the records into lines suitable for display. A certain number of entries are displayed at a given time (say, 30) with entry numbers. The user must use the entry number to refer to an entry. This is because in sequential record files only the "current" record can be modified. Therefore, if 30 records are displayed, the current record is now number 30, and the previous 29 cannot be modified unless the file is closed and reopened. My idea is to let the user specify which records he wants to change.
by record number. All specified records are deleted after the user quits the edit. If rewrites are commanded, the new records will be read into a new record file using `get_data'. When the user quits, these records are sorted and merged with the edited file.

The conversion and formatting phase of the Pinkdex project will be to actually get all the data on tape and to print it out on paper. First, the card decks will be read into three files (author, title, and shelf) on MULTICS. Then they will be reformatted with the reformat module. The resulting ASCII files will be edited directly with EMACS to eliminate major inconsistencies and errors. When these files are reasonably correct, they will be converted to record files using `dump_cards'. Also to be reformatted and edited are the update cards that have not yet been interfiled with the main Pinkdex. They will be sorted and merged with the main files. Once Pinkdex is up to date, it will be converted back to an unstructured, column formatted file to be printed out on paper.

Once the initial conversion is done, I envision the usual operation of Pinkdex to follow a pattern of incremental and full updates (see Appendix A for Operator's Guide). About once a month, the Panthercom (2) will log in to MULTICS and enter the new books acquired by the MITSFS in

(2) The Panthercom is obviously the committee that takes care of Pinkdex.
that month, by using data_entry. He will then sort the new file by author. Then he will use make_paper_file to create a hard copy of the updates to be used as a supplement to Pinkdex in the MITSFS. The next month, after sorting the new entries by author, he will merge them with the entries from the previous month and make a hard copy of that file. In this way, only one file of updates is stored on MULTICS at any one time. Twice a year the main body of Pinkdex should be updated. The Panthercom will dump the main Pinkdex from tape to memory. Then he will sort the new entries file to create two more files sorted by title and shelf code. Then he will merge each new entries file with the appropriate main file, and copy the resulting files back to tape. The new files will also be dumped to paper to make a new and entirely up to date Pinkdex.

VI. Suggestions for Future Work

Elimination of the filing letter

As I mentioned earlier, I hope to eliminate the filing letter system of identifying principal authors. This should be accomplished during the editing of the main body of Pinkdex at the end of May 1982. The change will require minor alterations to most of the code I have written.
Completion of the Edit module

This is another project which I plan to finish myself during May 1982. I have designed the overall program so that certain modules of it can be used by Edit; these are get_data and make_out_lines.

Accession numbering

Most libraries assign a unique number (accession number) to each book they have. This allows the library to distinguish between multiple copies of the same book. MITSFS currently has no such system; there is no way to tell whether MITSFS has more than one copy of any book listed in Pinkdex. Once Pinkdex is in the form of record files, it will be easy to implement an accession numbering system at a later date. This would be a useful way to use a DBMS, since the accession numbers would provide a short, unique key to individual records, and it is unlikely that anyone would want to change the accession number of a book.

Standard library classification system

In the same way that a DBMS could be used to implement the storage of accession numbers, one could be used to classify the books by some standard system. As I mentioned earlier, this would pose the problem of trying to classify books which have no standard classification, e.g. foreign
books have no classification in the Library of Congress system.

Online Pinkdex

Though I have said so much against the idea of having Pinkdex online at this time, I still think it would be a good idea eventually. It would be desirable for MITSFS to be independent concerning something as vital to it as Pinkdex. It would provide better control over Pinkdex and faster corrections. There are many problems to overcome with an online Pinkdex, but the result could easily be worth it. One way to make an online Pinkdex more economical would be to use existing systems and data bases -- only selecting those book records applicable to the MITSFS and assigning the shelf code.

VII. Conclusion

The software I've written and the changes in format and sorting order that it will require are a step forward for Pinkdex. I hope, though, that the MITSFS will not be content with this as a final state for Pinkdex. What I've really done is not to make a final product, but to make the next step up easier for someone else to do. Hopefully, the MITSFS won't wait for a recurrence of the present situation to make further improvements to Pinkdex.
APPENDIX A

OPERATOR'S GUIDE TO THE COMPUTERIZED PINKDEX
The reader is assumed to be familiar with the MULTICS system.

There are four operations that can be performed using the computerized Pinkdex. They are data entry, sorting, merging, and output. They are all invoked from command level.

Data entry prompts the user for the author, title, filing letter, and shelf code subentries. After typing each of these, type a line with just a "." on it. This tells the program that you are finished typing the subentry. At any time before you type the "." line after the shelf code, the entry may be altered. Type a line with a "/" on it to start the current subentry over. Type "//" to go back to the previous subentry, "///" to go back two subentries, and so on. You may quit data entry at any time by typing a line with just a "q" on it. If you are in the middle of an entry, the entry is discarded. The file containing the new entries is a MULTICS record file, not suitable for display or direct editing.

Sort sorts a MULTICS record file in an order specified by the user. If the order specified is not one of "author", "title", or "shelf" then the program halts. Otherwise, when the file has been sorted, the program prints "xx records sorted" at the user's terminal, xx being the number of records which were sorted, obviously.
Merge merges two MULTICS record files. The files must both be in the same order, and that order must be specified when the merge is performed. The order may be one of "author", "title", or "shelf". If it is not one of these then the program will halt. Otherwise, when the files have been merged, the program will print "xx records merged" at the user's terminal.

Output converts a MULTICS record file to a column formatted unstructured file suitable for display, editing, or printing. After this file has been created, use one of the MULTICS output commands to print it on paper.

Data Entry

usage:

data_entry newfilename

or

de newfilename

where:

newfilename is the path name of a MULTICS record file to be created. The new entries will be stored in this file.

Sort

Page 27
usage:

    sort_file sort_by infile outfile

or

    sf sort_by infile outfile

where:

    sort_by specifies whether the input file is to be
    sorted by author, title, or shelf. Possible values are:
    "author", "title", or "shelf".

    infile is the path name of a MULTICS record file to be
    sorted.

    outfile is the path name of a MULTICS record file which
    is to contain the sorted data.

Merge

usage:

    merge_file merge_by infile1 infile2 outfile

or

    mf merge_by infile1 infile2 outfile

where: merge_by specifies whether the input files are
sorted by author, title, or shelf. Possible values for
merge_by are "author", "title", or "shelf".

    infile1 and infile2 are the path names of the two
    MULTICS record files to be merged.
outfile is the path name of a MULTICS record file which will contain the merged data.

Output

usage:

make_paper_file infile outfile

or

mpf infile outfile

where: infile is the path name of the MULTICS record file to be converted to column format.

outfile is the path name of the unstructured file which will contain the column formatted output.
reformat: proc(cardname, newname);

/* This module reads card images one by one from an unstructured file */
/* and parses them into author, title, filing letter, and shelf code */
/* fields. These fields are concatenated into one string with delimeters */
/* "<" between the fields. The string is written into another unstruc- */
/* tuned file. */

dcl (cardname, newname) char(*) parm;

dcl (endfile condition,
 (x,y) fixed dec(2);
 (cardfile, newfile) file,
 dumb_var char(400) varying,
 (a1,a2) char(128) varying,
 (t1,t2) char(256) varying,
 (f1,d,b) char(1) varying,
 s1 char(8) varying,
 (Z1,Z2) char(9) varying,
 (card, next_card) char(80) varying,
 01 f.s,
 02 f char(1) varying,
 02 s char(8) varying);

d"=";
b:= " ";
open file(cardfile) title("vfile_ "; |cardname|) input stream;
open file(newfile) title("vfile_ "; |newname|) output stream;
read file(cardfile) into(card);
on endfile(cardfile) goto END;
a1=rtrim(substr(card, 1, min(length(card), 25)));
if (length(card)<25) then t1="";
else t1=rtrim(substr
 (card, 26, min(length(card)-25,46));
if (length(card)<71) then Z1="";
else Z1=rtrim(substr(card, 72));
do while ("f"b);
do while ("f"b);
read file(cardfile) into(next_card);
a2=rtrim(substr(next_card, 1, min(length(next_card), 25)));
if (length(next_card)<25) then t2="";
else t2=rtrim(substr
 (next_card, 26, min(length(next_card)-25,46));
if (length(next_card)<71) then Z2="";
else Z2=rtrim(substr(next_card, 72));
if (Z2="\"\") then do;
    a2=1trim(a2);
    t2=1trim(t2);
    if (a2="\"\") then a1=a1||b||a2;
    if (t2="\"\") then t1=t1||b||t2;
end;
else goto EXIT;
end;
EXIT:
    f_s=parse_cols(Z1);
    f1=f_s.f;
    s1=check(f_s.s);
    dumb_var=a1||d||t1||d||f1||d||s1;
    write file(newfile) from(dumb_ar);
    a1=a2;
    t1=t2;
    Z1=Z2;
end;
END:
    close file(cardfile);
    close file(newfile);

parse_cols: proc(cols) returns(01,
    02 char(1) varying,
    02 char(8) varying);
/* This module parses the last 9 characters of a primary card image */
/* into the filing letter and shelf code. It also strips away an * */
/* obsolete attempt at counting the number of books owned by the MITSFS. */
dcl cols char(9) varying parm;
dcl (other char(8) varying,
    01 f_shelf,
    02 f_col char(1) varying,
    02 s_col char(8) varying);
other = before(cols, " ");
if (length(other)>1) then do;
    f_shelf.s_col = other;
    f_shelf.f_col = " ";
end;
else do;
    f_shelf.s_col = 1trim(after(cols, " "));
    f_shelf.f_col = other;
end;
f_shelf.s_col = before(f_shelf.s_col, ", ");
return(f_shelf);
end;

check: proc(shelf) returns(char(8) varying);
/* This module checks the shelf code for certain codes which have been */
/* Inverted due to laziness on the part of the people who used to take */
/* care of Pinkdex. The proper code is returned. */
dcl (shelf) char(8) varying parm;
if shelf="P-SR" then shelf ="SR-P";
if shelf="H-SR" then shelf ="SR-H";
if shelf="PA-SR" then shelf ="SR-PA";
if shelf="MA-SR" then shelf ="SR-MA";
if shelf="LP-SR" then shelf ="SR-LP";
return(shelf);
end;
end;
dump_cards:proc(cardfile, recfile);
/* This module reads one line from an unstructured file. It parses the */
/* line into author, title, filing letter, and shelf code by looking */
/* for delimiter "<" between the fields. The information is written to */
/* a record, which is then written to a record file. */
    dcl (cardfile, recfile) char(*) parm;
    dcl (line char(400) varying,
        01 rec,
            02 author char(128) varying,
            02 title char(256) varying,
            02 filing char(1) varying,
            02 shelf char(8) varying,
        endif condition,
        (oldfile, newfile) file);
    open file(oldfile) title("vfile_" || cardfile) input stream;
    open file(newfile) title("vfile_" || recfile) sequential
        output record;
    do while("1b");
        read file(oldfile) into(line);
        on endfile(oldfile) goto QUIT;
        rec.author=before(line, "<");
        line=after(line, "<");
        rec.title=before(line, "<");
        line=after(line, "<");
        rec.filing=before(line, "<");
        line=after(line, "<");
        rec.shelf=after(line, "<");
        write file(newfile) from(rec);
    end;
QUIT:    close file(oldfile);
    close file(newfile);
end;
de:
data_entry: proc(newbookname);
/* This module opens a new record file and asks for new data through */
/* the user's terminal. The new data is written to a record, which */
/* is then written to the file. */
dcl newbookname char(*) parm;
dcl ((instream, outstream, newbooks) file,
quit condition;
 01 new_rec,
 02 a char(128) varying,
 02 t char(256) varying,
 02 f char(1) varying,
 02 s char(8) varying,
 02 b char(80) varying,
 01 quit char(15) init("quit? (y or n) "),
    repeat char(21) init("please answer y or n "));
open file(newbooks) title("vfile_ "||newbookname)
  sequential output record;
on file(instream) title("syn_user_input") input;
on file(outstream) title("syn_user_output") output;
do while ("1"b);
  new_rec=get_data(instream, outstream);
on quit goto LOOP;
write file(newbooks) from(new_rec);
end;
LOOP: write file(outstream) from(quitter);
  read file(instream) into(temp);
  if (temp="y") then goto EXIT;
  if (temp="n") then do;
    write file(outstream) from(repeat);
    goto LOOP;
  end;
EXIT: close file(newbooks);
close file(instream);
close file(outstream);
get_data: proc(instream, outstream) returns(01,
  02 char(128) varying,
  02 char(256) varying,
  02 char(1) varying,
  02 char(8) varying);

/* This module prompts the user for an author, a title, a filing letter */
/* and a shelf code. The user is permitted to alter any part of the current */
/* entry. When the entry is completed, the user types a line with a period */
/* on it. The record is then returned to the calling procedure. The user */
/* may decide to quit at any time by typing a line with a q on it. In that */
/* case, the current record is discarded and a quit condition is signalled. */

dcl (instream, outstream) file parm;

dcl (temp char(80) varying,
  quit condition,
  01 new_rec,
    02 author char(128) varying,
    02 title char(256) varying,
    02 f char(1) varying,
    02 shelf char(8) varying,
  astring char(8) init("author: "),
  tstring char(7) init("title: "),
  fstring char(15) init("filing letter: "),
  sstring char(7) init("shelf: "));

TOP:
new_rec.author="";
  temp="";
  write file(outstream) from(astring);
  do while (temp=".");
    new_rec.author=new_rec.author||temp;
    read file(instream) into(temp);
    if temp = "/" then goto TOP;
    if temp = "q" then signal quit;
  end;

TITLE:
  temp="";
  new_rec.title="";
  write file(outstream) from(tstring);
  do while (temp=".");
    new_rec.title = new_rec.title||temp;
    read file(instream) into(temp);
    if temp = "/" then goto TITLE;
    if temp = "///" then goto TOP;
if temp = "q" then signal quit;
end;

FILE:
  temp="";
new_rec.f="";
write file(outstream) from(fstring);
do while(temp=".");
  new_rec.f=new_rec.f||temp;
  read file(instream) into(temp);
  if temp = "/" then goto FILE;
  if temp = "///" then goto TITLE;
  if temp = "///" then goto TOP;
  if temp = "q" then signal quit;
end;

SHELF:
  temp="";
new_rec.shelf="";
write file(outstream) from(sstring);
do while (temp=".");
  new_rec.shelf=new_rec.shelf||temp;
  read file(instream) into(temp);
  if temp = "/" then goto SHELF;
  if temp = "///" then goto FILE;
  if temp = "///" then goto TITLE;
  if temp = "///" then goto TOP;
  if temp = "q" then signal quit;
end;

return(new_rec);
end;

end;
mpf:
make_paper_file:proc(infilename, outfilename);
/* This module reads records from a record file one by one and formats */
/* them in 8i-character lines with columns. The lines are written to an */
/* unstructured file. */
dcl (infilename, outfilename) char(*) parm;
dcl ((infile, outfile) file,
endfile condition,
  01 inrec,
    02 author char(128) varying,
    02 title char(256) varying,
    02 filing char(1) varying,
    02 shelf char(8) varying,
    lines(5) char(81),
    i fixed bin(24));
open file(infile) title("vfile_"||infilename) sequential
  input record;
open file(outfile) title("vfile_"||outfilename) output stream;
do while("""
  read file(infile) into(inrec);
on endfile(infile) goto CLOSE;
  lines = make_out_lines(inrec);
  do i = 1 by 1 to 5:
    if lines(i) = (81) " then i=5;
    else write file(outfile) from(lines(i));
  end;
end;
CLOSE:
close file(infile);
close file(outfile);

make_out_lines: proc(out_rec) returns (dim(5) char(81));
/* This module creates the lines which are written to the file. The */
/* author and title fields are broken into more than one line if necessary */
/* and the filing letter and shelf code are put on the first line. The lines */
/* are returned as an array. */
dcl 01 out_rec parm,
  02 author char(128) varying,
  02 title char(256) varying,
02 filling char(1) varying,
02 shelf char(8) varying;

dcl (char_10 char(10) init((10)" ") ,
     out_array(5) char(81) init("",",",",",",") ,
     A(5) char(25) init("",",",",",") ,
     T(5) char(46) init("",",",",",") ;
(aparts, tparts) char(256) varying ,
    f char(2) ,
    s char(8) ,
    i fixed dec(2) ;
if (length(out_rec.author)>24)
then A=break_string(out_rec.author,24) ;
else A(i)=out_rec.author ;
if (length(out_rec.title)>45)
then T=break_string(out_rec.title,45) ;
else T(i)=out_rec.title ;
f=out_rec.filing ;
s=out_rec.shelf ;
out_array(i) = A(i)||T(i)||f||s ;
do i = 2 by 1 to 5 ;
if (A(i)=(25) " ")& (T(i)=(46) " ") then goto QUIT ;
else out_array(i) = A(i)||T(i)||char_10 ;
end ;

QUIT: return(out_array ) ;

break_string: proc (string,max_length) returns (dim(5) char(*/
/* This module breaks a string between words into pieces not longer than */
/* max_length. The pieces are returned in an array. */
/*
dcl (string char(256) varying ,
     max_length fixed dec(2)) parm ;
dcl j fixed dec(2) ;
dcl pieces(5) char(max_length) varying ;
pieces(1)=before(string," ");
string=after(string," ");
do while("1"b);
   if ( ((length(pieces(1)) + length(before(string," ")))+1
      > max_length) then goto EXIT ;
   else do ;
      pieces(1)=pieces(1)||" ";
      before(string," ");
   end ;
   pieces(1)=pieces(1)||" ";
   after(string," ");
end ;
return (pieces ) ;

EXIT: return ( ) ;
string=after(string," ");
end;

end;

EXIT:  
j=2;
do while("1"b);
    pieces(j)=before(string," ");
    string=after(string," ");
do while("1"b);
    if ((length(pieces(j)) +length(before(string," "))+5) >max_length) then goto EXIT2;
else do:
    pieces(j)=pieces(j)||" "||before(string," ");
    string=after(string," ");
end;

exit2:  
pieces(j)=(4)" "||pieces(j);
if ((length(string)+4)<=max_length)
    then do:
    pieces(j+1)=(4)" "||string;
    goto EXIT3;
end;
j=j+1;
end;

EXIT3:  return(pieces);
end;

end;

end;
sort_file:proc(sort_by, infilename, outfilename);
/* This module calls Sort/Merge to sort a file in a specified order. */
/* User-defined compare modules are used to sort in the proper order. */

dcl (infilename, outfilename) char(*),
    sort_by char(*)) parm;

dcl (files(1) char(length(infilename)+4) init("-if ";infilename),
outputfile char(length(outfilename)+4)
    init("-of ";outfilename),
sm_desc(3) ptr,
    code fixed bin(35),
sort_$noexit entry external,
no2authors entry(ptr, fixed bin(21), fixed bin, bit(1)),
01 exits,
  02 version fixed bin init(1),
  02 compare entry,
  02 input_record entry init(no2authors),
  02 output_record entry init(sort_$noexit),
(authorsort, titlesort, shelfsort) entry(ptr, ptr)
returns(fixed bin(1)),
sort_entry((*)char(*), char(*), (*ptr, char(*),
    char(*), float bin(27), fixed bin(35)));

if sort_by="author" then do;
exits.compare=author_sort;
exits.input_record=sort_$noexit;
end;
else if sort_by="title" then exits.compare=titlesort;
else if sort_by="shelf" then exits.compare=shelfsort;
else goto END;

sm_desc(1)=null();
sm_desc(2)=addr(exits);
sm_desc(3)=null();

call sort_ (files, outputfile, sm_desc, ",", ",", 0, code);
END: end;
mf:
merge_file:proc(merge_by, infile1, infile2, outfile):
    /* This module calls Sort/Merge to merge two files according to a */
    /* specified order. User-defined compare functions are used to merge */
    /* the files in the proper order. */
    dcl (infile1 char(*),
         infile2 char(*),
         outfile char(*),
         merge_by char(*) ) parm;
    dcl (files(2) char(max(length(infile1), length(infile2))+4) varying
     init("-if " || infile1 , "-if " || infile2 ),
     outputfile char(length(outfile)+4) init("-of " || outfile),
     sm_desc(2) ptr, 
     code _ fixed bin(35),
     merge$noexit entry external,
     01 exits,
     02 version fixed bin init(1),
     02 compare entry,
     02 input_record entry init(merge$noexit),
     02 output_record entry init(merge$noexit),
     (authorsort, titlesort, shelfsort) entry(ptr, ptr)
      returns(fixed bin(1)),
     merge_entry(****char(*), char(*), (**ptr, char(*),
      fixed bin(35)));

    if merge_by="author" then exits.compare=authorsort;
    else if merge_by="title" then exits.compare=titlesort;
    else if merge_by="shelf" then exits.compare=shelfsort;
    else goto END;

    sm_desc(1)=null();
    sm_desc(2)=addr(exits);

    call merge_(files, outputfile, sm_desc, ",", code);

    END: end;
authorsort:
proc(rec_ptr1, rec_ptr2) returns(fixed bin(1));
/* This module compares two records by author, title, and shelf, and */
/* returns a value telling which record ranks first. */
dcl (rec_ptr1, rec_ptr2) ptr parm;
dcl (result fixed bin(1),
  01 rec1 based(rec_ptr1),
    02 author char(128) varying,
    02 title char(256) varying,
    02 filing char(1) varying,
    02 shelf char(8) varying,
  01 rec2 based(rec_ptr2),
    02 author char(128) varying,
    02 title char(256) varying,
    02 filing char(1) varying,
    02 shelf char(8) varying,
  (compare_authors, compare_titles) entry(char(*), char(*))
returns(fixed bin(1)));
result = compare_authors(rec1.author, rec2.author);
if result ^=0 then return(result);
result = compare_titles(rec1.title, rec2.title);
if result ^=0 then return(result);
if rec1.shelf>rec2.shelf then return(1);
if rec1.shelf<rec2.shelf then return(-1);
return(0);
end;
titlesort: proc(rec_ptr1, rec_ptr2) returns(fixed bin(1));
/* This module compares two records by title, author, and shelf code. */
/* It returns a value telling which record ranks first. */
dcl (rec_ptr1, rec_ptr2) ptr parm;
dcl (result fixed bin(1)),
  01 rec1 based(rec_ptr1),
    02 author char(128) varying,
    02 title char(256) varying,
    02 filing char(1) varying,
    02 shelf char(8) varying,
  01 rec2 based(rec_ptr2),
    02 author char(128) varying,
    02 title char(256) varying,
    02 filing char(1) varying,
    02 shelf char(8) varying,
  (compare_authors, compare_titles) entry(char(*), char(*))
    returns(fixed bin(1)));
result = compare_titles(rec1.title, rec2.title);
if result^=0 then return(result);
result=compare_authors(rec1.author, rec2.author);
if result^=0 then return(result);
if rec1.shelf>rec2.shelf then return(1);
if rec1.shelf<rec2.shelf then return(-1);
return(0);
end;
shellsort:proc(rec_ptr1, rec_ptr2)returns(fixed bin(1))
/* This module compares two records by shelf code, author, and title. */
/* A value is returned telling which record ranks first. */
dcl (rec_ptr1, rec_ptr2) ptr parm;
dcl (result fixed bin(1),
   (title1, title2) char(256) varying,
   01 rec1 based(rec_ptr1),
      02 author char(128) varying,
      02 title char(256) varying,
      02 filing char(1) varying,
      02 shelf char(8) varying,
   01 rec2 based(rec_ptr2),
      02 author char(128) varying,
      02 title char(256) varying,
      02 filing char(1) varying,
      02 shelf char(8) varying,
   (compare_authors, compare_titles) entry(char(*),char(*))
   returns(fixed bin(1)));
if rec1.shelf>rec2.shelf then return(1);
if rec1.shelf<rec2.shelf then return(-1);
result=compare_authors(rec1.author, rec2.author);
if result=0 then return(result);
if index(rec1.title, "#") = 0 then title1 = rec1.title;
   else if index(rec1.title, "(*)=0 then title1 = rec1.title;
   else title1=before(after(rec1.title, "(*)" ));
if index(rec2.title, "#") = 0 then title2 = rec2.title;
   else if index(rec2.title, "(*)=0 then title2 = rec2.title;
   else title2=before(after(rec2.title, "(*)" ));
result = compare_titles(title1, title2);
return(result);
end;
compareauthors:proc(author1, author2) returns(fixed bin(1));
/* This module compares two authors. Last names and first names */
/* are parsed out of the author strings and compared. A value is */
/* returned telling which author ranks first. */
dcl (author1, author2) char(*) parm;
dcl author(2) char(128) varying init(author1, author2);
dcl (last(2) char(128) varying,
    first(2) char(128) varying,
    x,y) fixed bin(24,0);
    i fixed dec(1));
if author(1)=author(2) then return(0);
do i=1 by 1 to 2;
   last(i)=before(author(i), ",");
   last(i)=before(last(i), "");
   x=index(author(i), ",");
   y=search(after(author(i), ",", ",", ",");
   if y=0 then first(i)=1trim(after(author(i), ",", ",");
   else first(i)=1trim(substr(author(i), x+1, y-1));
end;
if last(1)>last(2) then return(1);
if last(1)<last(2) then return(-1);
if first(1)>first(2) then return(1);
if first(1)<first(2) then return(-1);
return(0);
end;
compare_titles: proc(title1, title2) returns(fixed bin(1));
/* This module compares two titles. All punctuation except for */
/* (), . , and # are removed. Certain abbreviations are expanded, */
/* and number strings are modified to produce numerical ordering. */
/* A value is returned telling which title ranks first. */
dcl (title1, title2) char(*) parm;
dcl title(2) char(256) varying init(title1, title2);
dcl (i, j) fixed bin(24),
 (x, len, counter) fixed bin(24,0),
 (a, n) char(1),
 b char(1) init(" "),
 strg condition,
 newtitle(2) char(256) varying init(" . "),
 abb(5) char(5) varying
 init(" DR. ", " MR. ", " F ", " SF ", " & "),
 full(5) char(15) varying
 init(" DOCTOR ", " MISTER ", " FANTASY ",
 " SCIENCE FICTION ", " AND ");
if title1=title2 then return(0);
do i=1 by 1 to 2;
do j=1 by 1 to 5;
do while("1"b);
x=index(title1, abb(j));
if x=0 then goto EXIT;
else title1=before(title1, abb(j))
   || full(j)|| after(title1, abb(j));
end;
EXIT:
len=length(title1);
do j=1 by 1 to len;
a=substr(title1,1,1);
title1=substr(title1,2);
if index("/", a)=0
   then newtitle1 = newtitle1||a;
if (a="A")&(a="Z")
   then newtitle1 = newtitle1||a;
if (a="-")&(a=" ")
   then newtitle1 = newtitle1||a;
if (a="0")&(a="9") then do;
n=a;
do while ((n>"0")&(n<="9"));
counter=counter+1;
end;

if n=substr(title(i), counter, 1);
on str n=" ":
end;
newtitle(i)=newtitle(i);substr(collapse(),
counter,1);a;substr(title(i),1,
counter-1);
title(i)=substr(title(i), counter);
j=j+counter-1;
end;
end;

if newtitle(1)>newtitle(2) then return(1);
if newtitle(1)<newtitle(2) then return(-1);
return(0);
no2authors: proc(rec_ptr, rec_len, action, close_exit_sw):
/* This module is a user-defined input record module for the sort. */
/* It identifies secondary author entries to the calling function. */
dcl (rec_ptr ptr,
   rec_len fixed bin(21),
   action fixed bin,
   close_exit_sw bit(1)) parm;
dcl (01 rec based(rec_ptr),
     02 author char(128),
     02 title char(256),
     02 shelf char(8),
     last char(32));
if index(author, "**")=0 then action=0;
else do:
   last=before(author, ",,.");
   if index(last, "**")=0 then action=1;
   else action=0;
end;
end;
REFERENCES

Honeywell Information Systems Incorporated


ACKNOWLEDGEMENTS

I would like to thank the following people for all their personal help and moral support during the trauma of completing a thesis:

Charles Frankston
Wayne Gramlich
Michael Guenette
Lyman Hazleton
Fred Helenius
Barry Margolin
Jay Pattin
Wendy Rowe
Jeffrey Schiller
William York

In addition, I'd like to thank the following for making my thesis and graduation possible:

Mr. Richard Marcus
MIT Department of Electrical Engineering and Computer Science
MIT Student Information Processing Board
MIT Science Fiction Society
Fuzzy Pink Niven