The Division of Systems and Communications of the New Jersey Department of Labor and Industry is committed to the use of high level “productivity aid” systems in new applications wherever feasible. As part of that plan, SAS was installed several years ago and made available to both professional programmers and users. The use of SAS has decreased the time required to implement new applications, but has led to some problems that we are now in the process of solving.

Among the problems: Our professional programmers are proficient in COBOL and assembler language, but often have little or no familiarity with languages which syntactically resemble SAS. The user community members often have no programming experience at all, yet the desire to get professional programmers and users. The use of SAS has decreased the time required to implement new applications, but has led to some problems that we are now in the process of solving.

The Division has attacked these problems in three ways. First, a standardized coding style and technique has been developed. Second, a local SAS user’s manual is being produced which illustrates the programming techniques we want people to use and which includes information on statements incorporated in Release 79.3 such as DO-WHILE and DO-UNTIL. The manual will eventually include chapters on creating data sets, sorting records, recoding variables, data management, producing custom reports, using SAS PROCs, testing, and documentation. Third, we intend to offer internal training seminars when the manual is complete.

The techniques we have developed incorporate the salient principles of structured coding. First, program development and refinement should proceed from the general to the specific; second, when the refinement process is complete, each step thus identified should be coded as a program segment with one entry point and one exit point; and third, the code should be indented to reflect the structure of the process. In addition, to these general principles, we have set down several rules to be followed for program clarity, ease of maintenance, and reusability of code segments. For example, DATA steps begin with a "(KEEP = ...)") list to show what variable names are used and to avoid including unneeded variables in the data set thus created. SAS statements are grouped together at the beginning of the DATA step in the prescribed order of FORMAT, RETAIN, and ARRAY. When a program requires constants, they are associated with symbolic names in assignment statements placed just before a RETAIN statement. The assignment statements are put inside an "IF N = 1 THEN DO; ... END;" structure and the symbolic names are then listed in a RETAIN statement. Therefore, the assignment statements are executed only once. Macros are used whenever the same set of FORMAT, RETAIN, and ARRAY statements will be required in more than one DATA step. LINK-NEXT is utilized to define and call subroutines whenever the processing constitutes a well-defined step in the logic requiring a relatively large number of lines of code. Finally, SAS statements are written in columns 1-45 and the scan is limited to these columns using an "S-45" option. Columns 46-72 are used for comments which explain the purpose of each segment and identify symbolic names more fully.

SOME EXAMPLES

To illustrate some of these coding principles, four approaches are shown to the problem of selecting records having specified values of a selector variable. The heart of any selection process is an IF-THEN statement. If only one or two target values are to be selected, it is a simple matter to list the values in a multiple IF-THEN-ELSE structure or a compound IF statement. This approach, however, becomes cumbersome when there are a large number of target values. An alternative is to construct an array containing the target values and then search the array until a match between a selector variable and an array element occurs.

Binary Array Search

The first example illustrates binary search of the target array. This example illustrates our full production documentation style.

The main program contains a short main segment and two subroutines. In the main program, the data set is identified and the variables it is to contain are named in a KEEP list. Next, on the first pass only, subroutine LOADLIST is called to create the target array. Then a record is read from the master file. The last event is to call a subroutine which carries out the binary search.

Subroutine LOADLIST begins by defining two control variables and a set of array element names. Values of these variables are retained throughout DATA step execution. The INFILE statement refers to an OS data set containing the target values in ascending sequence. The DO-WHILE segment reads the target values and stores them in successive locations in the array. That process continues until the end of the
The target data set is reached. Next, the value of MAXLOCTN is set to one more than the array size. This is necessary to assure that the last element will be tested. Next, HBOUND becomes the ceiling of the base 2 logarithm of the number of elements. HBOUND is used later to limit the number of examinations of the target array.

Subroutine BINSERCH begins by setting LOW to zero and HIGH to MAXLOCTN. The search process is under control of a DO-loop. The process is terminated if LOW comes to within 1 of HIGH, because a match did not occur. POINT (the array index variable) is computed as the truncated midpoint of LOW and HIGH. A multiple IF-THEN-ELSE examines the relationship between the search variable and the target array element designated by POINT. If the search variable is too high, the left half of the array (the half below POINT) is eliminated; if the search variable is too low, the right half is eliminated. If a match is found, the input record is OUTPUT, and the loop control variable is set to its termination value to end the search.

**EXAMPLE 1. BINARY ARRAY SEARCH**

```sas
OPTIONS 5=45 OBS=5000;
************************************************************************
* DS * PROGRAM LOGIC * COMMENTS *
************************************************************************
DATA SELRECS
  KEEP = LASTTHREE PC REFCODE);
  IF _N_ = 1 THEN
    LINK LOADLIST;
    INFERENCE MASTFILE(SUGI(COMP));
    INPUT
      @7 LASTTHREE 3.
      @10 PC 2.
      @14 REFCODE $2.;
    LINK BINSERCH;
    RETURN;
************************************************************************
* LS * SEARCH TABLE LOAD SUBROUTINE * COMMENTS *
************************************************************************
LOADLIST:
  * LOAD LIST ENTRY POINT *
  * HOLD VALUES OF THESE *
  * KEEP VARS THRU DATA STEP *
  * DEFINE SEARCH ARRAY *
  * INPUT SEARCH ARGUMENTS *
  * FROM OS FILE *
  * INIT ELEMENT COUNTER *
  * WHILE NOT END-OF-FILE *
  * INCREMENT ELEMENT CNT *
  * READ SEARCH ARGUMENT *
  * EXECUTION *
  * ARRAY SIZE PLUS 1 *
  * MAX DECISIONS IS LOG2 *
RETURN;
************************************************************************
```
Example 1 continued.

```
* LS *
* BINARY SEARCH SUBROUTINE *
* *
;*
BINSEARCH:
*
LOW = D; HIGH = MAXLOCTN;
*
DO SEARCH = 1 TO HBOUND;
IF HIGH = LOW+1 THEN DELETE;
POINT = INT((HIGH+LOW)/2);
IF LASTHREE > SELCOTB THEN
  LOW = POINT;
ELSE IF LASTHREE < SELCOTB THEN
  HIGH = POINT;
ELSE /* LASTHREE = SELCOTB */
  DO;
    OUTPUT;
    SEARCH = HBOUND;
    END: /* ELSE */
  END:
END: /* DO */
RETURN:

* PS *
* PRINT THE DATA SET *
* *
;
PROC PRINT
DATA = SELRECS;

Hash Array Look-up

The binary search approach requires the array contents to be sorted. This requirement can be avoided by storing array contents in locations determined by the outcome of a hashing formula, as shown in Example 2.

For this example we have used separate DATA steps to load the array and select the records instead of subroutines. The array structure is defined in a macro to avoid writing these statements separately in each DATA step. A RETAIN statement defines PRIME as a constant for the hashing formula, and initializes the array contents to blanks. A character array is utilized because empty elements must be designated by a nonnumeric entry (in this case blanks). The array is several times larger than the number of nonblank search elements because space is needed to accommodate targets which may hash to the same location.

The CREATABL step reads each target value twice — once in positive integer binary (PIB) format and once as a character string. The PIB value and the PRIME constant are used in a remaindering function to compute a location (array index "POINT") between 1 and "PRIME". If an element has already been stored in that location, the DO-WHILE segment increments the array index until an empty location is found.

The SELRECS step utilizes the macro to define the array structure and brings in the contents of CREATABL on the first pass. Next, a master file record is read. As before, the search variable is read as both a PIB number and a character string, and the same hashing formula is used to determine an array location. In the DO-WHILE loop, the nonempty array elements and the search variable are compared. If a match occurs the record is output and the WHILE loop is exited. This restricted use of GOTO is not damaging to program clarity as long as the transfer is to the first statement after the end of the loop. A linear search for a match continues until either a match is found or an empty element is encountered. In the latter case, the input record is not written to the data set.

# 117
**EXAMPLE 2. HASH ARRAY SEARCH**

```
OPTIONS S=45 O BS=5000;
MACRO ARRAYDEF
  RETAIN PRIME 397
  LC1-LC500
  ARRAY SELCODTBIPOINTI $ 3 LCI-LC500;
  MANDRA
  DATA CREATABL (KEEP = LCI-LC500);
  INFILE HASHTAB{PULLRECS
  END = ENDTABLD;
  ARRAYDEF
  INPUT
    #1 CODE  P1B3.
    #1 LASTHREE $3.
    POINT = MOD{CODE,PRIMEI + 1;
  DO WHILE (SELCODTB ~='');
    POINT + 1;
    ENDTABLD THEN OUTPUT CREATABL;
  DATA SELRECS
    (KEEP = LASTHREE PC REFCODE);
  IF N = 1 THEN DO;
    ARRAYDEF
  SET CREATABL;
  END;
  INFILE MASTFILE (SUGICOMP);
  INPUT
    #7 CODE  P1B3.
    #10 LASTHREE $3.
    #10 PC 2.
    #14 REFCODE $2.
    POINT = MOD{CODE,PRIMEI + 1;
  DO WHILE (SELCODTB ~='');
    IF LASTHREE = SELCODTB THEN DO;
      OUTPUT;
      GOTO EXIT;
      END;
      END;
      GOTO EXIT;
  END: /* WHILE */
  EXIT: RETURN;
  PROC PRINT
  DATA = SELRECS;
```

**Linear Array Search**

A more straightforward approach than binary search or hashing is to create the target array and search it from the beginning until a match is found. This linear approach is shown in Example 3.

The array size is coded as a constant so that, if the number of targets changes, only the statements in lines 3, 4, and 5 need to be altered. Subroutine ARRLOAD fills the target array. A DO-loop compares each target element to the search variable. If a match is found the record is output and loop execution terminated by a transfer.

**COMPARISON OF TECHNIQUES**

An advantage of all of these techniques is that the target values are read from an OS data set. If new target values are to be selected, the programs do not need to be modified. The target data set could be created by another program in a system, and the same record selection program could produce new output without programmer intervention. The most obvious approach to record selection is to list the target values in a compound IF statement as shown in Example 4.
EXAMPLE 3. LINEAR ARRAY SEARCH

OPTIONS S=45 OBS=5000;
DATA SELRECS (KEEP = LASTTHREE PC REFCODE):
  RETAIN LT1-LT100 ARRAYSIZ 100;
  ARRAY SELCODTB(POINT) LT1-LT100;
IF N = 1 THEN LINK ARRLOAD;
INFILE MASTFILE (SUGICOMP);
INPUT @7 LASTTHREE 3.
  @10 PC 2.
  @14 REFCODE $2.;
DO POINT = 1 TO ARRAYSIZ;
  IF LASTTHREE = SELCODTB THEN
    DO;
      OUTPUT;
      GOTO EXIT;
    END;
  END; /* IF */
END; /* DO */
EXIT: RETURN;
ARRLOAD: INFILE SRCHLIST (PULLRECS);
  DO POINT = 1 TO ARRAYSIZ;
    INPUT @1 SELCODTB 3.;
  END;
RETURN;
PROC PRINT
  DATA = SELRECS:

EXAMPLE 4. COMPOUND IF SEARCH

OPTIONS OBS=5000;
DATA SELRECS:
INFILE MASTFILE (SUGICOMP);
INPUT @7 LT 3.
  @10 PC 2.
  @14 REFCODE $2.;
IF LT = 070 LT = 189 LT = 194
  LT = 263 LT = 352 LT = 594
  LT = 601 LT = 670 LT = 800
  LT = 941
THEN OUTPUT;
PROC PRINT;

Of course, in this case the program itself would have to be altered if new target values were to be used in different runs.

All four of these examples produce the same output. They differ in ease of maintenance and generality, and in execution speed. Execution times with files of 500 and 5000 records, using 10 and 100 targets, are shown in Table 1.

The compound IF technique tends to be fastest, especially with small files or limited numbers of targets. The overhead involved in building the arrays and indexing them is compensated by the increased element access time only when the file and target array begin to get large. When it is useful to store the targets in an OS data set, the extra effort required to use binary search or hashing instead of linear search pays off when the number of targets is large.
Table 1. Execution Times for Four Record Selection Techniques *

<table>
<thead>
<tr>
<th>Master File Size (Records)</th>
<th>500</th>
<th>5000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Unique Target Values</td>
<td>10</td>
<td>100</td>
</tr>
<tr>
<td>Search Technique</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Binary Array Search</td>
<td>1.92</td>
<td>2.48</td>
</tr>
<tr>
<td>(Example 1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hash Array Look-Up</td>
<td>2.48</td>
<td>12.01</td>
</tr>
<tr>
<td>(Example 2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Linear Array Search</td>
<td>1.98</td>
<td>4.73</td>
</tr>
<tr>
<td>(Example 3)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Compound If</td>
<td>1.19</td>
<td>2.26</td>
</tr>
<tr>
<td>(Example 4)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* Table entries are C.P.U. seconds on an IBM 370/158, 8 MEG.