ARGUS

AUTOMATIC ROUTINE GENERATING AND UPDATING SYSTEM

SORT AND COLLATE MANUAL

HONEYWELL

HONEYWELL 800

Transistorized Data Processing System

Litho in U.S.A.

DSI-43
11260

ARGUS

AUTOMATIC ROUTINE GENERATING AND UPDATING SYSTEM

SORT AND COLLATE MANUAL

Honeywell

H HONEYWELL *Electronic Data Processing*

TABLE OF CONTENTS

TABLE OF CONTENTS (cont)

TABLE OF CONTENTS (cont)

LIST OF ILLUSTRATIONS

FOREWORD

The sort and collate routines described in this manual are currently undergoing checkout on the Honeywell 800 System. The checked-out routines will be available in February, 1961.

The user of this manual will be able to obtain a general, as well as a detailed, description of the functions, operations and programming logic of these routines. Because these routines are still undergoing checkout, they may be subject to minor programming modifications.

For the reader who is interested in only a general description of these routines, he will find this information in the first part of each section. A more detailed discussion follows the general description.

SECTION I
INTRODUCTION

A series of sort generators for creating sort programs have been designed to be included in the ARGUS (Automatic Routine Generating and Updating System) Library of Routines. Assembly and use of the ARGUS sort routines have been simplified in keeping with two major considerations inherent in their design:

1. Efficient operation;
2. Universal application.

The ARGUS sort routines represent a departure from conventional sorting methods through the use of new programming techniques. The logic of these sorting routines provides for increased sorting speed while reducing the number of required tape drives and corresponding hardware.

Programming effort in developing a usable sort program for the accurate sorting of random data has been minimized to specifying, by means of an instruction, a desired routine from the ARGUS Library of Routines.

In order to generate a sort program, the programmer simply writes an instruction which specifies the desired routine from the ARGUS library and which supplies the parameters required by the routine. These parameters are interpreted and incorporated into a program tailored to perform a specific sorting function.

Although the ARGUS sorts are capable of handling a wide variety of situations, there are limitations imposed in the interest of providing the most efficient routines for the most common cases. Because of these limitations, there will be occasion to modify the routines produced by the ARGUS sorts, or even to hand-write a sort completely. The former is made easy, in many cases, by provisions for "own-coding" built into the ARGUS sorts (Section VI). These provisions allow, at specified points throughout the routine, detours into special coding added to the sort to perform some additional function.

The purpose of this manual is to provide a description of the ARGUS sorts and supply a working knowledge of them as an aid to own-coding or sort modification. Knowledge of these sorts may also be used as a guide in writing one-of-a-kind sorts for the Honeywell 800.

The sorts consist of two segments: presort and merge sort. The presort reads random data from one tape, orders it as much as memory space permits, and writes the data in ordered strings onto two or more output tapes. The merge sort then reads these outputs from the presort, and merges sets of strings, again writing them onto two or more output tapes. As this process continues, the strings become longer and fewer in number, until finally all of the data has been combined into one long string, written on a final output tape. In the case of large files which fill more than one tape, a collate routine is used to combine several ordered tapes (or files) into one continuous file.

At this point, a brief history of the ARGUS sorts might be of interest. The Honeywell 800 can read, write, and compute simultaneously. This type of computer is ideally suited for the presort method based on the building of continuous strings of items in memory as long as there are items which qualify. This method, as opposed to a strictly internal, fixed-length-string type of presort, takes advantage of any pre-ordering of data by producing correspondingly longer strings. Several other features of the Honeywell 800 are used to improve the presort. For example, index register addressing permits use of a single "tree" for several purposes (explained in Section III), and the large memory permits much data to be stored internally, resulting in long presort strings. The additional memory and the power of instructions used in conjuction with special registers result in a sophisticated generator, which makes very efficient use of data storage space.

The merge sort portion of the ARGUS sorts represents a complete departure from conventional merge sorting methods. The origin of this method can be traced back to 1956, when Honeywell mathematicians developed a unique sorting method which requires only three tapes. When study for the Honeywell 800 sorts was begun, it was felt that it would be desirable to include the three-tape sort along with conventional two- and three-way merge sorts, in order to accommodate small systems. As work progressed on the three-tape sort, methods were discovered which made it workable for any number of tapes. For a given number of tapes, it was discovered, this "n minus one" concept is always more efficient than conventional merge sorting. For greater flexibility, and in order to utilize a single set of merge sort generators, it was decided to use this new approach for all ARGUS merge sorts. Considerable revision has been made to the original three-tape concept, especially in the area of distribution of strings by the presort, and in the handling of an uneven number of strings.

General Sorting Function and Application

The basic function of any sorting operation is the ordering of data in a prescribed logical sequence to make that data accessible for later use. To control large amounts of data, a master file of all items of related information pertinent to a particular activity is usually established. These items are ordered according to a designated portion of each item. This portion, or field, is termed the key of the item.

To maintain a master file reflecting the latest transactions, it is necessary to update the master file periodically. This is accomplished most efficiently by ordering the transactions in the same sequence as the master file, each transaction containing the same key as the corresponding master file item. The power of the ARGUS sort generators can then be brought to bear to create the program which accomplishes this ordering.

Multi-precision Sorting

ARGUS sort routines are capable of sorting on a key of one field, two fields, or three fields; each field may be a full Honeywell word or any portion thereof. These operations are termed, respectively, single-precision sorting, double-precision sorting, and triple-precision sorting. Own-coding (as explained in Section VI), is used to handle longer sort keys.

Cascade Technique

To implement the merge sort on the Honeywell 800 System, Honeywell has developed a technique which is radically different from conventional methods employed with other data processing systems. This Cascade technique makes optimum use of the flexibility inherent in the hardware design of the Honeywell 800 to provide faster sorting with fewer magnetic tape units. In fact, Cascade sorting can be performed on as few as three tapes. The speed and power of the sort can be enhanced by adding any odd or even number of tapes. Conventional methods, on the other hand, require a minimum of four tapes and, in all cases, an even number.

The ARGUS presort differs most from the conventional presort in its distribution of strings on the two or more output tapes. Consider the case of a three-tape sort, in which data from one input tape is distributed to two output tapes. Instead of alternately writing strings on the two output tapes, as is normally done, this new method writes more strings on one tape than on the other. The ratio of the number of strings on the two tapes, which can be determined by a simple counting system, is 1.618 to 1.

The merge sort reads the two presort output tapes backward, merging successive strings and writing them on a third tape. When the strings on the shorter input tape are finally exhausted, there are still a number of strings remaining on the longer tape. These are copied onto the tape just emptied, in order to have all data on both remaining tapes in the same ascending (or descending) order. Thus, there are again two tapes with strings in the approximate ratio of 1.618 to 1 and another merging pass is begun. This process continues until one string remains.

As indicated, the same theory of sorting may be applied to any number of tape drives (n), in which case the presort writes strings in a prescribed ratio onto n-1 tapes, and the merge sort begins with an n-1 way merge onto the single remaining tape. Thus, the term n-1 sorting has been used in reference to the Cascade technique. As n becomes greater than 3, the Cascade method provides a proportionately greater sorting speed than the conventional merge (two way for four tapes, three way for six tapes, etc.) used with the same number of tapes.

Collate

The ARGUS collate is used to combine large files of sorted information extending over one or more tapes into one continuous file. While the reading and writing controls and tape handling are somewhat more complex, the aim and construction of the collate program are quite similar to those of the merge sort.

SECTION II
SPECIFYING AND USING THE SORTS

Basic Routines

The ARGUS sort and collate routines each consist of a skeleton routine and a modifier-generator. The skeleton routine provides the basic coding, or the programming framework, on which the modifier-generator can build a working program. The programmer's pseudo instruction and information from the file identification record of the input tape together supply the parameters needed to adapt this framework to the requirements of a specific sort or collate. The modifier-generator consists of two portions which together interpret these parameters to create the desired routine. The generator portion establishes storage and buffer areas in unused portions of memory (generating concept), while the modifier portion performs the actual adaptation of the skeleton routine (modifying concept).

Sort and Collate Routines within the ARGUS Library

When a programmer wants to execute a sort or collate operation, he inserts an instruction calling for that operation into his ARGUS program. Because this instruction is not directly executed by the logic built into the computer, it is called a pseudo instruction. Such a pseudo instruction calls for the desired subroutine (in this case a sort or collate) from the ARGUS library tape and specifies the parameters required for the execution of that subroutine. Parameters for a sort subroutine specify the precision of the desired sort, the number of input tapes, the size of available high-speed memory, the number of work tapes, etc.

Sort Routine Specifications

INPUT: The ARGUS sort routines are designed to handle one tape reel of unordered data or the equivalent data on several partially filled reels. Data in excess of this amount can be handled, provided that there is no overflow from any work tape at the end of the presort or of any intermediate pass of the merge sort. In other words, the input to the presort and the final sorted output may exceed one reel in length. However, if the capacity of any work tape is exceeded during the sort, the program will stop with a comment at the console and will have to be rerun with less data. The exact amount of data that can be handled by any sort is a function of the structure of the data and the amount of pre-ordering that exists, and therefore cannot be stated for the general case. Data which exceeds the capacity of the sort can be handled by multiple executions of the sort pseudo instruction. Each such execution generates and performs a separate sort routine, creating a file out of a portion of the data. The resulting files can be then merged by performing a collate.

The input data, which is read in the forward direction, must be preceded and followed by standard file (or segment) identification records, as described below. The sort routines can handle either fixed-length or variable-length items; however, the number of items per record must be fixed. Variable-length items are handled as such throughout the sort and must be followed by end-of-item symbols. Such symbols are also required with fixed-length items exceeding 63 words in length. The record size (in items), the maximum item size (in words), and the fixed or variable nature of the items are specified as parameters in the file identification record, as are the location and masking (if any) of the key(s).

OUTPUT: The output of the sort is a single file in ascending sequence, preceded and followed by file (or segment) identification records derived from the input data. The end identification record contains a segment number of hexadecimal G's, as required by the collate routine. The console typewriter produces a listing of all error and restart information, together with a count of the number of items sorted.

Collate Specifications

INPUT: The ARGUS collate routines are designated to handle up to 99 files of input. Each file may be contained on one or more reels of tape, and must conform to the following specifications. Each tape (whether a complete file or part of a file) must have standard identification records preceding and following the data to be collated. These identification records may be either file or segment identifications. The identification record preceding the data must be a beginning file (or segment) identification, and the one following the data must be an end identification. The beginning identification record must always be the record immediately following the tape identification block.

Word 1 of the identification record is a standard banner word, identical to those recognized by the sorts. Word 2 contains the name of the file; this must be the same for all identification records of a file. The low-order four digit positions of word 3 contain the segment number of the tape within the file, and have significance when a file is contained on more than one tape. These numbers in the beginning and end identification records of each successive tape of a file must be a sequence of decimal numbers. The first tape of a file may have any segment number, but the following tapes of that file must have numbers following it in monotonically increasing order. The segment number of the end identification record of the last tape of a file must be hexadecimal G's, as this marks the end of a file for the collate. This also holds true for files contained on a single tape, in which case the first tape is also the last.

The collate routines can handle the same range of variables as can the sort routines (except for the banner option), and these are specified in an identical manner in the beginning file

(or segment) identification record of one of the input tapes. The collate will use the tape first mounted on the "A" input as its source of parameters; the parameters specified in any other beginning identification records are bypassed.

OUTPUT: The output of the collate is a single file in ascending sequence, each tape of which conforms to the above specifications for input tapes. The beginning identification record of the first tape and the end identification record of the last tape will be file identification records; all others will be segment identification records. The file name of all identification records will be the name supplied in the pseudo instruction, and the segment number of the first tape will be 0001. Each tape (except the last) will be filled to capacity, unless a maximum number of records per tape is specified in the pseudo instruction.

INTERMEDIATE: If a collate consists of more than a single pass, which it will if the number of files to be collated exceeds the way of the collate, then the output of each pass except the last is an intermediate file which will act as input to a later pass. These intermediate files are identical in format to the final output except that tapes are always filled to capacity, and the file name in word 2 of the identification records will be an identification number arbitrarily assigned by the routine. These identification numbers specify each file uniquely, and coincide with the numbers printed in the plan. Further information on these numbers can be found in Section V, Collate.

Tape Positioning

Sort Input: Positioned so that the first record read forward by the sort will be the beginning-of-file identification. If both input drives or the "save" option are used, all input tapes will be rewound after the sort. Otherwise, the input tape will be positioned following the end-of-file identification record after the sort. There must be at least one record following the end identification record.

Sort Output: Initial positioning of the output tape has the same specifications as that of the work tapes (see below). After the sort, positioning takes place following the end-of-file identification record of the sorted file, and before an end-of-file information record.

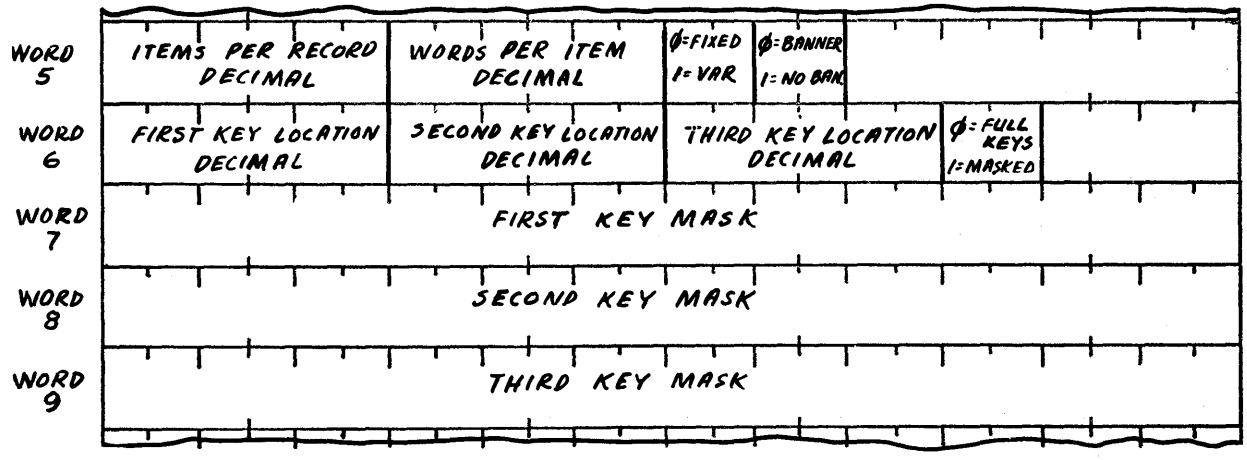
Sort Work Tapes: If at the beginning of tape, the sort will bypass the identification block and begin writing with the record immediately following. Positioning after the sort (except output tape) will be immediately after the tape identification block. It is necessary that there be at least one record written following the

tape identification block; this will be destroyed by the sort. Tapes positioned other than at the beginning of tape will be written starting at their current location. Positioning after the sort (except the output tape) will be at this current location. It is necessary that there be at least one record written beyond the current location; this will be destroyed by the sort.

Collate: Because of the tape changing necessary throughout the collate, all beginning identification records (input and output) are assumed to be the record immediately following the tape identification label. All tapes are rewound after use. The collate will bypass the tape identification block and begin reading or writing with the record immediately following. It is necessary that there be at least one record written following the tape identification block on the output tapes; this will be destroyed by the collate. Input tapes must have at least one record following the end identification record.

Identification Record

Every file to be sorted or collated must contain a standard beginning identification record as its first record. This record must not be greater in length than any of the data records in the file, and will have the following format in words 5-9:



Words 5 and 6 are to be specified as decimal constants. The number of items per record, the maximum number of words per item, and the three key positions relative to the first word of the item, each use three digits. The three options use one digit apiece and are as follows: 0 if the item length is fixed, and 1 if variable length; 0 if there are banner words at the beginning of each record, or 1 if there are no banner words except in the file identification records (must be 0 for the collate); 0 if the keys are not masked, and 1 if the keys are masked. The masks in words 7-9 are to be specified by the programmer in hexadecimal.

The sort assumes that the input tape will be positioned so that the first read forward order will read the beginning-of-file identification record. The address of the proper input tape is obtained from the macrocoding, and the identification record is then read into memory in the lowest available register. It is addressed by using index register X6. The sort verifies the first record as being an identification record by checking for the beginning-of-file banner word.

Banner words have a special significance to sort routines. The first word of all identification records is assumed to be a banner word of standard format, and will be used by the sorts to sense for beginning- and end-of-file records. Banner words are also used by the sorts to identify beginning-of-string records during the operation of the merge sort, and to give the number of each record written on tape relative to the first file identification record (low order 16 bits). If the input file to be sorted does not contain normal banner words on data records, the presort will add banner words to each record, and the final pass of the merge will eliminate them. This addition of banner words by the presort will occur if that portion of the parameter (one digit of the B address field) used for the banner option is 1.

Specifying and Using the Sorts

A single-, double-, or triple-precision sort routine is requested from the ARGUS Library of Routines through the use of a pseudo instruction. The format of the pseudo instruction is shown below.

ARGUS CODING FORM

PROBLEM _____ PROGRAMMER _____ DATE _____ PAGE _____ OF _____

	1 LOCATION 10	11 COMMAND CODE 22	S/C	24 A ADDRESS 37	38 B ADDRESS 51	52 C ADDRESS 65	REMARKS 66 LINE NUMBER 73	74 80
1								
2		L,SORTp,S		EI/EO/MERNAME	MMMM/II/AI/W1	W2/W3/W4/W5/W6		
3								
4								
5								
6								
7								

L - designates library routine.

SORT - designates the subroutine within the library.

p - designates the precision.

1 - single precision.

2 - double precision.

3 - triple precision.

S - designates an option to allow saving the input tape after the presort if that drive is to be used by the merge sort for a work tape.

S - change tape after presort.

M - indicates a multi-reel input with only one input tape unit available. If the input tape unit is assigned as the n^{th} work tape, the input file is saved, as if S were specified.

0 - do not change tape after presort. If this drive is assigned as the n^{th} work tape, the portion of the tape beyond the input will be used as a work tape.

EI - designates an input edit option (presort own-coding).

00 - no input edit will take place.

01 - edit option number 1 will be used to modify the parameters found in the file identification record of the input file.

02 - edit option number 2 will be performed immediately after the generation of the presort.

03 - edit option number 3 will be performed to modify each item in the input buffer before transferring it to item storage.

04 - edit option number 4 will perform all three of the above edit options.

EO - designates an output edit option (merge sort own-coding).

00 - no output edit will take place.

02 - edit option number 2 will be performed immediately after generation of the merge sort.

03 - edit option number 3 will be used to modify each item only during the final merge pass, at the point where an item has been transferred to the output buffer.

04 - edit option number 4 consists of the combined use of output edit option number 2 and number 3.

MERNAME - designates the segment name to be assigned to the merge sort own-coding. If there is no output editing, this field is left blank. The name assigned to merge sort own-coding can be any seven characters, but must be different from the name assigned to the segment which contains the sort pseudo instruction.

MMMM - designates in decimal the number of words of memory available to the sort in the preceding bank(s) in excess of the basic requirement of one bank. The basic requirement includes only the coding for the sort itself, and excludes own-coding, macrocoding, Executive Routine, etc. If sufficient memory is available for the sort specified, this information will be printed immediately and the sort will stop.

II - designates input tape drive assignment. All tape drive assignments are in standard ARGUS format consisting of a two-character code from AA to HH, except GG.

AI - designates alternate input tape drive assignment; if only one input drive is available, II is repeated.

W1 - designates final output tape assignment.

W2-W5 - designates work tape assignments. A code of "GG" should be designated for all work tapes not being used.

W6 - nth work tape assignment. The input tape assignment must be duplicated as the final work tape assignment (W6) whenever it is to be used as a work tape. If the input tape is to be saved, the S option must be an S (or an M).

A single-, double-, or triple-precision collate routine, like a sort routine, is called for by means of a pseudo instruction. The format of the collate pseudo instruction is shown below.

ARGUS CODING FORM

PROBLEM _____ PROGRAMMER _____ DATE _____ PAGE _____ OF _____

	LOCATION 1–10	COMMAND CODE 11–22	S/C	A ADDRESS 24–37	B ADDRESS 38–51	C ADDRESS 52–65	REMARKS 66–80
1		L, COLLATEp		EO/OTPTNAME	MMMM/NOREC	NF	
2		TAC		A1/A2/B1/B2/C1	/C2/GG/GG		
3		TAC		D1/02/E1/E2/01	/02/DT/GG		
4							
5							
6							
7							

L - designates library routine.

COLLATE - designates the subroutine within the library.

p designates the precision.
1 - single precision.
2 - double precision.
3 - triple precision.

EO - designates an edit option.
00 - no edit will take place.
01 - edit option number 1 will occur to modify the parameters found in the file identification record of input file.
02 - edit option number 2 will be performed to allow for own-coding after generation of the collate.
03 - edit option number 3 will occur to modify each item only during the final pass, at the point where an item has been transferred to the output buffer.
04 - edit option number 4 will occur which will provide for all three edit options.

OTPTNAME - designates the file name to be placed in the second word of the file or segment identification records of the final output file produced by the collate routine.

MMMM - designates memory (in decimal) available to the collate in the preceding bank(s) in excess of the basic requirement of one bank. The basic requirement includes only the coding for the collate itself, and excludes own-coding, macrocoding, Executive Routine, etc.

NOREC - designates (in decimal) the maximum number of records to be written on each tape of the final output file. If this field is left blank, tapes of the final output file will be filled.

NF - designates (in decimal) the number of files to be collated. The collate can handle up to 99 original input files, and thus the range of NF is 02 to 99.

The two Tape Address Constants (TAC) following the pseudo instruction are used to specify the large number of drives which the collate may use. Although not part of the pseudo instruction, these must be written in the prescribed format immediately following the pseudo instruction, so that they will be located immediately after the macrocoding associated with the collate. When finished, the collate exits to whatever instruction follows the second TAC.

A1 - designates the first, or main, "A" input tape assignment. This will be the source of the beginning-of-file identification from which the collate will obtain its file format parameters.

A2 - designates the second, or alternate, "A" input tape assignment. If the "A" input is to be restricted to a single drive, then A1 should be repeated.

B1 - designates the first, or main, "B" input tape assignment.

B2 - designates the second, or alternate, "B" input tape assignment. If the "B" input is to be restricted to a single drive, then B1 should be repeated.

C1 - designates the first, or main, "C" input tape assignment. If a two-way collate is to be performed, this as well as C2 should be written as "GG".

C2 - designates the second, or alternate, "C" input tape assignment. If the "C" input is to be restricted to a single drive, then C1 should be repeated.

GG - unused. These fields should be written as such, however, in order to fill the TAC instruction.

D1 - designates the first, or main, "D" input tape assignment. If a two- or three-way collate is to be performed, this as well as D2 should be written as "GG".

D2 - designates the second, or alternate, "D" input tape assignment. If the "D" input is to be restricted to a single drive, then D1 should be repeated.

E1	designates the first, or main, "E" input tape assignment. If a two-, three-, or four-way collate is to be performed, this as well as E2 should be written as "GG".
E2	designates the second, or alternate, "E" input tape assignment. If the "E" input is to be restricted to a single drive, then E1 should be repeated.
O1	designates the first, or main, output tape assignment. This is always the first tape to be written by the collate.
O2	designates the second, or alternate, output tape assignment. If the output is to be restricted to a single drive, then O1 should be repeated.
DT	designates the restart dump tape assignment. The code "GG" specifies that no restart points are to be established during the collate.
GG	unused. This field should be written as such, however, in order to fill the TAC instruction.

The parameters dealing with the file format are set up in the file identification record exactly as they are for the sort. Although all of the inputs to the collate must have standard beginning-of-file identification records, only the first "A" input file is used as the source of file parameters. In addition to the requirements mentioned in connection with the sort, the collate requires that the segment number (digits 11 and 12 of word 3 of the file identification record) of each reel of a file be in decimal sequence with respect to the reel preceding it. The segment number of the first reel of a file may be any number the user wishes. The end-of-file identification record of the final reel of a file must have a segment number of "GG" to indicate end-of-file to the collate. Because reels are normally mounted and dismounted during the course of the collate, the beginning-of-file (or segment) identification record of each reel is assumed to be the second record on tape, immediately following the tape label record.

Additional Memory Requirements

In both the sort and the collate macro instructions, there is a field (MMMM) which specifies the amount of additional memory available to the sort, in addition to the basic requirement of one bank. This figure is the same for both the presort and merge sort, but the requirements for each are somewhat different.

The nature of the presort is such that it can always function within a single bank (MMMM = 0000), regardless of item or record size. In some cases, namely when item size and record size are both relatively small, nothing is to be gained by providing additional memory space to the presort, since it is able to fit its maximum number of storage locations (216) within the basic bank. However, when items or records are larger, less than the maximum number of

storage locations will fit within the basic bank, resulting in shorter strings from the presort, and thus more merge sort passes and a longer sort routine. In such a case, providing the presort with additional memory would benefit the user by producing more storage, longer strings, fewer passes, and thus a shorter sort.

The merge sort is less flexible in this respect. Under some conditions, a merge sort will fit entirely within the basic bank; in other cases, it will have to have additional memory. The two variables which influence this requirement the most are the maximum "way" of the merge, determined by the number of tapes used by the sort, and the record size; together, these determine the size and the number of buffers used by the merge sort. Also influencing the amount of memory required is the span between the highest and the lowest key locations within the item, since this amount of space must be reserved for stoppering purposes. In single-precision sorting, the span is, of course, zero. The following formula gives the amount of additional memory required by the merge sort:

$$m = p + 3tr + s - 2,048$$

where

- m - designates the additional memory locations required (if the result is negative, less than the basic bank is required, and MMMM may be zero).
- p - designates the number of memory locations used by the program: 750 for single precision, 800 for double precision, 825 for triple precision.
- t - designates the number of tapes used by the merge sort: three, four, five or six (W1 through W6 of the macro instruction).
- r - designates the maximum number of words per record, including banner word, data, two ortho words, and end-of-record word, up to 254.
- s - designates key span, or highest key location minus lowest key location.

For example, in a double-precision sort with key fields in words 12 and 2 and a maximum record size of 100 words to be performed as a six-tape sort, p would be 800, t would be 6, r would be 100, and s would be 10; also 562 additional memory locations would be required. However, a four-tape sort of the same specifications (t would be 4) would be contained within the basic bank.

A similar formula may be stated for the collate as follows:

$$m = p + (3w \div 2)r + s - 2,048$$

where

- m - designates the additional memory locations required (if the result is negative, less than the basic bank is required, and MMMM may be zero).

p - designates the number of memory locations used by the program: 1, 250 for single precision, 1, 300 for double precision, 1, 350 for triple precision.

w - designates the maximum "way" of the collate: two, three, four, or five.

r - designates the maximum number of words per record, including banner word, data, two ortho words, and end-of-record word, up to 254.

s - designates key span, or highest key location minus lowest key location.

Calling for, Assembling, and Executing the Sort

There will be occasions when a particular sort may be one of several programs which are run as a series; at other times, it may be one of several segments making up a program. In either case, from a programming standpoint, a sort is a logical entity, and the ARGUS System goes to considerable effort to maintain it as such. Thus, a programmer has only to write the single pseudo instruction, L,SORT , in order to produce a complex pair of routines which together form the sort.

Although consisting of two separate phases, presort and merge sort, the sort is called for and loaded as a single subroutine. This is entered at execution time by a special calling sequence, or macrocoding routine. Thus, the L, SORT is a macro (pseudo) instruction which is replaced during Assembly by a 15-word calling sequence.

Being a subroutine, the sort is loaded along with whatever coding (including the 15-word calling sequence) the programmer has written. After the macrocoding is reached, it turns control over to the presort generator portion of the sort subroutine. This, in turn, generates and modifies the presort coding, using parameters set up by Assembly in the calling sequence, in order to provide a working presort; in doing so, the merge sort portion of the sort routine is destroyed. The presort is then performed, and when it is finished, it returns control to the calling sequence. The calling sequence now reloads the entire segment, namely programmer coding, calling sequence, and sort subroutine. This time the calling sequence specifies to the loader that control is to remain within the calling sequence rather than starting the segment at the beginning again. Once reloaded, the calling sequence tests the MERNAME field (from the original pseudo instruction), and if it is equal to alpha blanks, the sequence assumes no own-coding has been written for the merge sort. The merge sort generator portion of the subroutine is then entered, and this generates and modifies the merge sort coding, this time destroying the presort. The merge sort is performed, and when it is finished, it returns control to the location just beyond the calling sequence. Thus, whatever the programmer had written following

the pseudo instruction would now be performed in so much as the sort has been completed.

Any own-coding that is performed during the presort (see Section VI) is written as part of the same segment, together with the rest of the programmer's coding. As part of the same segment, it is loaded both before and after the presort, along with the programmer coding, calling sequence, and sort subroutine. Merge sort own-coding, however, is written as a separate segment. When the presort has been executed, and the basic sort segment has been reloaded, and if the calling sequence discovers that something other than alpha blanks was written in the MERNAME field, it then loads the segment by that name. The segment representing the merge sort own-coding must therefore be given the same name as in the MERNAME field and this will now be loaded. This approach allows merge sort own-coding to overlay the presort own-coding in order to conserve space, if so desired. Also, special register Z, S2 is used as the address for the own-coding to which the sort is to branch. This may be loaded as the address of the presort own-coding by the main sort segment, and then loaded as the address of merge sort own-coding by the auxiliary segment. More details concerning the handling of own-coding are contained in Section VI, Own-Coding.

The sort, being a subroutine, is handled as such by Assembly, and is therefore placed immediately following the programmer's coding. However, since the sort uses almost an entire bank, it can only be relocated modulo 2, 048. Within its bank, the sort program starts at location 20 (0020) and ends four locations before the end of the bank. The former unused area of memory permits the programmer to SETLOC the calling sequence plus the normal exiting macrocoding (usually either L, EXIT or L, READSEG) at location zero of a bank to enable the program to fit entirely within that bank. The four locations at the high end of the bank allow the sort routine to occupy the highest bank of a given installation without interfering with the stopper.

If the highest location used by the programmer's coding within a bank is greater than 20 (which it inevitably will be unless the programmer specifies a SETLOC of zero and includes nothing in his program or segment except the L, SORTp and the exiting pseudo instruction), the sort will be located in the succeeding bank of memory. By judicious use of SETLOC's, the programmer may determine exactly the space relationship between his coding and the sort routine. It is possible to separate the programmer coding from the sort by any number of banks by specifying, within the programmer's coding, a SETLOC whose sole function is to establish a memory location in some higher bank. Thus, a programmer might SETLOC his coding in bank 0 and follow the coding by a SETLOC of location 0000, bank 3. The coding (consisting of the programmer's instructions, the calling sequence, any own-coding, etc.) would be assigned to

bank 0, and, insomuch as a point has been established within the first 20 locations of bank 3, the sort routine would thereby occupy bank 3. The same result would occur if the second SETLOC had specified a location higher than 0020 in bank 2.

Macrocoding

The macrocoding corresponding to the L, SORTp pseudo instructions consists of the following:

ARGUS CODING FORM

PROBLEM _____ PROGRAMMER _____ DATE _____ PAGE _____ OF _____

	LOCATION 1–10	COMMAND CODE 11–22	S/C	A ADDRESS 24–37	B ADDRESS 38–51	C ADDRESS 52–65	REMARKS 66–80
1		MACRODEF					
2							
3		L,SORTp,S		EI/EO/MERNAME	MMMM/II/AI/WI	W2/W3/W4/W5/W6	
4							
5	MACRO	TS		C,+12	Z,AUI	N,AUI	
6		M,D,EO		D,EI	A,S		
7		FXBIN		(-MMMM)			
8		TAC		II,AI,WI,W2,W3,	W4,W5,W6		
9		OCT		26ØØ4ØØØ16ØØ77	77		
10		SEGNAME		(NAME OF SORT SEGMENT)			
11		SPEC		-	-	C,+4	
12		OCT		26ØØ4ØØØ16ØØ77	77		
13		ALF		MERNAME			
14		SPEC		-	-	C,+2	
15		NA		C,-2	C,+4	C,-3	
16		TS		C,+2	Z,AUI	N,AUI	
17		SUBCALL		-	-	SORTp	
18		SUBCALL		-	-	SORTp+1	
19		ALF		(BLANKS)			
20		FINIS					

During ARGUS Assembly, any L, SORTp pseudo instruction appearing in a line of coding is replaced by these 15 istructions starting with MACRO. The Assembly Routine matches all of the fields in the sample L, SORTp pseudo call of the above MACRO definition with the fields in the various constants below it. Thus MACRO+1, a mixed constant, will contain whatever the programmer wrote in the EO, EI, and S fields of the pseudo instruction. MACRO+2 will contain the quantity written in the MMMM field. MACRO+3, a tape-address constant, will contain all of the tape addresses from the pseudo instruction in compacted form. MACRO+5 is actually an alphabetic constant, set up by Assembly to contain the name of the segment containing the

L, SORTp pseudo instruction. MACRO+8 on the other hand, is set up as an alphabetic constant containing whatever was written in the MERNAME field of the pseudo instruction. MACRO+12 and MACRO+13 are set up by Assembly as SPEC constants, containing the address of the respective first and second locations occupied by the sort subroutine.

At execution time, the first order encountered (after any preliminary programmer coding) is MACRO. This transfers the contents of MACRO+12 to Z, AU1 and goes to N,AU1. MACRO+12 is a SPEC constant combining the first location used by the sort routine. This first location of the sort is, in turn, a TS instruction which stores the sequence history register, and transfers control to the entrance of the presort generator. The address which is stored from the sequence history register is MACRO+1, and this gives the presort generator access to the constants in the macrocoding.

When it is finished, the presort exits to MACRO+4 via the stored value of the sequence history register (just mentioned) which will have been incremented by 3 in the process of interpreting the parameter constants. MACRO+4, +5, +6, and also MACRO+7, +8, +9 are the macrocoding equivalents of the L, READSEG pseudo instructions. They are written here in expanded macrocoding form because macrocoding cannot call other macrocoding. MACRO+4 (as well as MACRO+7) are actually MPC instructions which must be written as constants in order that the reference to group 0, which they contain, will not be relocated in another group. These MPC orders specify that groups 1 through 7 will be turned off, group 0 be turned on, and the contents of the PCR be stored in the accumulator; the A address contains a code which specifies to the Executive Routine that this was a READSEG instruction. Executing one of these instructions causes the Executive Routine to load the segment specified in the following location, and to transfer control to the location specified in the SPEC constant following that instruction. Thus, when the presort exits to MACRO+4, the segment containing the L, SORT pseudo instruction is unloaded and control is turned over to MACRO+10.

Whatever was written in the MERNAME field (now in MACRO+8) is compared with alphabetic blanks (MACRO+14) in MACRO+10. If these are not equal, then merge sort own-coding exists, and that segment will have to be loaded. If MERNAME was left blank, however, the next move would be to MACRO+11, which transfers the address of the second location used by the sort into Z, AU1 and goes there. The second location of the sort program is similar to the first, except that it transfers control to the merge sort generator. The sequence history register is again saved, so that the merge sort, when finished, can use it to exit to the location beyond MACRO+14, thus returning to whatever programmer coding followed the L, SORT pseudo instruction.

If it had been determined in MACRO+10 that merge sort own-coding existed (MERNAME field not equal to blanks), the next move would have been to go to MACRO+7. This constant, like MACRO+4, is actually an MPC instruction which turns control over to the Executive Routine, telling it to load the segment whose name is in MACRO+8 and to go to the address specified in MACRO+9. This, in turn, leads to MACRO+11, an entrance to the merge sort generator, as stated in the preceeding paragraph, with the only difference being that the segment whose name was specified in the MERNAME field has been loaded.

In summary, the rather complex macrocoding associated with the L, SORTp pseudo instruction can be reviewed as accomplishing several purposes. First, it provides for translation of the parameters supplied in the fields of the pseudo instruction into a form usable by the sort generators (this function is performed by all subroutine macro calls). Secondly, it provides for reloading the sort's segment over again, thus allowing what is effectively a two-segment program (the sort) to be called for, and treated, as one. This enables the programmer to consider a sort routine as a single instruction. Thirdly, it allows the separate merge sort own-coding to overlay the presort own-coding after the presort is finished, so that memory space may be utilized as efficiently as possible.

Checking Sorts Using the Program Test System, PTS

The L, SORT macrocoding is, of course, designed for use with the ARGUS System; thus, ARGUS Assembly will set up the parameters, and properly locate the sort subroutine relative to the programmer's coding. The Executive Routine will relocate the entire segment, if necessary, and will handle the loading and overlaying at execution time as described previously. Program Test System, PTS, however, is designed to check out just single segments; hence, special care must be taken when checking out sort routines or when checking out programs of which sorts are a part. For each program tested, PTS requires a START card, which specifies the ending location of that program. The instruction in this location, however, will not be performed. Therefore, the first MPC to be encountered, MACRO+4, should be specified as the ending address in the START card of the sort segment (since PTS will not recognize the Executive Routine MPC). Insomuch as PTS will automatically load the next program on its tape, the same sort program should be specified a second time in order that the sort will be reloaded. For this second sort program, the sequence counter should be set to MACRO+10, which can be accomplished by means of a transfer instruction inserted in place of the first instruction of the program, which places MACRO+6 in the sequence counter.

If there is no merge sort own-coding, the START card for the second program of the pair can specify the final order of programmer coding as the exit location. However, if merge sort

own-coding is employed, then MACRO+7 should be specified as the exit of the second program, and a third program, which consists of the merge sort own-coding segment, will have to be included. In this case, only one instruction of the second program will be performed (MACRO+ 10), whereupon the third program will overlay the second. BACKGROUND should not be loaded between the second and third programs since the subroutine and programmer coding brought in with the second program must remain in memory. The entrance to the third program should be temporarily specified as MACRO+11, following the same manner used in specifying the entrance to the second program. The START card for the third program specifies the final instruction of programmer coding as the exit.

Thus, the method of checking out a sort with PTS is the same as that used with any multi-segment program. Each segment, which would normally be loaded automatically in a production run by the Executive Routine, is treated as an individual program. Because of the manner in which it is reloaded, the sort subroutine must be handled as two programs during PTS, both of which will have the same name, but each of which will have different entrances and exits. Merge sort own-coding similarly requires the inclusion of a third program to represent the additional own-coding segment.

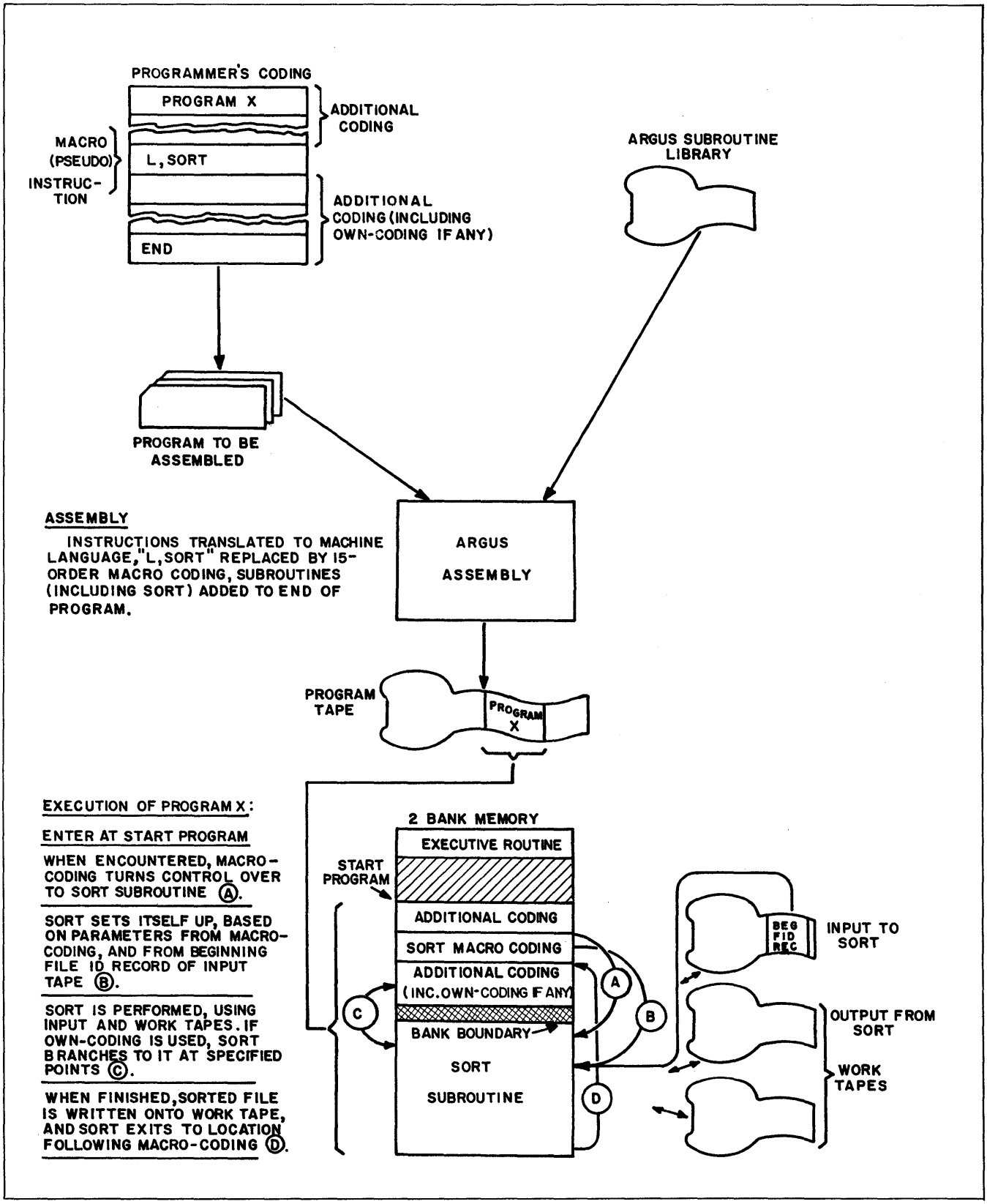


Figure 1. Derivation of a Sort Program through ARGUS

SECTION III
PRESORT

General Method

Figure 2 shows a simple version of the ARGUS presort. Storage in this simple case consists of only three items, the items each representing a complete record on tape, and the keys consist of a single digit. The tape on the left (input) contains 10 of these items, in random order.

Initially (a) the storage area is empty, and all tapes are positioned at the beginning (as indicated by the arrows). In the first step (b), the storage area is filled with the first three items from tape; the input tape is now positioned just after the third item. The item with the smallest key is then selected and written on one of the output tapes, and replaced with an item from the input tape. Thus in (c), the 0 is written out, and replaced with the 6. At this time it is ascertained that the key which was brought in (6) can be included in the present string because it is greater than the key just written out (0).

This process continues, and in (d) the 3 is written out and replaced with a 5, and in (e) this 5 is found smallest and written out, being replaced by a 4. Now, however, since the 4 is smaller than the last item written, namely the 5, it cannot logically be included in the present string, so it is stoppered (temporarily removed from consideration), as indicated by the cross (X) placed through it in the example.

The choice is now confined to the remaining items in storage and the 6, being the smaller, is written on the output tape and replaced with a 7, as indicated in (f). The 7 is next written on the output tape (g) and replaced by the 1, at which point this item must also be stoppered. There is no choice left but to pick the remaining item in storage, so the 9 is written on the output tape and replaced by the 8, which is also stoppered. At this point (h), all three of the items in storage are stoppered, and one string has been completed on the output tape.

Writing (i) is now switched to the other tape; all items that are presently stoppered in storage are unstoppered; and again the smallest item is selected. The 1 is written and replaced by the 2, which happens to be the last input item. When the end of the input is sensed, reading is discontinued and each item in storage is stoppered after it has been used (j). Thus, in the

remaining three figures, (j, k, l) each item is stoppered, after it is selected and written, and the presort is completed.

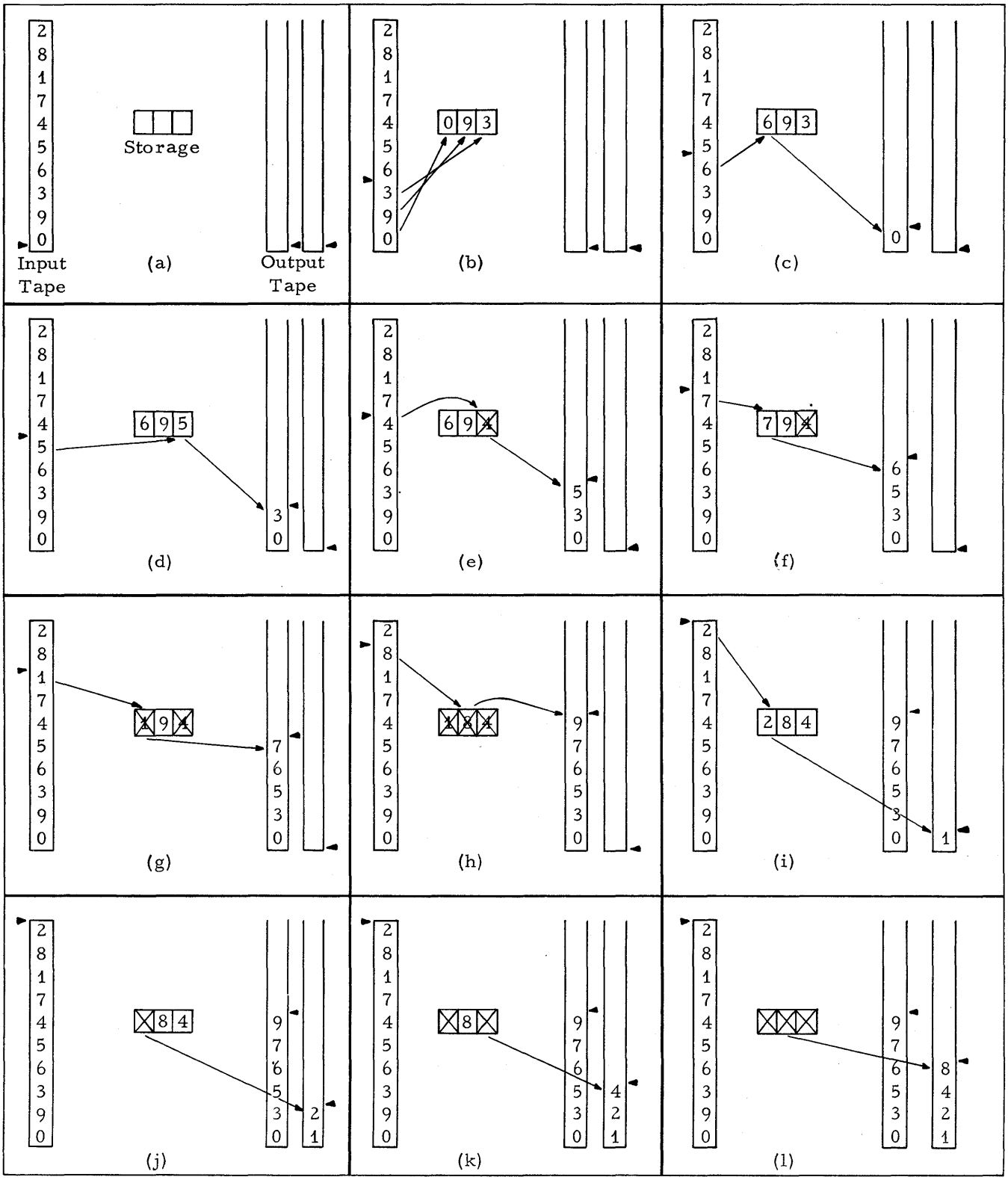


Figure 2. Simple Presort Example

Reading and Writing Controls

Reading and writing are both handled by the presort in a conventional manner, using two input and two output buffer areas to allow simultaneous read-compute-write operations. All four buffer areas are essentially the same size as the records on tape. As one input buffer is being loaded from tape and one output buffer is being written on tape, the presort processes data using the remaining input and output buffers. Since one tape is being read and one is being written at any one time, with the same amount of data coming in as going out, reading and writing cycles are synchronized, although they may operate independently if own-coding (see Section VI) is used to modify record or item size or to add or delete items.

Building Strings

Besides the reading and writing controls, the presort consists basically of the coding necessary to find the smallest usable key in storage. In a multi-precision sort, all specified keys are juxtaposed and the presort seeks the smallest value of the combined keys.

The simple example of the ARGUS presort method, included in this section, demonstrates how variable-length strings are produced by a cascade presort. The potential length of each string is directly influenced by two basic factors: the amount of memory that is made available to store items while comparing for the smallest key; and the randomness of the data itself. The more items which can be included in storage, the longer the generated strings will be. Moreover, any preordering or natural ordering of the data will directly bias the length of the generated strings. If the input data is completely random, meaning that each new item brought in has a 50-50 chance of having a key which is smaller than the key of the preceding item, the generated string length will average twice the number of items that are stored internally. Thus, in the simple example of the presort method, which uses a storage capacity of three items, the first string happens to contain six items.

The presort modifier-generator takes full advantage of any amount of memory made available to it, setting up as many item storage locations as possible. The skeleton routine can be adapted to work with any number of storage locations up to and including 216 items, thus being capable of producing strings of 432 items average length.

Trees

The process of finding the smallest key in storage is accomplished by means of a "tree" or a series of trees. A tree is a section of coding having a single entrance and several exits. Figure 3 illustrates a comparison tree having a single entrance and six exits. This tree

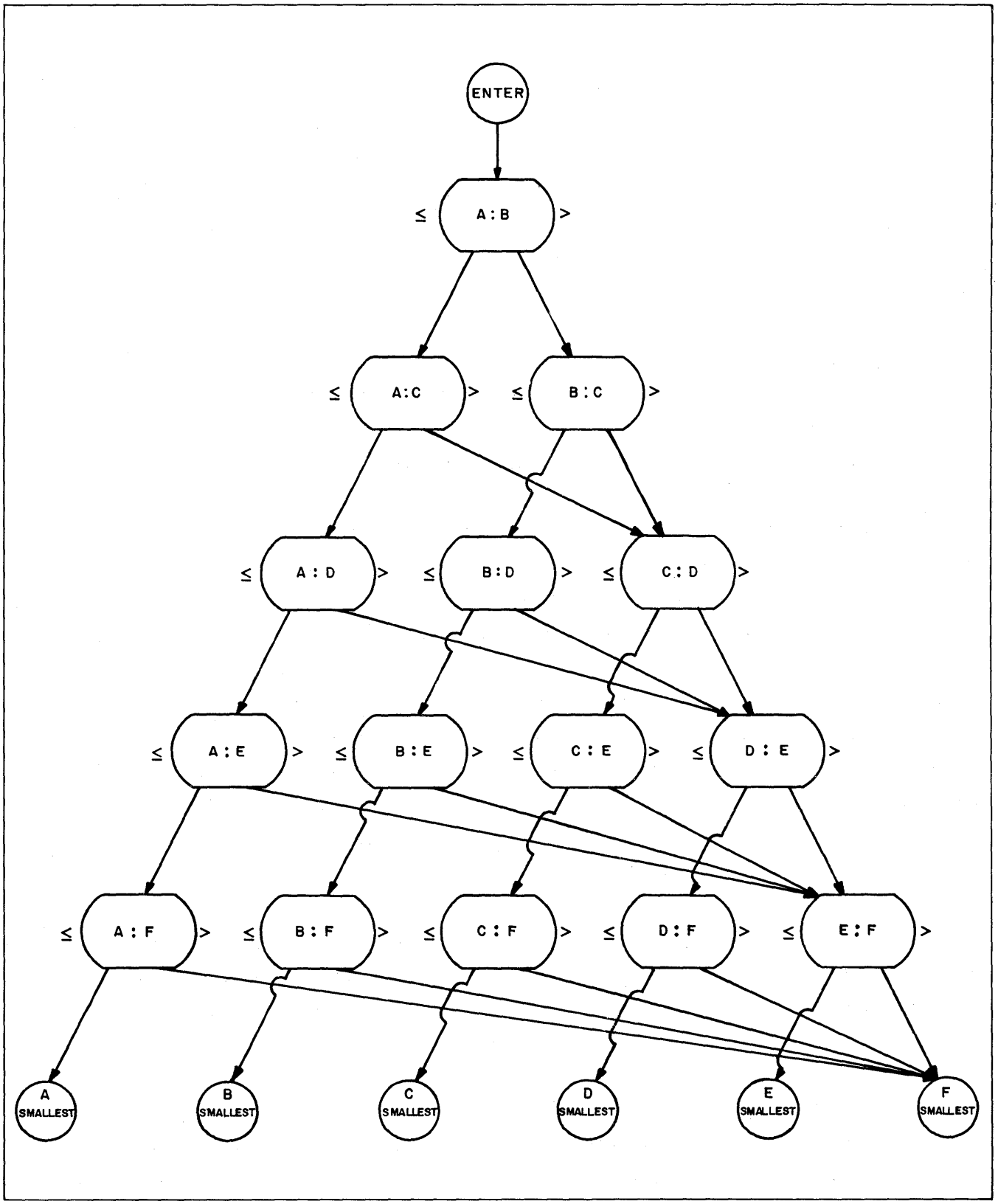


Figure 3. ARGUS Presort Tree

examines six keys (A, B, C, D, E, and F) and determines which is the smallest, which in turn determines which exit from the tree is to be used.

Bins and Tags (General Description)

Associated with each tree in the ARGUS presort are several storage areas called bins which contain the keys being compared. As comparisons are completed at one level of trees, it is necessary to transfer the key that was found to be smallest in each tree at that level to the bin associated with the next level. Since moving the entire item would in most cases be unwieldy, both in terms of transfer time and of space required, tags are used to minimize the amount of data that has to be transferred.

As each item is brought into memory, the related key is detached and sent to an area within a bin which corresponds to that item's storage location. Within a bin, every other location is set up with an identification word which designates the start of an item in storage. Each key is placed in the bin adjacent to the ID word which designates the start of the corresponding item. Thus, each item is represented in the bin by a two-word group, called a tag, which contains the key of the item and its starting location. It is this tag which is actually moved as control proceeds from layer to layer. When the smallest key is determined, the ID portion of the tag is interpreted to find the location of the corresponding item, and that item is transferred to the output buffer.

Layers of Trees in an ARGUS Presort

In general, trees are most efficient when they have from four to six exits. Therefore, the trees used in the ARGUS presort never have more than six exits. When more then six keys are to be compared, the concept of layers of trees is used. For example, a full ARGUS presort of 216 items uses three such layers of six-way trees, as shown in Figure 4.

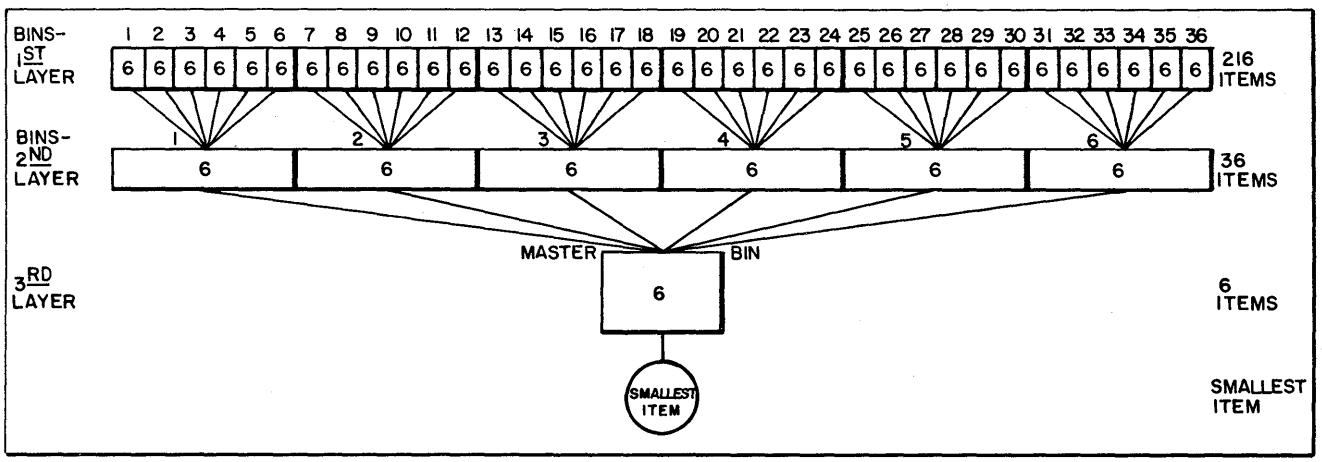


Figure 4. Bins Accompanying Each Layer of Trees

In this example, the first layer consists of 36 bins, each containing six tags. As a result of all the comparisons at this level, the smallest key in each of the 36 bins is determined and the corresponding tags are delivered to the six bins at the second layer. In the same fashion, the second-level comparisons result in the transfer of six tags to the final bin, or master bin, which comprises the third layer. The tag transferred from the master bin contains the smallest of the original 216 keys, as well as the starting location of the corresponding item. Various programming techniques used to minimize the number of passes through the trees are explained later in this section, as trees are covered in more detail.

Stoppering

In the presort example, stoppering is represented by drawing a cross through an item in storage. Stoppering is performed when a new key is found to be too small for the current string. A constant of hexadecimal G's is transferred to the corresponding key location in the bin. This constant will never be found smaller than any other key in the bin. When all of the key locations in a bin contain hex G constants, the program unstoppers all key locations and starts a new string. This is accomplished by transferring the keys from all stoppered items to their respective bins and then repeating all trees to find the new smallest key. The related item is then written as the start of a new string.

Old Key Area

That portion of the presort coding which determines whether an item should be stoppered or whether it qualifies to be included in the present string is termed the "old key area". The key of the last item written in the present string is retained in this area and compared with the key of the next item coming into the storage area. If the new key is smaller than the old key, the new item does not qualify for inclusion in the present string and, therefore, a stopper is provided.

Master Routine (General Description)

The master routine is used in conjunction with the old key area and the presort tree and represents the basic routine of the presort. As long as one or more items remain unstoppered, it transfers out the item having the smallest key, brings in a new one, and gives control to the old key area. The latter, in turn, gives control to the tree after determining whether the new item must be stoppered. The master routine checks input and output buffers and branches to read or write routines whenever necessary. When all of the items in memory become stoppered, the master routine gives control to an end-of-string routine to close the current string and start the next.

End of String

Each time that a string is completed, there are several tasks to be performed. Because banner words (one per record) are used by the merge sort to identify the beginnings of strings, it follows that breaks between strings must correspond with breaks between records. Thus, if the output buffer is only partially full when a string is completed, filler items, known to have keys larger than any item of the file, must be generated in order to complete the record. Items consisting entirely of hex G's are generated by the end-of-string routine for this purpose.

NOTE: The following text refers to many special registers used in the ARGUS presort program. Appendix B of this manual provides a list and functional description of these special registers.

Bins and Tags (Detailed Description)

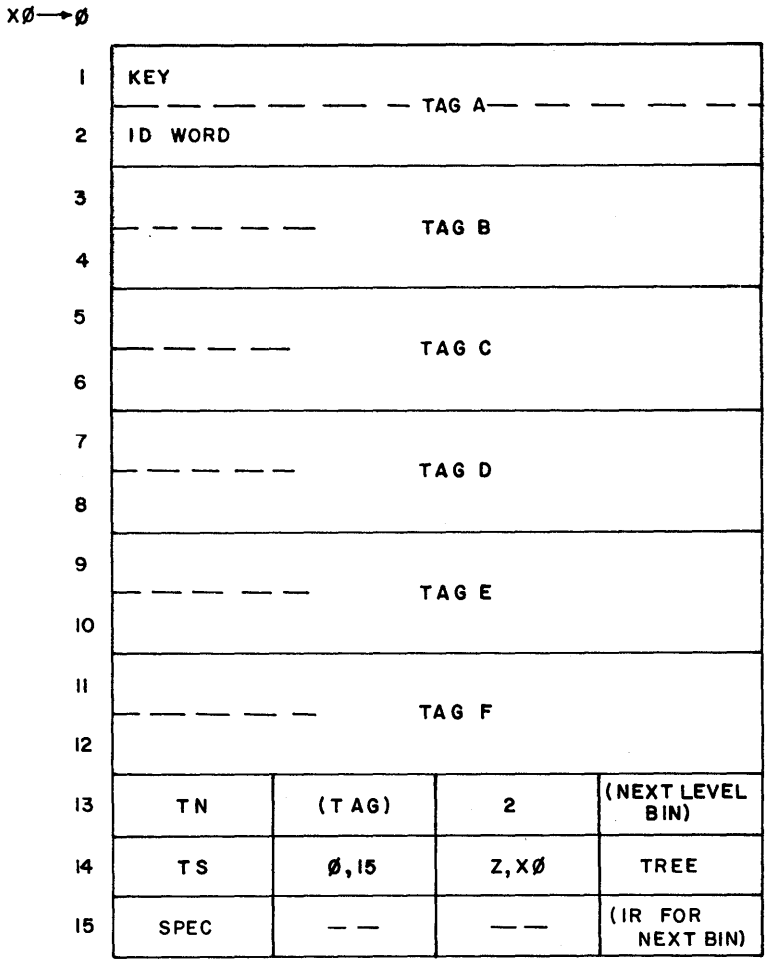
As previously explained, six items are compared with the tree in the ARGUS presort, thus accounting for the fact that six tags are grouped together in the accompanying bin. In addition to these six tags, the bin contains several instructions and pieces of information which will be used when the tree has determined which of the tags in the bin is the smallest. This is necessary because, in order to preserve space, the ARGUS presorts use only one tree, which is associated with the appropriate bin via an index register. After the smallest item in any bin has been found, the additional orders of the bin transfer that smallest tag to the next appropriate bin, and after having set the index register to that bin, control is transferred once again to the six-way tree.

Figure 5 includes a detailed arrangement of a typical tag bin. This represents the layout of a single-precision bin. Double-precision bins are similar except that an additional word for each tag is used as a second key. The triple-precision bins are similar to the single, the tree being supplemented by additional coding to compare the second and third keys within the items themselves.

When each of the bins is in use, the index register is set to the location just prior to word 1 (X0 in the illustration) so that the increment to the index register will correspond with the word number. Starting at the top of the bin, the six tags occupy pairs of words, the first word being the key, and the second being the ID word. The ID word is illustrated at the bottom of Figure 5. Following the six tags, which occupy words 1 through 12 in the bin, are three additional words (13, 14, and 15) which are additional orders and a constant.

Word 13, called the bin TN instruction, transfers the smallest tag to the next bin. This is constant except for the A address which is set up by the tree each time the tag representing

SINGLE-PRECISION BIN FORMAT



ID WORD FORMAT

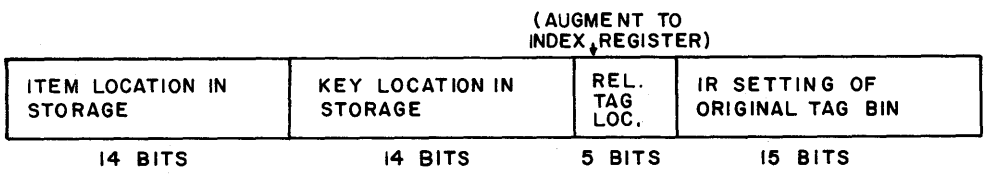


Figure 5. Tag Bins and ID Word Format

the smallest item is found. The tree stores the (indexed) address of the tag here. The B address indicates the number of words to be transferred (two in this case). The C address is the location of the tag position in the next bin which corresponds to this tag bin. (It has been explained previously that six initial bins feed into one bin of the next level; also the smallest tag from each of these six bins is placed in a corresponding position of the next level bin.) After setting the A address of word 13, the tree exits to word 13, all through indexed addressing.

After word 13 is performed, word 14, which represents the bin TS instruction, transfers control to the next level bin. This is performed by storing the contents of word 15, a constant, into the index register, and going to the tree, now associated via the index register with the next bin, and the process continues. As indicated in the diagram, the A address is indexed, and the B address is a direct special register address. Word 15 is a SPEC constant, representing word 0 of the next level tag bin.

Master Bin and ID Word

To avoid going from tag bin to tag bin indefinitely, the final tag bin is slightly different from the others. This final bin is termed the master bin, and as previously explained, is the bin from which the smallest tag of all stored items is found and transferred out. The master bin is actually the same as the other bins through word 13, except that the C address of word 13 refers to a working area called the old key area. Thus, the same tree can be used in association with the master bin as the others, and the exit will also be the same.

The ID word is used in the section of coding following word 13 of the master bin. At this point, the ID word serves to identify the item in storage from which the attached key came and the tag bin to which that key was originally transferred. The ID words are generated originally as constants and are placed in the appropriate positions of the initial tag bins to which the keys from the items are transferred. In the intermediate bins and master bin, these ID words have been brought along with the keys from previous bins (see Figure 5).

Single-Precision Tree

The description of the bins, and an explanation of their operation, form the external specifications of the tree. A tree, then, must compare indexed words 1, 3, 5, etc. (key words of the tags) to find the smallest, substitute the indexed address of the smallest tag into the A address of indexed word 13, and go to indexed word 13 of the bin. At each exit of the tree, there is a masked TS order which is used to substitute the address of the smallest tag into the A address of indexed 13. In order to economize constants, it picks up the address from one of

the instructions in the tree (in the one case when such an address does not appear in the A address position of the tree, a constant is used instead as the source). This TS instruction then sequence changes to indexed 13. Figure 6(a) illustrates a single-precision tree and how it differs from a (b) double-precision tree and (c) a triple-precision tree.

Double-Precision Tree

When a key comprises more than one word, the logical comparison of such a key will consist of more than the one LA instruction necessary with single-precision keys. Only if the first two words compared (high-order portion of key) are equal, is it necessary to compare the next word of the key. This requires that an NA instruction be performed to determine if the first are completely equal. Figure 6 illustrates this relationship; (a) illustrates a single instruction needed for a single-precision comparison between A and B; (b) illustrates double-precision keys, A1 and B1 being the high-order portions (first keys), and A2 and B2 being the low-order portions (second keys). The entire array in (b) corresponds to each of the comparisons shown in the earlier tree. It becomes apparent that each additional word in a key adds considerably to the number of instructions required as well as the time needed to go through the trees. The final comparison is simply an LA instruction, so that if two keys are completely equal, one is arbitrarily picked as being "smallest". A superfluous TS instruction is saved by reversing the final LA instruction (that is, B:A instead of A:B). In (b), if both LA instructions were A:B, one will have to be followed by a sequence change, since both obviously cannot remain in sequence, when A is greater than B, and still end up at the same place. By reversing the second LA instruction, a sequence of memory locations, as indicated by the address numbers over each comparison, can be assigned. When several such comparison arrays are grouped together in a tree, the sequence of instructions follows down the tree rather than through the levels of comparison. Figure 6(b) illustrates this in the double-precision tree.

Triple-Precision Tree

It would be possible to extend the reasoning used in the double-precision tree to handle a triple-precision tree (accompanied by a corresponding increase in tree size), but in order to conserve memory as well as facilitate the addition of higher precision, the triple-precision presort makes use of a space-saving technique. The basic triple-precision tree itself is identical to the single-precision tree insomuch as each comparison set consists of one LA instruction and one NA instruction; however, if the first keys are found equal, a special section of coding is entered. This coding represents a common second and third key comparison array used for all comparisons after the first key. With own-coding, this section can be easily modified to work with any number of additional keys. The tag bins are actually simplified, being identical to

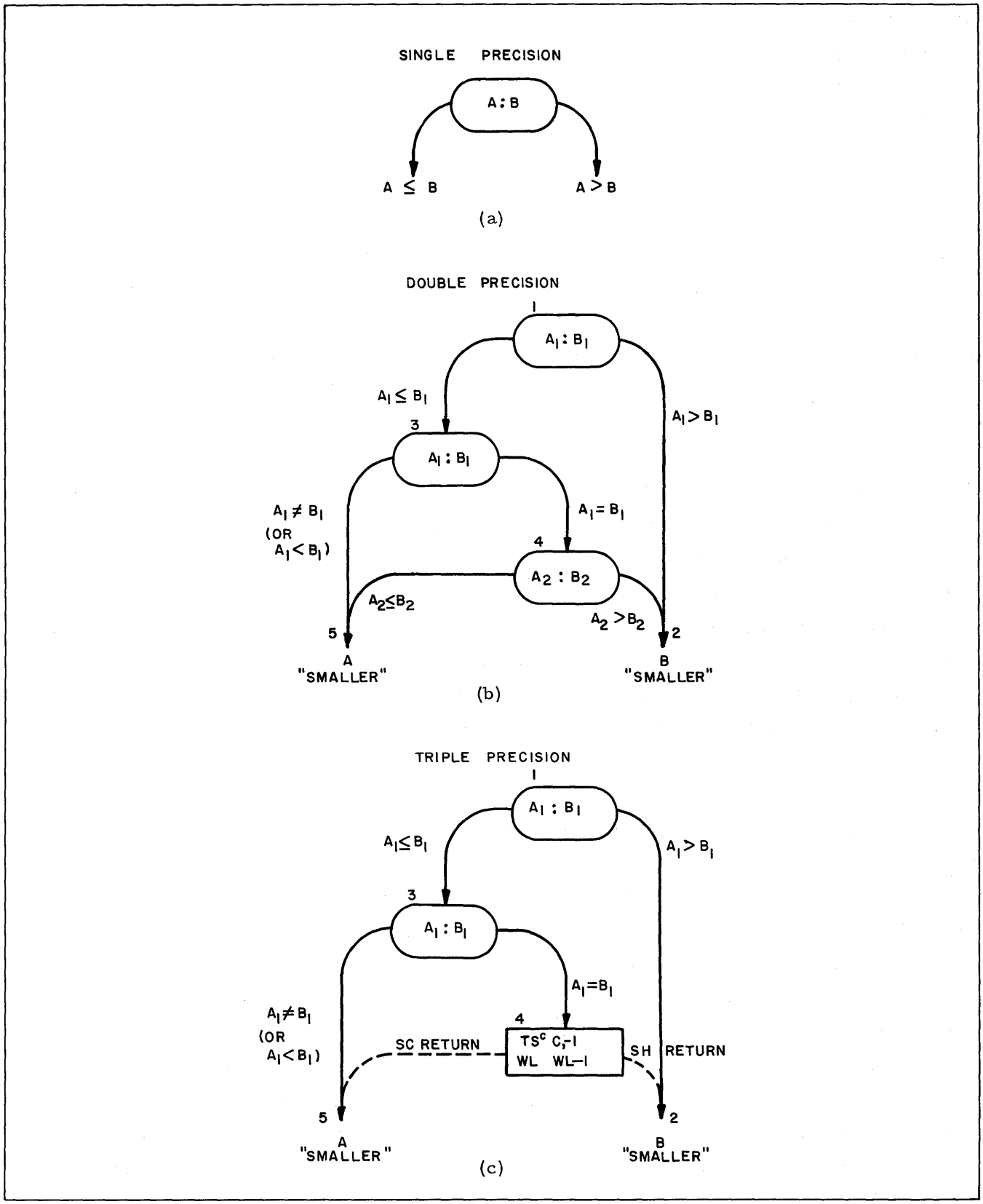
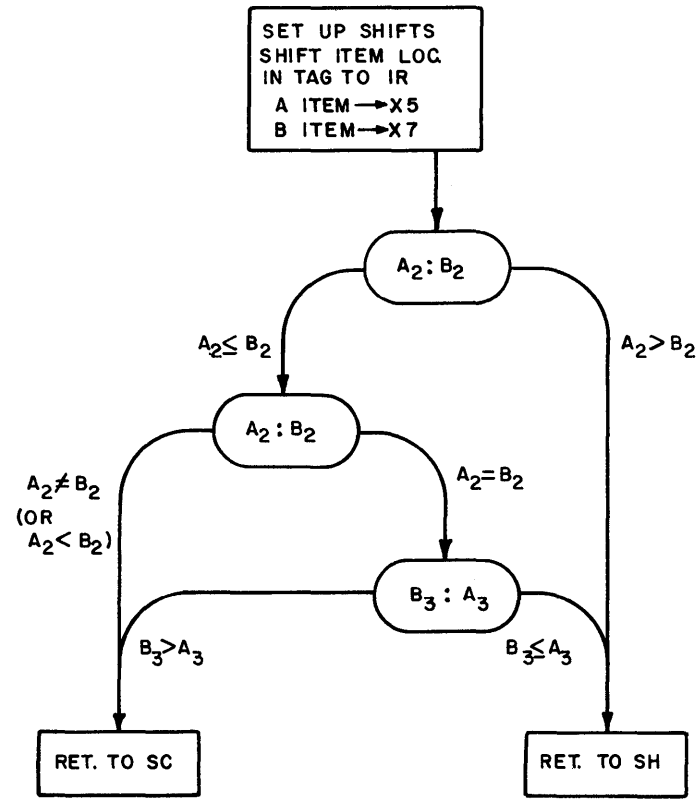


Figure 6. Single, Double, and Triple Precision

single-precision bins, since only the first key word is carried in the tag. The second and third keys are compared directly within the item itself. If the first key words in a triple-precision sort are fairly random, which is the normal case with a presort, the special coding will seldom be used. When the additional levels of precision are used, this technique is slower than the straight-forward tree.

The triple-precision tree is illustrated in Figure 6(c), as well as in Figure 7, which shows the special common coding area. The latter illustrates a tree that has been arranged so that when the first keys are found equal, the sequence history register is set to the next instruction which would be executed if one of the keys were smaller, while the sequence counter is set to the next instruction to be executed if that key were greater. In place of the second LA instruction in each comparison group, the triple-precision tree has a TS instruction to switch to cosequence. This transfers the NA instruction just performed to a working location in the common coding before going to that area. Figure 7 illustrates the common coding. Two index registers (5 and 7) are set to the values of the two corresponding items in storage by using the ID words of the tags being compared. Next, second and third key comparison instructions (still in cosequence mode) compare the additional keys directly in item storage through these two additional index registers. The third key LA instruction can easily be changed to go to own-coding, upon finding possible equality, and any number of additional keys can be compared there in the same way. Two exits are possible: in one, the contents of the sequence history register are stored in the sequence counter and the program drops out of the cosequence mode so that return is made to the sequence counter; in the other, the program drops out of the cosequence mode without modifying the sequence counter.

Suppose that two keys, A and B, are equal in the first and second words, but A3 is smaller than B3. (Figure 7 illustrates this example.) The first order in the tree is A1:B1, and since A1 is less than or equal, go to 3. In order to go to 3, however, the sequence must be changed, and so the address of 2 is stored in the sequence history register. In 3, A1:B1 is tested for equality and since they are equal, 4 is performed next. This (4) is a TS instruction specifying a switch to cosequence, which transfers 3 to WL, and goes to WL+1. In WL+1, a WA instruction, masked to include the A address only, is performed to add the contents of WL to a 1 bit in the A position and then stores the result in WL+2. This, in effect, sets the A address of WL+2 to the ID word of the A tag. WL+2 shifts the ID word so that the item location portion is justified right and places this into index register 5. WL+3, 4, and 5 similarly put the location of the B item into index register 7, the only difference being that the B address of WL must first be shifted to the A address position.



CODING

WL	(NA		Ø,1	Ø,3	#5)	STORED NA
	WA,AMASK	C	WL	ONEINA	WL+2	
	SWS,14 BITS	C	(Ø,2)	34	Z,X5	SHIFT A ID WORD
	SWS,AMASK	C	WL	36	WL+5	
	WA,AMASK	C	WL+5	ONEINA	WL+5	
	SWS,14 BITS	C	(Ø,4)	34	Z,X7	SHIFT B ID WORD
	LA	C	5,(2)	7,(2)	WL+8	AUGMENTS GENERATED
	TX		Z,SH	—	Z,SC	RETURN TO SH
	NA	C	5,(2)	7,(2)	WL+1Ø	AUGMENTS GENERATED
	LA	C	7,(3)	5,(3)	WL+7	AUGMENTS GENERATED
	PR					RETURN TO SC

Figure 7. Triple-Precision Common Comparison Coding

Now WL+6, the second key LA comparison, is performed. Because the second keys are equal, the equality test in WL+8 is performed next. This second key comparison almost yields equality; therefore, WL+9 is performed next. Here, because A3 is less than B3, WL+7 is performed next to transfer the contents of the sequence history register to the sequence counter, and returns control to the sequence counter. Alternatively, if A3 were greater than B3, the program would have proceeded to WL+10 and then reverted directly to the sequence counter. To extend precision with own-coding, the C address of WL+9 would have to be replaced with the address of an NA order in own-coding and the process would continue from WL+9. Also the contents of the A and B addresses of WL+9 would be interchanged to keep the logic straightforward.

Master Routine (MASTER) (Detailed Description)

The explanation thus far has been related to the trees and bins and just how they operate together to determine the smallest item in memory. After it is determined that an item has the smallest key, that key is transferred to the storage area termed Old Key (OLDKEY). The item is then processed through the Master Routine (MASTER).

The first operation in the MASTER is a test to see if the key of the smallest item is all hex G's (or a stopper). A stopper indicates that there are no more valid items in memory, and control is turned to the End of String Routine (ENDSTR). Next, the ID word, which is situated in OLDKEY along with the smallest key, is used to find the item's location in memory. It is stored in X7 before X7 is used to transfer the item to the output buffer. The output buffer is standard, using an index register (X3), which is modified to step through the buffer as items are transferred to it. A check is made to see if the buffer is full, and if so, control is transferred to the Write Routine (WRITE), which will return to the same point when finished. If own-coding option number 3 is specified, the routine now branches to modify an item in the input buffer before it is brought into the sort.

Now the next item is transferred to storage from the input buffer by a section called Item Transfer (ITEMTRAN). Here the input buffer is stepped through with a modified index register (X1), and the item is transferred to the location in storage just vacated (addressed by index register X7). If the input buffer is now empty, control is transferred to the Read Routine (READ) which will return to this point upon completion.

Using the ID word still in OLDKEY, the tree index register (X0) is set to the bin associated with the item just replaced. The old key coming from OLDKEY is then compared with the new key from item storage via X7. If the old key is less than or equal to the new key, the new item will be included in the present string. From the ID word, an order is set up to

transfer the key of the item to its bin. This order is performed, and a sequence change is made to the tree to sort this item with others. Setting the tree index register associates the tree with the bin to which the new key was just transferred. When the smallest item in this bin is found, transfer is then made to the next level bin, and finally to the master bin with a new smallest item.

If an old item had been greater than the new one, the new item would have been stoppered, as demonstrated in Figure 2. To do this the Dummy Key Routine (DUMKEY) sets up an order to transfer all hex G's to the bin, performs this order, and proceeds to the tree. Except for transferring G's instead of the real key, this procedure is the same as described in the preceding paragraph.

It should be noted that items having words of all hex G's as keys will appear to MASTER as stoppers, and will, therefore, indicate an end-of-string just as an ordinary stopper key would do. However, since an item with a key of hex G's is never replaced by any other item in memory storage (for it will never be transferred out), such items will tend to accumulate in the storage area and accordingly reduce its effective size. Furthermore, if as many hex G-key items are brought in as there are storage locations, the presort will go into an endless loop through the end-of-string procedure, reading and writing nothing. Because of this, keys of hex G's should be avoided, except possibly to fill up the final record of the data file.

End-of-String Routine (ENDSTR)

The first step in the End-of-String Routine (ENDSTR) is to check the output buffer. If it is empty, the string did end integrally with a record and control is transferred to Write Switch (WRSWCH) to determine where the next string is to be written. If the buffer is not empty, dummy items (hex G's) are transferred, one at a time, to the output buffer. The buffer is checked after each transfer. When it is full, transfer is made to the Write Routine (WRITE) after setting the exit of the write routine to return to WRSWCH. Just before entering WRSWCH, a dummy write forward (WF) instruction is performed for the tape presently being written to get an error check for a bad record before switching tapes.

Write Switch Routine (WRSWCH)

WRSWCH is used in conjunction with ENDSTR to determine on which tape each string is to be written. A table of ideal ratios of strings on each tape is calculated. The logic of the distribution of these strings onto the tapes for the merge sort is introduced and explained in Section IV. To maintain this proper distribution, the presort, at WRSWCH, calculates this table of

ideal ratios and provides a constant for each tape, indicating how many strings should be on that tape. Associated with each tape is a string counter which shows how many strings have actually been written. In general, strings are placed on one tape until the corresponding string counter equals the ideal count for that tape. The presort then switches to writing on the next tape. When all tapes are at their ideal number, the next higher ideal distribution is calculated, and the process to bring tapes up to it begins. WRSWCH, therefore, consists of a series of comparisons which test each tape to see if there are as many strings as there should be on it. If there should not be enough strings on a tape, the write order is set to address that tape and the string counter is incremented. When all tapes are full as specified, control is transferred to the calculating portion to determine the next higher perfect distribution.

The logic involved in the calculation of this ideal distribution is based on the cascade method of merge sorting used by the ARGUS sorts. This method is explained in Section IV.

Fill Bins Routine (FILBIN)

When it is determined onto which tape the next string will be written, transfer is then made to FILBIN from WRSWCH to start the new string. This routine performs two functions; first, it sets a switch (Banner Switch) so that the beginning banner word of the new string will indicate a beginning of string; secondly, it unstoppers all items. This is accomplished by transferring all of the keys directly from the items to the bins, using several special registers properly incremented. At this point, transfer is made to the Switch Routine (SWITCH).

Switch Routine (SWITCH)

The Switch Routine (SWITCH) performs the initial sorting of all items in storage. As already stated, only one bin in each level was sorted to process one new item. However, at the beginning of string, all items must be processed together, and as a result, all bins must be sorted. SWITCH places temporary "detours" in the TS instructions of all but the last bin. These detours change sequence back to another portion of SWITCH which increments the tree index register to associate with the next bin, and then returns to the Comparison Tree (CT). A switch is also placed in the first order of MASTER which changes sequence to an area termed the Reset Area (RESET). Now the tree index register is set to the first bin, and transfer is made to the tree. Because the bins are adjacent to one another, each bin will be sorted in turn down to the master bin. The master bin will be sorted, after which control is transferred to RESET. At this point, all bins have been sorted, and after the SWITCH modifications have been removed by RESET, control is transferred to MASTER. RESET accomplishes this by restoring the TS orders in the bins and the first order in MASTER, and then exiting to the first MASTER

instruction. Processing now continues as before until another end of string is sensed.

Read Routine (READ)

READ is entered whenever an input buffer is depleted, as determined in MASTER. For checking purposes, a counter is incremented in READ which keeps a tally of the number of records of input to the sort. READ also sets up X2 with the address of the empty buffer, and X1 with the address of the other. The latter process is the actual buffer switch, and is accomplished by shifting (end around) a special word containing both addresses. As a check, hex G's are placed in the end-of-record word position of the buffer into which reading will take place and into the word just beyond that. The record is then read while the record in the other buffer is checked. If this is an end-of-file record, transfer is made to an End-of-File Routine (EDOFILE) or to the Multiple Input Routine (MULTINPT) (if that was specified in the macrocoding). A check is made, when working with fixed-length records, to see if the record is too short (hex G's instead of an end-of-record word) or too long (other than hex G's in word beyond the end-of-record word). Otherwise, only the test for too long a record is made. If all of these tests are passed, the input buffer item counter (R7) is reset to unity, and return is made to the master routine. During the initial loading of storage only, this exit is set to return to the Fill Storage Routine (FILSTR).

Write Routine (WRITE)

WRITE is entered when the output buffer is full. WRITE first stores AU2 in X6 when working with variable-size items, since this will contain the address just beyond the last item. The same type of buffer switch as used in READ is used to switch the empty buffer to X3 and the full buffer to X4. The banner switch is checked; this switch, which is normally set to 1, is set to zero at the start of each string by FILBIN. If the switch is zero, " beginning-of-string" bits are substituted into the banner word, and the banner switch is set to 1; otherwise, "middle-of-string" bits are substituted into the banner word. The ortho count is then computed, and the current write order (as set up by WRSWCH) is performed. The record count is incremented in the banner word and transferred from the current buffer to the alternate one to maintain the current count from record to record. The exit from WRITE, called COMMONEX, normally leads to MASTER. At the end of string it leads to WRSWCH.

Beginning the Presort

The information contained within this section, up to this point, has been limited to the "steady-state" portion of the ARGUS presort. The initializing and beginning portion of the sort routine which precedes the steady-state portion is discussed in this section and is followed by a discussion of the ending portion of the presort.

Not to be confused with the beginning portion of the sort are the generator and modifier functions, which are discussed later in this section. The generator and modifier, although normally performed immediately before the sort, serve only to set up a specific sort routine. However, a sort routine actually starts manipulating data in its beginning portion.

The first such section is termed BEGIN, and it starts by initializing the input buffers. X2 is set up, the switch is rotated, and words of hex G's are placed in the last locations of the initial buffer, into which a read is then performed. Other special registers, used as counters and as addresses, are also set up. The storage area is filled with words of hex G's in case there should be insufficient items in the file to fill it. The initial banner word is set up, and control is transferred to the standard read routine.

READ is set up initially to exit to the beginning of the Fill Storage Routine (FILSTR). The first order in FILSTR switches the READ exit to go to FILSTR4 to avoid the initializing orders of FILSTR after the first time through. These orders set R2 to unity to count the number of items brought in. They also set X7 to the address of the first item in storage, and proceed to the Item Transfer Routine (ITEMTRAN). If specified, exit to own-coding option 3 is made before the item is brought in. This option is used to modify an item before it is brought from the buffer to item storage. For variable-sized items, the number of words in the item are determined, and that number is stored in the low-order portion of the end-of-item word for later use by the merge sort. The input buffer is checked, and if empty, control is turned over to READ. Otherwise, X1 is set to the next item in the buffer (using AU1) and a transfer is made to FILSTR4, to which READ also exits.

In FILSTR4, item storage capacity is checked against R2 which contains a count of the items brought in. R2 is incremented. If storage is not full, a return is made to ITEMTRAN. When storage is finally full, the last remaining switches for normal operation are set up. These include EDOFILE which has been set up especially for FILSTR, and the exits of ITEMTRAN and READ, which will now go directly to MASTER. Exit is then made to FILBIN where the first string is begun in the usual manner.

Ending the Presort

An end-of-file record, sensed by READ, initiates the "ending the presort" process. If a multi-input was specified, transfer is made to the Multi-option Area (MULTOP), where the addresses of read orders are switched to the alternate input tape. Control is then transferred to the Multi-input Area (MULTINPT). MULTINPT is printed out, the old input tape is rewound

with interlock, the new tape is positioned, and data from that tape is read into one input buffer. Return is then made to the portion of READ which performs a read order to insure that the first record has been brought in. READ now continues as usual. When all the input tapes have been used, both input drives should be left interlocked or empty, in which case the program will stall. At this point, the operator may start the presort at the cosequence counter and a normal ending will take place. Upon finding an end-of-file record, READ will lead to EDOFILE if the multi-input option is not specified. If the presort is still in the initial process of filling storage, an initial version of EDOFILE is performed; otherwise, the normal EDOFILE is performed. (An item design of the end-of-file record is contained in Appendix A.)

The initial EDOFILE section simply modifies ITEMTRAN to place into item storage an item of hex G's (dummy item) instead of a new item from the input buffer. Also, a switch is stored at the beginning of the WRSWCH which will lead to the End of Sort (ENDSORT). This is done because there already are some items in storage which must be put out as an initial string. After setting WRSWCH, EDOFILE exits to FILBIN to create the first (and only) string.

The normal EDOFILE is somewhat more complicated because any time during the steady-state portion of the presort there may be some items in storage which are stoppered; if the sort should be ended at the completion of the current string, these items would never be put out on tape. To avoid this, ENDSTR is set up with a switch to go to Check Items Routine (CHKIT) which will determine if there are any items besides dummy items in storage. WRSWCH is set up to allow one more string to be written and then control is transferred to ENDSORT. This is accomplished by replacing the first order of WRSWCH with a transfer which will replace itself and which will go to a constant equivalent to WRSWCH. The constant used to replace the transfer is a TS order to ENDSORT. CHKIT is a simple looping routine which uses incremented special registers, R4 and R5, to compare each key in storage with hex G's. If all the keys in storage are hex G's, a TS order to ENDSORT is placed in WRSWCH, and the initial order of ENDSTR is replaced and return is made to it. If any key in storage is not equal to hex G's, WRSWCH is ignored because it was set by EDOFILE.

When ENDSORT is reached, the first routine performed is called Check String (CHKSTR). This checks the distribution level counter, which is equivalent to the number of passes the merge sort will have to make. If this count is equal to 2 (the lowest possible value it can have), provisions must be made to insure that a minimum number of strings are written on all tapes. To accomplish this, CHKSTR sets MASTER to bypass the beginning of ENDSTR (which checks to see if the output buffer is partially full) and then goes directly to the section of ENDSTR where it is assumed that dummy items are needed. This guarantees at least one record of dummy items

for each string which is necessary on any tape, even though no real data is transferred out. The portion of WRSWCH which calculates the next distribution level is set to go to the main ENDSORT; this provides that dummy strings are put out until the current distribution level is full. Due to the nature of the merge sort, once two distribution levels have been reached, or beyond two merge sort passes, it is not necessary to provide as many strings as required for the particular distribution. Thus, if the pass counter were not 2, ENDSORT would be reached directly from CHKSTR.

Finally, at ENDSORT, all data records have been written. The total number of records from the read counter are now printed for control purposes. If the "save input" option was specified, "SAVEINPT" is printed, and the tape is rewound with interlock. (A newly mounted tape is read past the beginning record.) The string deficiencies on each tape are calculated by subtracting the actual string count from the ideal string count which was calculated for the current level; and the numbers of passes are obtained from the level count. This information is put together to be written as an end-of-file record on all the merge sort data tapes. Two beginning-of-file records are written on the "nth" work tape (the one not being used by the presort), and the statistical end-of-file records are written on all other tapes. End-of-information records are then written on these data tapes, and the tapes are positioned just before these end-of-information records. "TO MERGE" is printed, and control is transferred to the macro-coding.

Over-all Flow of the ARGUS Presort

Thus far, a simple presort has been discussed in general terms, and various components of this presort have been explained in detail. These components are tied together in the following paragraphs to present a complete presort picture. A presort flowchart is shown in Figure 8.

In BEGIN and FILSTR, special registers are set up and the storage area is filled with hex G's. The input buffer settings are initialized, and an initial record of data is read into memory. From then on READ functions normally. A loop is used to transfer items from the input buffer to storage until the storage area is filled. Now normal operation can be established and control is transferred to FILBIN.

FILBIN is the first working part of the sort. It contains the instructions used to transfer all keys to the tag bins, and is performed at the beginning of each string.

In the next section, SWITCH and TREE-BINS, each bin is inspected once to find the smallest item in storage, and all bins are sorted.

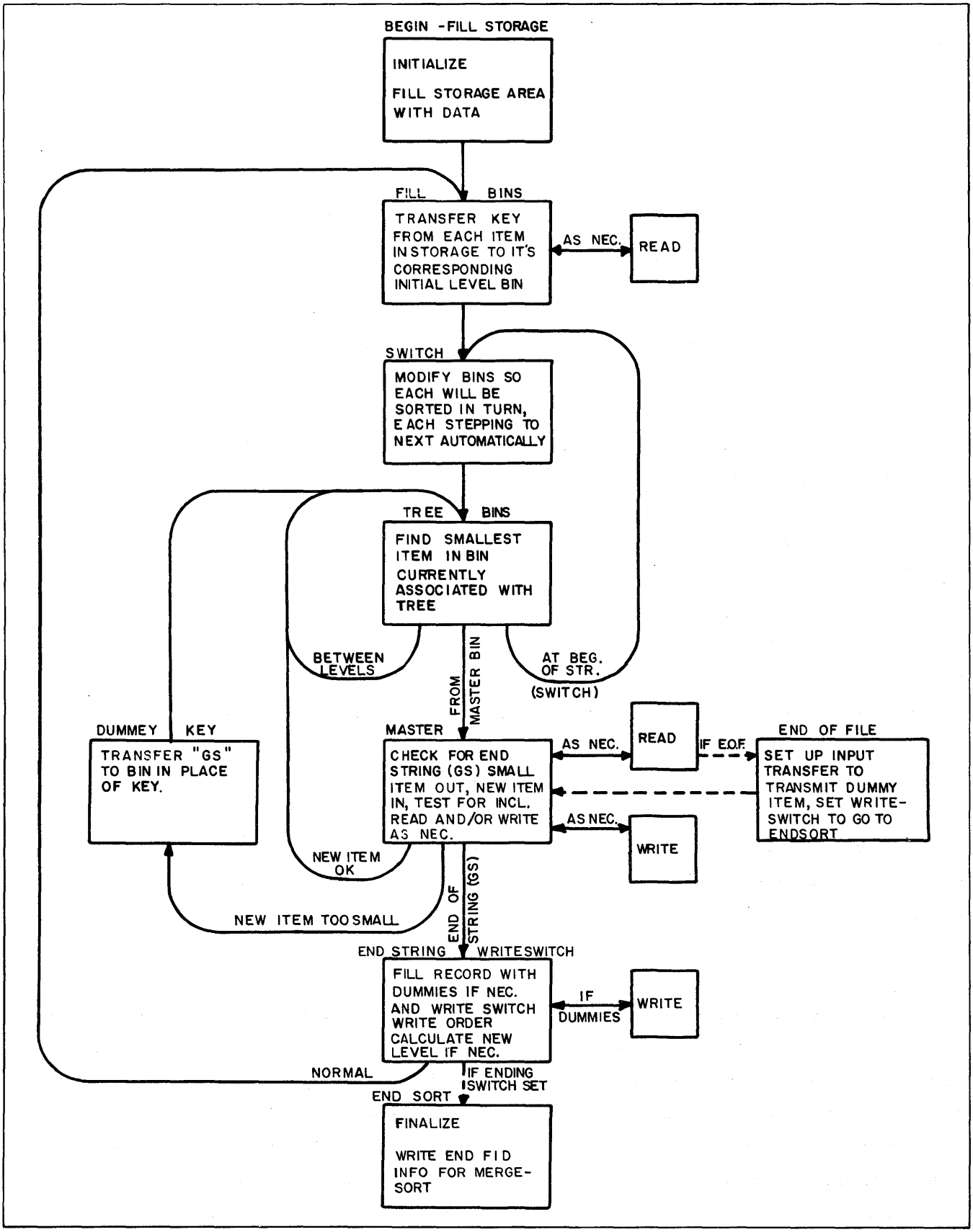


Figure 8. Over-all Flow Chart of the ARGUS Presort

The program then proceeds to MASTER where a number of functions are performed. A check is made for end of string (hex G's as smallest item). If an end-of-string condition is encountered, a transfer is made to ENDSTR. The sorted item is sent to the output buffer and the write counter is incremented. When the output buffer is full, control goes to WRITE to write the record, and switch and initialize buffers. The sorted item is then replaced with a new one from the input buffer, and the read counter is incremented. When an input buffer is empty, control is turned over to READ to read a new record, and switch and initialize buffers. If the check in READ finds an end-of-file record, control is turned over to EDOFILE. Back in MASTER, the key of the new item is tested against that of the old one to see if the new one can be included in the current string. If it can, its key is transferred to its bin; otherwise, DUMKEY puts a key of hex G's in the bin. In either case, a return is made (via the ID word) to sort that bin. This bin will, in turn, lead the way to sorting the bin into which it feeds, eventually leading to the master bin and once again to MASTER. This loop continues as long as a string is being built. Eventually, when MASTER detects all hex G's as the key of the smallest item, the program exits from the loop to ENDSTR.

In ENDSTR the last record of the current string is finished with dummy items, and the output tapes are switched (if necessary) at WRSWCH. Return is made to FILBIN to transfer all the new keys to the tag bins. Each bin is processed once again at SWITCH and return is made to the main loop in MASTER. When an end-of-file record is detected in READ, control is transferred to EDOFILE. At this point, the program will be modified slightly; rather than bringing in new items from the input buffer, dummy items are brought in to fill storage with permanent stopper items. WRSWCH is set with a switch which will allow the completion of the present string, plus one further string if there are any items remaining in storage.

Finally ENDSORT calculates the string deficiencies on each tape and writes this information as part of the end-of-file record on each tape. From here, exit is made back to macro-coding.

Modifier-Generator

Upon entering the modifier-generator coding, the parameters in the FID record, which are specified in decimal, are converted to binary, and the three options are checked. The maximum number of words per item specified in word 5 of the FID includes the end-of-item word, whenever it is used. An end-of-item word must be specified with all items greater than 63 words in size, as well as with variable-length items. The end-of-item symbol is a word (other than the end-of-record word) whose high-order 32 bits are BB00FFFF. Its low-order 16 bits are to be reserved for use in variable-length items to specify the number of words in that

particular item. Thus, if the fixed or variable option is a 1, the program is modified to handle variable-length items by determining the number of words in the item and retaining the count in the end-of-item symbol of each item. Unless the banner word option is a 0, the program is modified to add banner words to each data record. If the mask option is a 1, the necessary masks are set up in memory and the presort routine is modified to handle masked keys. Since the parameters and necessary statistical information will be transmitted via tape from the presort to the merge generator in the form of an end-of-file identification record, the data to be transmitted to the merge is stored in memory as each parameter is checked or converted, and as each tape address is obtained from the macrocoding.

All tape addresses are compared to determine the "way" merge, and the necessary read and write instructions in the presort routine are generated with the appropriate tape addresses. Also the routine for switching work tapes at the end of a string is modified according to the determined "way" merge.

The S option is checked for S, and for M. If S, ENDSORT is modified to save input, as explained earlier. If M, the read routine in the program is set up to allow for changing input tapes at the end of each input tape and the input tape assignment is checked for equality with the "nth" work tape. If II equals the nth work tape assignment, ENDSORT is set up to re-position the tape on the input tape drive in the case of multiple input. If not M, II and I'I' are checked for equality and if they are not equal, the read routine is set up for multi-tape input.

The item size is checked for 63 or less words per item. It there are less than 64 words per item, the program is modified to handle the items more efficiently. If unmodified, the program handles items up to a maximum of 250 words per item, and assumes that an end-of-item symbol is specified with each item.

To complete modification of the presort, the buffer addresses are set up in specified constants, index register X0 is set up with the address of the location just before the first tag bin, and a location tagged FITLC is set up with the address of the first item in item storage. The beginning-of-file identification record, which has remained untouched in memory, is then written twice on all but one work tape. It is written once as the given FID, and a second time as a full length data record for the particular sort. It is assumed that all the work tapes were positioned before operation to preserve any desired information on tape. Hence, the FID is used to mark the beginning of the file which is currently being sorted. For restart purposes, the generator and modified basic program are dumped onto the second work tape, and two more FID's are written on that tape.

The presort generator, which was loaded into the high-order registers of the specified bank along with the basic program, is entered upon completion of the modifier. In the generator, the identification tags for each item in storage are placed in the tag bins, and the necessary transfer orders between the different level bins are set up. During generation, complete advantage of item size and record size is taken to determine the most efficient use of the available memory for allocation between item and bin storage.

Error Correction and Restarts

The presort makes use of the orthotronic error-correction routines provided by the Executive Routine, thereby saving memory space which would otherwise be duplicated. In cases where, for some reason, the Executive Routine is not available, special sort error routines may be added by means of own-coding; however, they result in a corresponding decrease in the amount of memory available to the sort.

In the event of a read error indication, the address and size of the suspected record is determined by the sort, and control is turned over to the Executive Routine to repair the record. If the record cannot be repaired, an attempt is made to reread the information and if it is still erroneous, control is turned over to the Executive Routine once again. In any case, a comment is printed at the console typewriter to tell the operator what has happened.

If the physical end of any work tape is reached, a printout informs the operator and the tape is rewound with interlock, whereupon control returns to the restart point. The program will stall on this tape until it is exchanged, presumably for a longer one.

The restart (initiated by starting at R0) is included in the sort coding. After modification of the presort, but before its generation (the distinction being that the former is caused by parameters in the macrocoding and the latter by the parameters in the file ID), the contents of memory are "dumped" onto the second work tape of the sort. If a restart is initiated during the presort, all tapes will be positioned backward to their beginning ID records, and memory will be reloaded from the second work tape, whereupon the presort will automatically go through generation and start again. Normally the restart information on the second work tape will not be used, and will be ignored by the merge sort (a second set of file ID records having been written after the dump). At the completion of the sort, this tape will be repositioned to where it was before the sort.

SECTION IV
MERGE SORT

General Method

Figure 9 shows a simplified version of a three-tape ARGUS merge sort. For the sake of clarity, each string from the presort will be considered as a unit and designated by a letter, rather than showing individual items within each string. It will be assumed that the presort (a) wrote eight such strings, distributed as in (b). This example represents a simplified case of a three-tape merge sort, and an ideal distribution for the merge operation.

The merge sort (b) is ready to read tapes A and B backward, merging the last strings from each tape, and writing the result on tape C. Thus, string G is merged with string H to produce GH as in (c). It is important to understand that both G and H are composed of a number of ordered items and that, during the merging process, these are combined to produce a single ordered series of items, which is called string GH. Likewise, D and F are combined to form DF, and B and E to form BE. GH, DF, and BE are written, end to end, on tape C. This process stops when the end of information is sensed on tape B (the shorter tape).

Thus, (c), all the data is on two tapes, the information on the A tape in ascending order but, because in the merge sort data is read backward and written forward, the data on the C tape is in descending order. Therefore, to arrange all the data in the same order, tape A is copied, reading backward, onto tape B. At this point (d), one full merge pass has been completed over all the data. The number of strings now on the longer tape is equal to the number formerly on the shorter tape. The number of strings now on the shorter tape is equal to the number formerly on the longer tape minus the number formerly on the shorter tape.

In (e) and (f) another pass is completed, following which the data is again in ascending order, and the number of strings is reduced as before. (g) and (h) show that the next pass results in two descending strings, one on each tape. (i) shows the end product of the merge where all the data has been merged onto a single tape in ascending order.

Each merge pass, except the last, is composed of two phases, or subpasses; a two-way merge (in this simple example) and a copy pass. This type of merging can be extended to any number of tapes (up to a total of six in the ARGUS sorts).

In a six-tape merge, a five-way merge is first performed onto the sixth tape, then a four-way merge onto the fifth, a three-way onto the fourth, a two-way onto the third, and finally a copy (or "one-way" merge) onto the second tape, leaving the first tape empty for the initial phase of the next pass. Thus, each pass in a six-way merge consists of five subpasses.

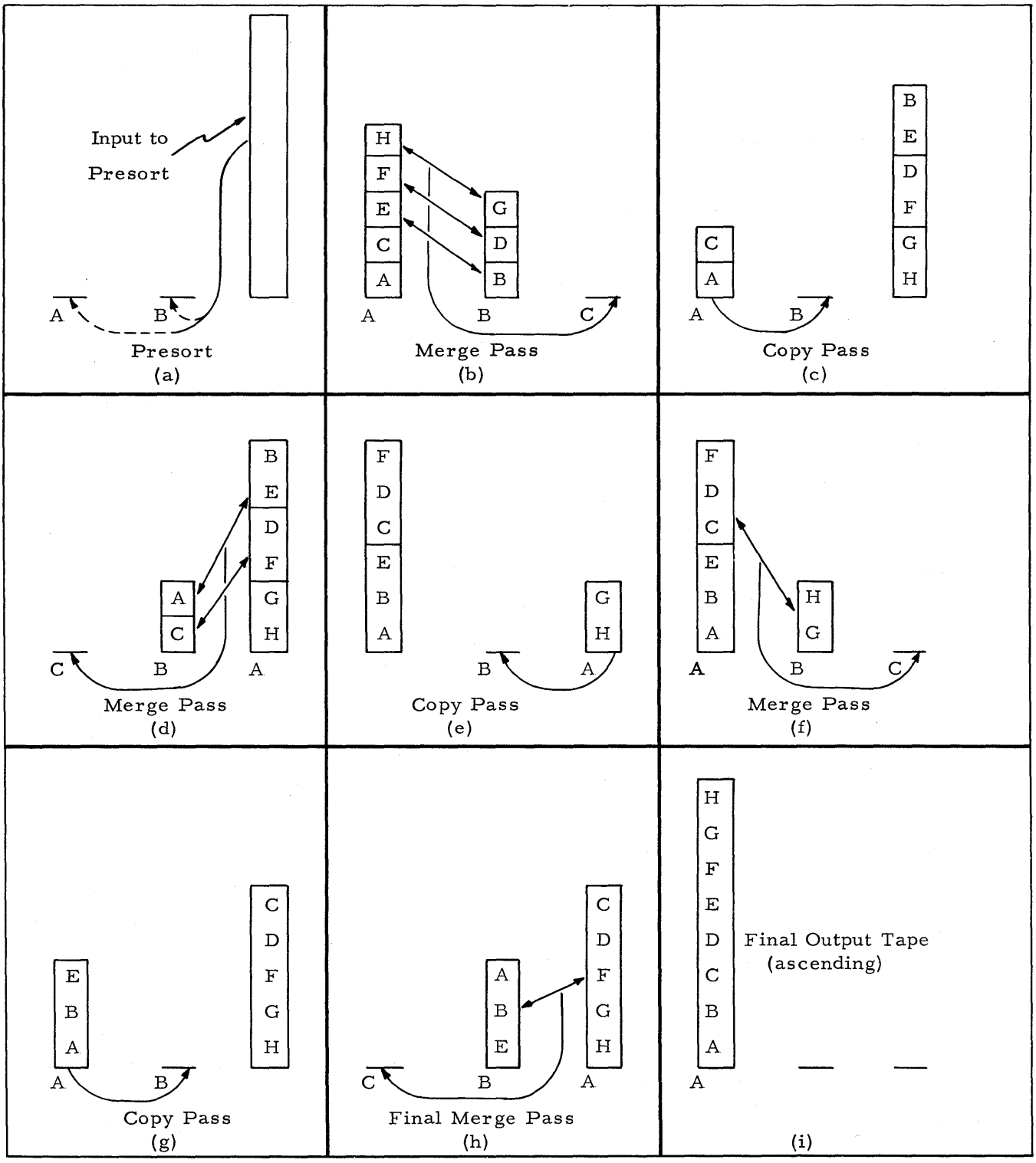


Figure 9. Simple Three-tape Merge Sort

Reading and Writing Controls

The reading technique used in the merge sort differs considerably from that used in the presort. Associated with each input tape is a set of three buffers: a "current" buffer; a "next" buffer; and an "open" buffer. These buffers provide for continuity of data availability. If less than three buffers were provided, the merge program would frequently be interrupted and delayed to await refilling of depleted buffers.

Writing in the merge sort is handled in the same manner as in the presort, using two output buffers, a "working" buffer and a "writing" buffer. Since data comes into the merge sort from two or more tapes but goes out on one tape at a time, the output operation is the limiting speed factor. Since as much information must come in as goes out, ideally one input record should be read each time an output record is written. This balance is realized by using the three buffers per input tape in conjunction with a technique known as "read anticipation". All buffer areas in the merge sort, as in the presort, are essentially the same size as the records on tape.

Read Anticipation

Before an input record is read into memory, the key of the last item in each current buffer is inspected to determine which current buffer will be depleted first, so that the corresponding tape can be read into the next available open buffer of that input set. That tape is then stoppered (temporarily not considered for reading) until the current buffer is depleted. Thus the three buffers provide:

1. A current buffer from which items are being taken in order to always provide input;
2. A next buffer to insure merging will not be delayed when the current buffer is depleted;
3. An open buffer available to allow reading at any time.

Equipment and Memory Considerations

The presort is often machine limited, especially if there are many items in storage and if these items are small. Machine speed is not nearly so critical during the merge sort, since even for the five-way merge, the program must choose only among five items to select the output item. Therefore, the merge sort (except when sorting very small items) always proceeds at tape speed.

The most obvious speed-limiting factor of a merge sort is the number of tapes used by the sort. It will be noted that this number may be dependent on the size of available machine memory. This dependence stems from the fact that three input buffer areas and two output

buffer areas are provided for each tape to keep the tapes moving at maximum speed. Since these buffers must be as large as tape records, a six-tape merge sort requires considerable memory space for buffering alone. If this memory is not available, fewer tapes have to be used for the sort.

Trees (General Description)

The meaning of the word "tree" is the same for the merge sort as for the presort. Like the presorts, the three merge sort routines (single, double, and triple precision) differ mainly in the structure of their trees. Also, like the presort, the triple-precision merge sort tree may be modified with own-coding to accommodate any number of additional keys. However, the merge sort trees differ from those of the presort in structure.

The merge sort uses a "return" tree (see Figure 10), which contains more comparisons and more exits than a corresponding presort tree (Figure 3). Note that there are often several possible exits for one particular item selected (item E, for instance, in this example). The reason for this apparent duplication is to provide one unique path through the tree for each exit. This permits storing a "return" to the tree at the time of exit, so that return can be made to that point along the path at which the selected item was first compared. Thus, when A and B, then B and C, then C and D, and then D and E are compared, and E is selected the smallest, a return is made, after replacing E directly back to the D vs E comparison. Use of this type of tree in the merge sort allows going from a five-way to a four-, three-, two-, and one-way merge without going through any more comparisons than needed for the particular "way" in progress.

There are six trees which are used by the merge sort. Two are required for each precision because the passes of the merge sort alternate between ascending and descending merging; the ascending trees (like the presort trees) find the smallest of the keys of the items compared, while the descending trees find the largest. During any one merge sort pass, only one of the two trees is used, the entrance to it having been set up at the beginning of each pass.

Perfect Distribution of Strings for Merge Sort

The ARGUS presort produces an ideal distribution of strings among the various output tapes. This is done by counting the number of actual strings produced and calculating the additional number needed for a perfect distribution. The number of required dummy strings, as well as the number of passes required, is passed on to the merge sort through the end-of-file identification records. These numbers are stored by the merge sort in a table (one entry for each tape) and are used to effectively create the required number of dummy strings.

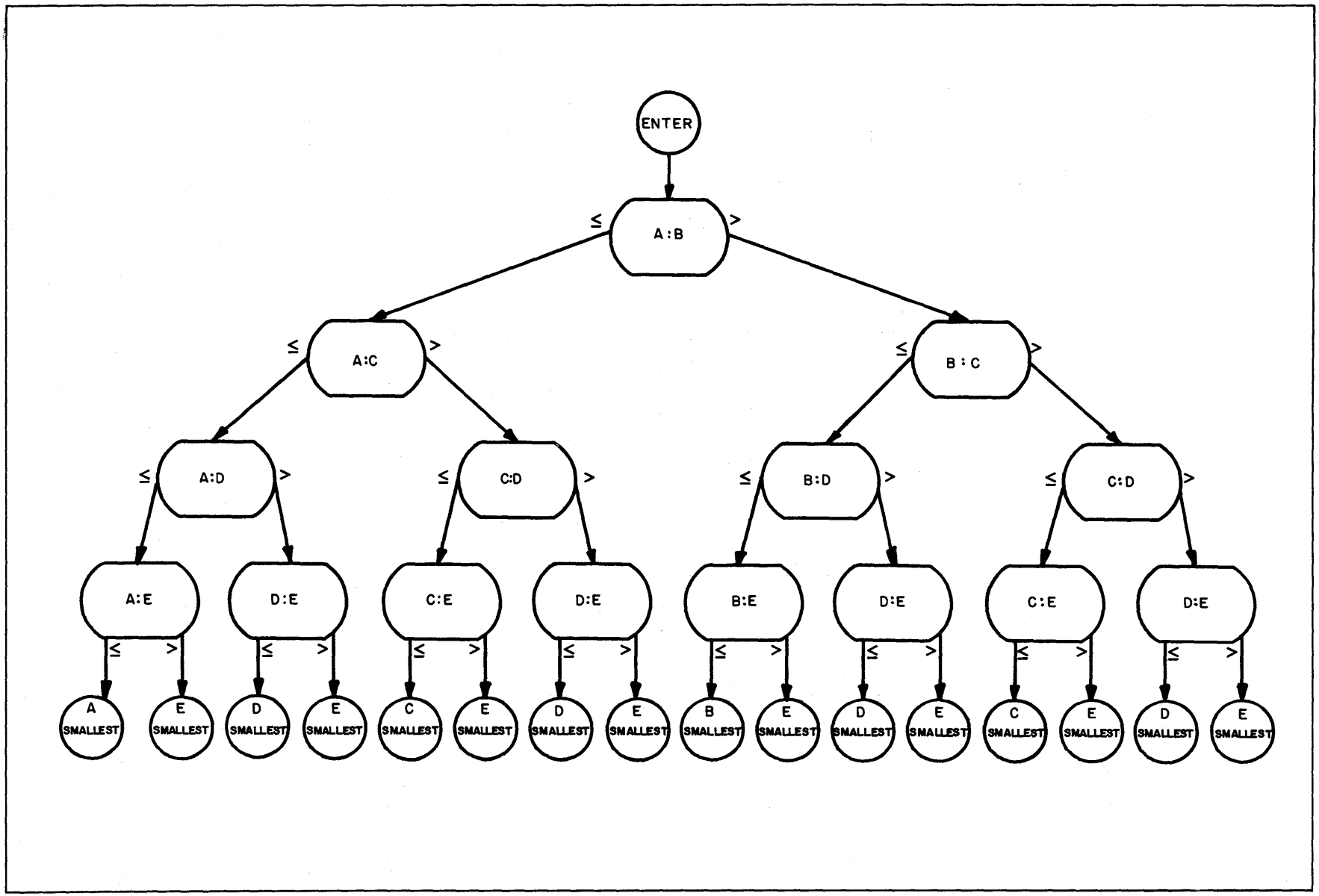


Figure 10. ARGUS Merge Sort Tree

The distributive logic that the presort follows in placing strings on each of the tapes for the subsequent merge is directly controlled by two factors: first, the number of actual strings generated through its sorting operation; second, the "way" of the merge operation that will follow the presort. For example, Figure 11 illustrates a two-way merge of 34 actual strings generated by a presort. (For a two-way merge 34 strings represent an ideal number to be distributed in the ratio 21 to 13, with 21 strings on tape A, 13 strings on tape B.)

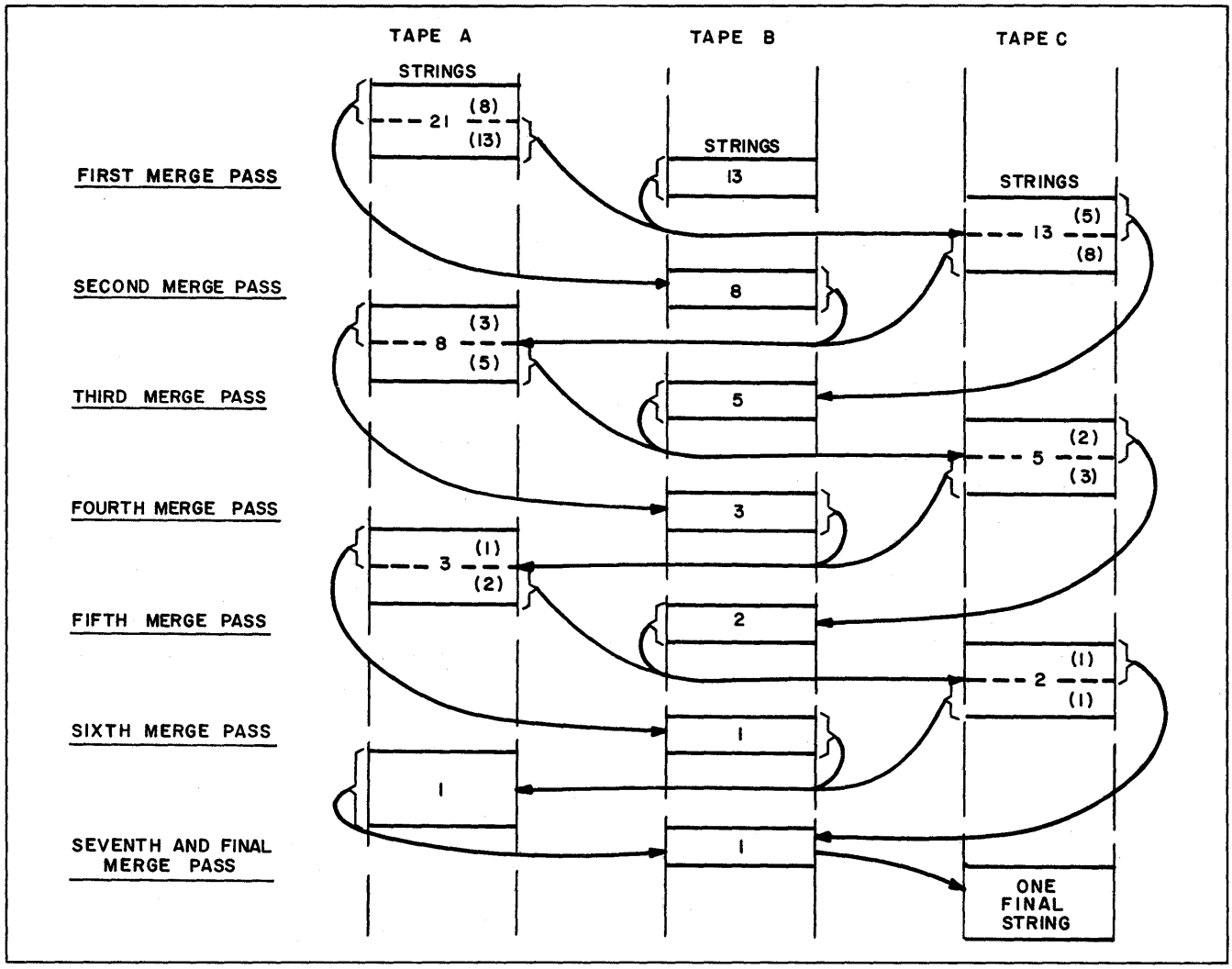


Figure 11. Two-way Merge

This example illustrates a merge where the number of actual strings produced by the presort represents an ideal number for a two-way merge and no dummy strings are necessary to adjust this number. Figure 12 demonstrates a situation where the actual number of strings generated by the presort does not represent an ideal number for a five-way merge.

For example, 2, 318 actual strings are produced by the presort. Six tapes are allotted for the merge sort. For a five-way merge, the presort calculates that 2, 318 strings do not

represent an ideal number to be distributed onto five tapes. Based on this number, it determines that 2,353 strings represent the next higher perfect distribution for a five-way merge. Therefore, 35 dummy strings (2,353 - 2,318 = 35) are added to bring the number of actual strings up to the ideal number.

	TAPE A	TAPE B	TAPE C	TAPE D	TAPE E	TAPE F
First Pass	671	616	511	365	155+35 (or 190)	---
Second Pass	---	55	105	146	175	190
Third Pass	55	50	41	29	15	---
Fourth Pass	---	5	9	12	14	15
Fifth Pass	5	4	3	2	1	---
Final Pass	---	1	1	1	1	1

Figure 12. Five-way Merge

Three basic factors of the cascade technique by which its distributive logic can be best understood are as follows:

1. There is always one, and only one, string (variable length) on each work tape for the last (or final) merge pass.
2. The number of strings on the longest tape after a complete merge pass equals the number formerly on the shortest tape prior to that merge pass. (In Figure 12, third merge pass, 55 strings on tape A represent the largest number at that level, while for the previous or second pass, 55 strings on tape B represents the smallest number at that level.) This relationship carries through the entire merge.
3. The number of strings on the longest tape (for any level of the merge) minus the number on the next-to-longest tape equals the number on the shortest tape for the next lower level of the merge. (In Figure 12, first merge pass, 671 strings on tape A minus 616 strings on tape B equals 55 strings, or the number of strings on tape B for the next or second merge pass. Again, for the first merge pass, 365 strings on tape D minus 190 strings on tape E equals 175 strings, or the number of strings on tape F for the next or second merge pass.) This relationship can be calculated for all tapes throughout the entire merge.

Banner Words

During the sorts, each record on tape contains a banner word. This word identifies the record's content, provides a record count, defines the position of the record in the string, and provides information used for restart purposes.

The banner word of the first record in each string is adjusted by the presort to become a beginning-of-string marker for the merge sort. Other banner words become middle-of-string

markers. Because data is always read backward in the merge sort, the beginning-of-string marker actually identifies the end of a string. Therefore, each beginning-of-string marker calls for the tape on which it is discovered to be stoppered until a marker is found for each tape being merged. All tapes are then unstoppered and a new output string is begun.

The beginning-of-string record of the first string to be merged is preceded by the beginning-of-file identification record, which includes a beginning-of-file marker as its first word. Also because the merge always reads data backward, each beginning-of-file marker actually signals an end-of-file for the merge. Therefore, each beginning-of-file marker calls for the tape on which it is found to become the new output tape, indicating that the "way" of the merge is to be decreased, and a new subpass is to begin. When the number of beginning-of-file markers encountered during a pass equals one less than the number of tapes, the entire pass is complete, and the merge readies itself for the next complete pass (either ascending or descending). Bit number 30 is a "1" for beginning-of-string, and "0" for middle-of-string.

NOTE: The following text refers to many special registers used in the ARGUS merge sort program. Appendix C of this manual provides a list and functional description of these special registers.

Dummy String (DUMSTR)

Dummy strings are calculated by the presort to bring the number of actual strings up to an ideal number for perfect merge distribution. Logically, these dummy strings are processed just like actual strings. However, they are processed before the actual ones so that they can be eliminated as soon as possible and allow uninterrupted merging of the actual ones. At the beginning of each string, the dummy string counters of all input tapes are inspected; if there are no dummy strings, the merging process proceeds in the normal manner. If any input tapes have dummy strings, the corresponding dummy string counters are reduced by 1, and those tapes are stoppered; effectively, those strings have now been processed. If all input tapes have dummy strings, a process is followed which subtracts 1 from each counter, then stoppers, and adds 1 to the dummy string counter corresponding to the output tape. Effectively, a number of dummy strings has been merged together into one dummy string. This method of handling an imperfect number of strings from the presort has the advantage that the ideal ratios are maintained, yet no extra data is processed, since handling the fictitious dummy strings is a purely internal process which takes very little time.

When beginning a string, an End-of-String Switch (SWEOS) is used. (An EOS Switch signifies that the preceding string has just ended.) This switch is normally set to go to the Dummy

String Adjustment Area (DUMSTRE, DUMSTRD, DUMSTRC, DUMSTRB, DUMSTRA). In the Dummy String Adjustment Area there are five groups of instructions corresponding to the E, D, C, B, and A inputs; for less than a five-way merge, SWEOS is set to go to the appropriate intermediate group. At the end of the series of groups is another switch, Exit A Switch (EXITA) which is normally set to add an instruction which will increment the output tape dummy string counter and return to SWEOS.

Each group consists of the following instructions. First, a comparison tests the appropriate dummy string counter for zero. If not zero, 1 is subtracted from the counter and a comparison is then made in the next group. If the counter does equal zero, several instructions are performed to set up the appropriate input buffer to be ready to merge, and EXITA is set to Beginning of String Switch (SWBOS). The process of setting up the input buffer for merging is made clearer in the following section which contains detailed information concerning the buffers. Essentially, the operation consists of unstoppering the appropriate input (insomuch as everything is already stoppered). Thus, if none of the counters are zero, 1 is subtracted from each counter, going from group to group, in turn going from EXITA to an add instruction, which increments the output counter, and returns to SWEOS to start a new string. However, if any of the counters are zero, the program proceeds from EXITA to SWBOS, which sets up the banner switch (in the output area) to write a beginning-of-string banner word, and then to the tree to merge. Since only inputs with zero counters were unstoppered, the tree will merge only the zero counter inputs from which a normal output string will be written. As each input string is ended, it is stoppered. When the tree detects that all inputs are stoppered, it goes to SWEOS to begin a new string. (This also starts a new cycle.)

Buffers

The input buffers are divided into sets, designated A, B, etc., up to E, which correspond to the number of input tapes (from two to five). The physical tape assignment to each buffer set rests on whether the pass is ascending or descending. As far as the reading and writing controls are concerned, the buffer sets remain the same. The A tape is always longest at the beginning of the pass (in terms of strings) and the last to run out. The B tape is next longest, and so on, down to the E tape (if used), which is always the shortest. For a three-tape merge, the two inputs are always A and B. For a six-tape merge, which uses all five inputs at the start of a pass, tape E is the first to be depleted, followed by D, C, B, and A, in that order. Figure 9 shows this relationship for a three-tape sort (two-way merge).

Associated with each of these input sets are three buffer areas in memory. These rotate among themselves, one being termed "current", one "next", and one "open". When the current buffer is depleted, it becomes open, and the other two move up accordingly. To accomplish this

three-way switching with a minimum number of memory locations, use is made of a three-part word, actually a Complete Address Constant (CAC) whose three sections correspond to the ending locations of the three buffers. To switch, this word is shifted (end around) 16 bits.

Associated with the A buffer set are index register X1 and special register R1. The former is used to keep track of the items in the "current" buffer, and the latter acts as an item count to determine when the buffer has been depleted. It will be recalled that all tapes are read backward during the merge sort, so the buffers have to be emptied backward to maintain the correct sequence of items. Thus, X1 is first used as a base of reference for comparing the current A item with the others, and assuming this item is selected, X1 is also used to transmit it to the output area. Then X1 is decremented by the item size (found in the end-of-item word in the case of variable-length items) and R1 is incremented by 1.

When the "current" buffer is depleted, the CAC-type switch is shifted, R1 is reset, and X1 is set to the last item in the new "current" buffer. A location called LAST KEY, which contains the address of the key of the first item in the new "current" buffer, is set up. This location is called LAST KEY because it will be the last key processed from the record. LAST KEY is used to determine which buffer will be depleted first. The coding which accomplishes the switching of the input buffers is in DUMSTR. It will be recalled that an input set is switched and unstoppered in DUMSTR whenever, at the start of a new string, a dummy counter of zero is found. As a string is being merged, the program switches from an input buffer when that buffer is emptied. This is done in the Beginning-of-String (BOS) check by branching off to the coding in DUMSTR. The switch coding is somewhat different for fixed as opposed to variable-size (or over 63 words) items.

To switch a buffer, assuming variable-size items, the CAC-type table is first switched by shifting it with a mask of all hex G's (16 bits) back into itself. Then the variables are set up in an area called Variable Switch (VARSW), which is a common routine used by all five input sets. This is effected with two TS instructions and one TN instruction. The first TS instruction leads (in cosequence) to a common instruction which saves AU2, since the TN instruction will destroy the contents which will be needed later in the routine. The second TS instruction leads (also in cosequence) to the VARSW section, and here an initializing constant of 2 is transferred to R6, a working register. The appropriate CAC table minus 3 is subtracted into R7, which is also a working register that provides the end-of-item word of the last item. Then Z,R7 minus N, R7 is subtracted into Z, R7, which leads to the next lower end-of-item word. R6 is then compared with 2 which will be equal the first time only; if they are not equal, the next instruction is skipped. The next instruction sets up the input index register (X1-X5) by adding 1 to Z, R7 into the appro-

priate index register. R6 is compared (incrementing it by 1) with the constant Number of Items per Block (NIB). If less than or equal, return is made to the Word Difference (WD) of Z, R7 minus N, R7 into Z, R7. This loop will be repeated until the buffer has been worked down to the beginning of the variable-size record, at which time there is an LN instruction. One is added to Z, R7 into the appropriate Last Key Area (LASTKEY-LASTKEY+4), finally dropping from cosequence with a transfer of 1 into the appropriate input buffer counter (R1-R5). Back in the DUMSTR group that came before VARSW, transfer is made from SWBOS to EXITA (indicating that at least one real string must be merged) going next to the DUMSTR group, unless a branch-off had been made from the BOS check. In this case, EXITA is not affected, but return is made to the merge process.

For fixed-size items, the switching instructions in the DUMSTR group are much simpler and faster. As before, the CAC-type table is switched with a 16-bit shift. The buffer index register is then set up with a WD of the CAC-type table minus the constant NPLUS2 into the appropriate index register (X1-X5). The constant WPLUS2 is subtracted from the CAC-type table to set up the appropriate Last Key Area (LASTKEY-LASTKEY+4). Then 1 is transferred directly to the buffer counter (R1-R5). Since these three instructions replace the three instructions which set up VARSW, as well as doing all that is done in VARSW, control is transferred directly to the instruction which puts SWBOS in EXITA, and the program proceeds with the next DUMSTR group. All that has been said for the A input set also holds for B, C, D and E sets, even if the latter sets are not used. Index registers X1 through X5 and special registers R1 through R5 correspond to the five sets A through E. There are five LASTKEY sections (LASTKEY-LASTKEY+4) and five input buffer switches. The CAC-type switches are called Table A through Table E.

Output buffering is done in a similar, yet simpler, manner. There are only two output buffers, and the output buffer switch is divided into two equal parts instead of three. It is switched by a shift order in the same manner. X0 is used to step through the output buffer, and S1 is the output item counter (since R0 must be reserved for restarts). There is nothing corresponding to LASTKEY. Thus, switching the output buffer consists of shifting the switch, and resetting X0 and S1.

The input buffers are primed at the beginning of each pass by filling two of each set (current and next), and starting a read into one of the open buffers (as determined by LASTKEY). This will be the set whose current buffer will be depleted first. After the initial priming, another record is read just before the output buffer is ready to be written. Each time the LASTKEY from a current buffer is used to initiate a read into that set, the LASTKEY area for that set is stoppered. It is unstoppered when the current buffer is depleted and the next one becomes current.

Trees (Detailed Description)

In the merge, the input items are compared directly as they appear in the buffers in order to minimize transfer time. To do this, index registers X1 through X5 are set to the first word of the current input item, and the tree compares via indexed addressing. The augments to the indexed addresses in the tree are set to refer to the particular key locations, the beginning of the item being the base of reference. To stopper an input (for instance, when a beginning of string is found in that set), the corresponding index register is set to a special stopper base, which is set up so that the augment will result in addressing a stopper word of all hex G's for the ascending tree, or all zeros for the descending tree.

Special register S0 is used at the exits of the tree both for storing the return, and for going to the appropriate coding to transfer the selected item to the output area. At each exit there is an instruction "TS x N, S0, y N, S0", where x is a TS sequence change instruction representing the return, and y is an increment of a multiple of 5 representing the item selected. After storing the return in the location specified by N, S0, S0 is incremented to a new value, and the program proceeds to the location thus specified. S0 is set initially to "MERGE" where it refers to a special table arranged in groups of five instructions each. Each group contains all the instructions needed to process the item from one input set. This use of S0 at the exits from the tree allows an ascending or descending tree to be related to the same set of processing instructions. It also determines which input tape is to be read next. This function is performed to synchronize reading and writing every time the output buffer is filled to a certain point. When the time comes to read, the contents of X1 through X5 are stored in STORE, and then the index register is set to the values found in the area termed LASTKEY. A section of LASTKEY is set up for each input set whenever that input buffer is depleted and switched. To stopper an input read, the corresponding LASTKEY location is set to the stopper base. S0 is also set to READ+3 to refer to a table of instructions in the READ section similar to the one used in the MERGE section. In the reading mode, the tree is always entered at the top rather than at the return, since any of the input sets may have been stoppered or unstoppered since the last read. Thus, the return stored in N, S0 is not used, although S0 (incremented) is used to lead to the proper set of instructions to read the next tape.

As in the presort, it is necessary to know when all of the items being compared are stoppers in order to finish one string and start another. The most economical way to make this check is to go to a special section of coding from the one exit to the tree which will be used if all items being compared are equal, and then check if they all are stoppers. Thus, this one exit of each tree increments S0 to a special value, used only for this purpose, which leads to the special checking routine.

Figure 13 illustrates the table of instructions in the MERGE section which are performed when an exit is made from the tree. The first instruction of each group of five is an item transfer, which transfers the selected item from the input buffer directly to the output buffer. Notice that the B address of the item transfer is indexed, with a base at the beginning of the item to be transferred, and with an augment equal to the item size. In the case of variable-size items, or items greater than 63 words, the B address is set to dump the end-of-item word. The item transfer will be terminated either by the end-of-item word thus produced, or by the end-of-item word already associated with the item if it is a variable-size item. Since the input buffers are emptied backward, this will not destroy any useful information, as it would if they were emptied forward. Also, the item size of variable-length items (which is carried in the low-order portion of the end-of-item word) will not be destroyed. The current output buffer setting, addressed through index register X0, is an indirect rather than indexed address in order to minimize the time required for the item transfer.

The next instruction, a WD, sets AU1 (used as a working special register) to the word previous to the item just transferred. This will be used subsequently to obtain the item size of the next item, and (in the BOS section) to check the banner word for the beginning-of-string indication. For fixed-length items, only the latter use occurs. The third instruction of the set compares the appropriate special R register (incrementing it once) with the constant NIB, going to the BOS area when the counts are equal (indicating the input buffer is empty). The fourth instruction is WD, which resets the input buffer index register to the next item. The amount in N, AU1 is used for variable items or those over 63 words long, and the constant Number of Words per Item (NW) is used for fixed-length items as the amount subtracted. The final instruction of the set is a TS sequence change to MERGE+28, which is the common area which modifies the output buffer, resets S0, goes to own-coding, and checks to see if the output buffer is full.

When the tree is in the READ state, exit is made to a similar set of five-instruction groups, in the same way as in the MERGE state. The first instruction, of the five, sets up the read index register, X6, by means of a masked shift of the appropriate buffer switch. The second instruction is the read itself, which reads the appropriate tape into the address specified by X6. The third order transfers the stopper base address to the appropriate position in the LASTKEY area (corresponding to this input set). The fourth instruction has the effect of a multi-word transfer from STORE back to X1 through X5 to set the tree up for merging. Because of timing considerations, a series of TX instructions are used in the cosequence mode rather than making use of one TN instruction. The fifth instruction sets up S0 for the merge mode, and goes to MERGE+33 which will return to the main merging loop.

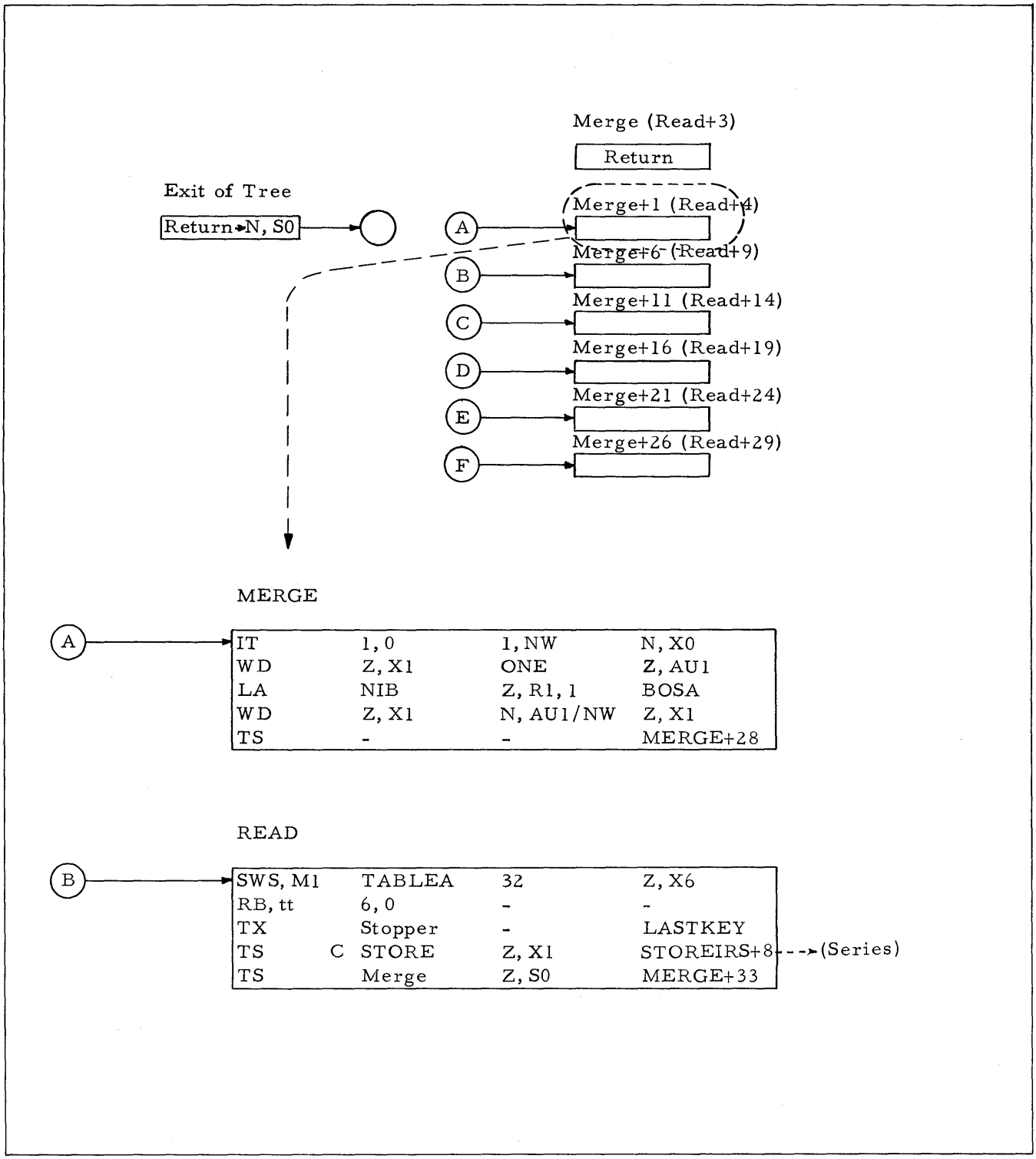


Figure 13. Use of Register S0 at Exit of Tree

Multi-precision

Just as with the presort, it is possible with own-coding to extend the triple-precision merge sort to accommodate any number of keys. One slight difference between the presort and merge sort triple-precision trees is that the merge sort tree accommodates the first and second precision within the tree, rather than just the first. This is done because, as the sort progresses, more equal first keys are expected to be found, necessitating comparison on the second key. Of course, if the second keys are also equal, then some time will be lost in setting up the third (and following) key comparisons.

In modifying the comparison order stored in the COMMON area to accommodate additional keys, care should be taken to modify only the augment portions of the A and B addresses, and not the index register bits, since these are the only means of determining the proper input set being compared. Otherwise, the procedure is exactly as outlined for the presort. The stopper area may cause some trouble in extending precision. At the time of generation, the three specified keys are inspected to determine the largest spread of words between keys, and enough stopper words are set up accordingly. Thus, if keys are in words 5, 10, and 2, an area of nine words of G's (or, during descending passes, zeros) is set up, with the base address being that of the previous word. To insure that this area gets set up properly for a multi-precision sort, the user must specify within the first three keys, the physically first, second, and last key of the item. If these are other than the logical first three keys, appropriate modifications will have to be made, via own-coding, to the COMMON comparison area and possibly to the stopper base address. This is much easier than expanding the stopper area, which is surrounded by instructions and constants. In such a case, the best procedure to follow is to specify a fake third key, leaving the logical first two keys as they are. This fake third key may be any word in or beyond the item. It simply serves the purpose of providing the spread in the stopper area. Then only the COMMON comparison routine, and possibly the stopper base address, need be changed with own-coding.

To further clarify these procedures, assume the case of a 10-word item with keys in words 5, 6, 7, and 1. It is necessary to specify to the sort that the first two keys are in words 5 and 6, as the generator sets up the trees to handle the first two keys. The third key should be specified so that the range of the stopper area is seven words. The third key should, therefore, be specified as word 11. The comparison in the COMMON comparison area will be set up wrong, but this can easily be corrected by the same set of own-coding that sets up the comparison for the fourth key.

Merge and Read Loop

The main loop of the program, that which is performed for each item processed, has been essentially covered in the preceding paragraphs. Only a few instructions are involved in completing the loop. In the tree, the smallest item is found, a return is stored, and the program proceeds to one of the appropriate groups of instructions in the beginning of the MERGE. Here the item is transferred to the output buffer, and the input buffer is adjusted and tested to see if it is empty. Then a sequence change is made to MERGE+28.

MERGE+28 is normally a proceed instruction, but it becomes the branch to own-coding (if called for) during the final pass of the merge sort. Following this, S0 is reset to MERGE, and X0, the output buffer index register, is set to the next item by transferring the contents of AU2 to it. The output buffer item counter (as yet not incremented) is compared with a constant NMINUS1 to see if the buffer is full but for one item. If it is, transfer is made to READ to initiate selecting and reading the next input tape. The first three instructions in READ store X1 through X5 (again by a series of TX instructions in cosequence), set X1 through X5 to LASTKEY areas, and set S0 to READ+3, with a sequence change to TREE. The process of selecting the input set to be read, and the actual reading, have been discussed in preceding paragraphs. At the end of these two operations, X1 through X5 and S0 are reset to the merge mode and go to MERGE+33. If the output counter had not been equal to NMINUS1, control would also have been transferred to MERGE+33. In MERGE+33, the output counter is compared (incrementing it in the A address) with NIB to see if the buffer is full. If it is, control is transferred to WRITE. If not, return is made to TREE. Here the next item is processed, and the loop is completed.

Beginning-of-String Check

After selecting the smallest item and transferring it out, as described in the preceding paragraphs relating to the merge sort trees, the input buffer is stepped to the next item. If, in this process, the input buffer is found empty, control is transferred to the appropriate beginning-of-string section, BOSA through BOSE, depending upon which input set is being dealt with. All five of these sections are similar, and so it will suffice to describe one, namely BOSA.

In BOSA the contents of AU1 are transferred to X1. This replaces the word difference of AU1 minus the item size into X1 which would have been done in the normal MERGE section after the comparison of the item count. The word specified by X1 (the banner word of the depleted record) is compared with a constant to see if it is a beginning-of-string indicator. If it is not, control is transferred to SWITCHA (sections SWITCHB through SWITCHE correspond to BOSB through BOSE). SWITCHA, as the name implies, switches the A input buffers so "next" becomes

"current". At SWITCHA, to save instructions, a "return-and-restore" is set up in the appropriate Dummy String Group (DUMSTRA) and control is transferred there. Just as in the dummy string adjustment routine, the table is switched with a shift instruction, X1 is set to the next buffer, LASTKEY is set, and S1 is reset to +1. Following this is a restore instruction which replaces DUMSTRA+5 with its original contents and return is made to MERGE+28.

If the banner word is a beginning-of-string mark (meaning end of string insomuch as reading is backward), the next record is checked to see if it is a beginning-of-file record. To do this, the table is shifted one position (16 bits) to X1, WPLUS2 (the constant referring to buffer size) is subtracted from this to locate the banner word in the new buffer, and the banner word is compared with the constant Beginning-of-File Banner Word (BOFBAN). If this record is a beginning of file, the SWEOS is set to go to the Beginning-of-File Routine (BOFRTNE), and then proceeds in sequence to BOFA+6. Alternately, if the next record were not a beginning of file, control would be transferred directly to BOFA+6. Here X1 is set to stopper, then going to MERGE+28. BOFRTNE simply sets the banner switch to a special setting which will write the end FID record, and end the current subpass.

In summary, when an item is taken from an input buffer and transferred out, the buffer is stepped, and there are four possibilities:

1. The buffer is not yet empty;
2. It is empty but not at the beginning of string;
3. It is at the beginning of string but the next buffer is not the beginning of file; or
4. The next buffer is the beginning of file.

The first is the normal case, and control is transferred to MERGE+28. The second occurs once per (input) record. Here the buffer table is switched and the index register, counter and LASTKEY are switched. The third possibility occurs once per string and it results in stoppering this input. In the fourth case, SWEOS is set to go to the beginning-of-file routine at the end of this string so that the current subpass may be ended.

All Items Equal

Upon reaching the exit of the tree where all items could be equal, an increment, as mentioned previously, is applied to refer to a unique section of coding that tests to see if everything is stoppered. The merge mode procedure is slightly different from the read mode procedure. In MERGE, end of string is indicated by all stoppers and control is transferred to SWEOS. READ uses all stoppers to indicate that no read should take place (this will be the case at the very end of a pass).

In the merge mode, a test is made first to see if the selected item (A) is stoppered. If it is not, control goes directly to MERGE+1 to transfer the A item, which is either less than or equal to the other items. To determine if the A item is stoppered, the value of X1 is checked directly, comparing it with the constant representing the stopper base address. Thus the test is independent of the value of the key itself (allowing keys of any values to be used in the merge sort). If X1 is set to the stopper base address, the other index registers (X2-X5) are checked in the same way to be sure that all are stoppered. If they are, the program proceeds to SWEOS. If any had not been stoppered, control would have gone to the merge group of instructions corresponding to the first non-stopper set reached (B-E).

Similarly, in the read mode, each index register is tested to be sure it is stoppered. If one is found that is not, control goes to the corresponding read group of instructions. If all are stoppered, control goes to the final portion of one of the sets of instructions to restore the index registers and S0 to the merge mode only (no reading), and then returns to the merge coding.

Write Routine (WRITE)

WRITE is entered once per (output) record, as determined in MERGE, and it is the section which writes and switches the output buffers. Since any break in string, subpass, or pass coincides with an integral record of output, it is in WRITE that a good deal of switching takes place.

First, WRITE resets the output counter to unity by transferring 1 to Z, S1. The address of the first orthoword in the buffer is set up in X7 (by transferring the contents of X0 to it), and then X0 is set to the address previous to the smallest item. Depending upon whether this is an ascending or descending pass, the smallest item is defined as the item in the output buffer with the smallest key, which will be either the first or the last item. WRITE+2 and WRITE+3 accomplish this setting up of X0, and these instructions in turn are set up at the beginning of each pass as explained in the following paragraphs.

For an ascending pass (smallest item is first), WRITE+2 is a WD instruction of the output buffer table minus 1 into Z, X0. WRITE+3 is a PR instruction. Since the output buffer table (which is the two-part switch) is set to the first data word of the current buffer, subtracting 1 will lead to the location just prior to the first item.

For a descending pass (smallest item is last), WRITE+2 is a WD of Z, X0 minus 1 into Z, X0 which leads to the last word of the last item. If the items are fixed, WRITE+3 is then the WD of Z, X0 minus NW (constant for the number of words or the item size) into Z, X0. If the items are variable or over 63 words, WRITE+3 is a WD of Z, X0 minus N, X0 into Z, X0, using the end-of-

item word for the item size.

WRITE+4 compares the key of the smallest item (using X0 with the proper augment) for equality with a constant of all hex G's. If the result is equal, a record of fillers has been accumulated which can be omitted from the output to reduce the amount of data on tape. In this case, control is transferred to WRITE+13. But normally the smallest item will not be hex G's and the program will remain in sequence to write and switch buffers. Thus, in WRITE+5, X0 is set to the banner word position by subtracting 1 from the buffer table. WRITE+6 is the banner word switch, which is similar to the one in the presort. Usually, this transfers a normal banner word to 0, 0. At the beginning of string, the banner switch is set to SETBOS, which transfers a BOS banner word to 0, 0 and, in cosequence, transfers the normal setting to WRITE+6. One additional setting of the banner switch Set End-of-File (SETEOF) occurs when an output tape ends, in which case an EOF banner is transferred to the FID reserve area IDRES and several instructions are performed in cosequence. These instructions add Z, X0 and 4 into Z, X7; transfer four words from IDRES to 0, 0; transfer Set End-of Record Indicator (SETEORI) to the banner switch; and set WRITE+12 to go to the banner switch.

In all these cases, the next instructions performed are WRITE+7 through WRITE+12, which actually perform the write. The record count is incremented and is substituted into 0, 0. Ortho is computed from 0, 0 to N, X7, and 0, 0 is written. Finally, the output buffer table is switched (by shifting end around, as with the input buffer tables) leading to WRITE+12 which is the write exit switch. Normally, this transfers the new output buffer table setting to Z, X0, and returns to MERGE. But during the end-of-file procedure (discussed in the preceding paragraphs), this exit switch is set to return to the banner switch, presently set to SETEORI. In this special case, the banner switch will transfer an End-of-Record Indicator (EORI) word to 0, 0, and continue in cosequence to restore the banner switch to SETBOS. Next, the write exit switch is set to transfer the output buffer table to Z, X0 (as usual), and then control goes to EOF (described below).

Thus, in normal usage, the banner switch either sets up a BOS banner word and restores itself, or sets up a normal banner word. In the special case when ending a tape, it allows a loop to be established going through WRITE twice to write an EOF record and an EORI record. As noted earlier, an output buffer full of filler items (keys of all hex G's) will occasionally be found. In that case, control is transferred to WRITE+13. Normally, such a record can be discarded. However, if it is the only record of a string, dropping it would upset the balance of strings on tape, and the sort would not end properly. To overcome this possibility, the banner switch is checked (in WRITE+13) to see if it is set to indicate that this record is a beginning of string. If it is, control is transferred to WRITE+5 and the record is written in a normal manner. If it is not a

beginning of string, control is transferred to WRITE+12 to discard it. This will bypass the process of writing the record, but will reset the output buffer index register for the next record.

The EOF consists of a straightforward series of instructions (including several substitutes, transfers, and reads). One dummy write and a read backward (into the stopper register) of the output tape just completed is executed. EXITA is set up in the dummy string area to the new "way" according to N, S3, 1 (S3 is the subpass counter used for this purpose, which sets EXITA back to the largest "way" after the last subpass). The SWEOS (EOS+4) is set to refer to the next section in the dummy string area. Three read forward and one read backward of the new tape to be written are set up and performed, each followed by a dummy read, as well as the common write instruction (WRITE+10). The record counter is set to the record count found by the positioning instructions, which leads to a comparison that tests to see if EXITA is set to the largest "way" setting (add to nth counter). If it is, control is transferred to End of Pass (ENDPASS). If not (meaning that only a subpass has been completed), EOF proceeds to modify some of the EOF instructions for the next time through. The A addresses of two of the instructions which pick up the new tape addresses are incremented (with WA) instructions. The WRITE exit (WRITE+12) is restored to normal, and control goes to SWEOS to start the next string.

End of Pass (ENDPASS)

As noted above, upon coming to the end of an input file, a check is made to see if it is the last file depleted. When the last file is depleted, control is transferred to a section of coding called ENDPASS. ENDPASS determines whether another pass is to be performed, which type (ascending, descending, or a special last pass), and modifies the merge sort routine accordingly.

First, a counter called PASSES (which starts with the total number of passes as determined by the presort, and is reduced by 1 after each pass is completed) is compared for less than or equality with 2. If less than or equal, the next pass will be number one (the last), and control is turned over to a special set-up section termed Last Pass (LASTPASS). Otherwise, the program continues in sequence. Working register R7 is set to READ+5 (the first of the sets of read instructions in READ) for later use in ENDPASS. Switch tree (SWTREE), an indicator which shows which type the current pass happens to be, is checked and control is accordingly turned over to either APASS or DPASS to set up an ascending or descending pass. These two sections of coding are similar, and they set up all areas which vary between ascending and descending passes. In APASS or DPASS, an initial "return" to the top of the appropriate tree is set up in READ+2. A word of all hex G's or a word of all zeros is stored in STOPPER. (In double precision, two such words are stored. In triple precision, the area between the two extremes of the stopper area are filled.) The final orders in the EOF area are modified, as

necessary, to step the EOF area one way for ascending passes and the opposite way for descending. The instructions in WRITE+2 and WRITE+3 are set up to refer to the first or last item in the output buffer. A small loop is used to set up the read instructions, using R7 (previously set up) with an increment of 5 to step through the read groups. SWTREE is switched to its opposite value.

The coding, which is common to both APASS and DPASS, is reached at this point. Here, 1 is subtracted from PASSES, and the write exit switch (WRITE+12) is restored to its normal value. The next section is called Switch Counters (SWCTRS). This section switches the dummy string counters around to correspond to the new pass. It will be recalled that each counter corresponds to a specific tape, but that the A through E designations change from pass to pass, and the dummy string adjustment area modifies counters on an A through E basis. This section of the routine, although quite straightforward for an individual sort, is set up by the generator differently for a three-, four-, five-, or six-tape sort. Thus, for a three-tape sort, counters A, B, and C become C, B, and A. From SWCTRS, control transfers to the Beginning-of-Pass Section (BEGPASS), an initializing routine entered at the beginning of the merge sort as well as before each further pass. The LASTPASS coding, mentioned above, is explained later in the ENDSORT discussion.

Beginning of Pass (BEGPASS)

BEGPASS is used at the start of every pass, including the first. Its basic function is to prime two buffers of each input set and to set up an additional read, based on depletion, when MERGE is entered the first time. The output buffer is initialized, and the input index registers are stoppered. The SWEOS, and the instructions in EOF which modify it, are reset to the proper initial value in relation to the dummy string adjustment area. MERGE is set to perform a read, based on expected depletion, when it is first entered after the dummy string routine. And finally the banner switch is set to an initial BOS setting, ready to write the first output string on the output tape.

The buffer is primed using R6 and R7, working special registers. R7 is set to the first actual read instruction (READ+5) in READ. READ+26 is temporarily set to an instruction which will replace itself and return control to BEGPASS. R6 is set to the A CAC switch (TABLEA). A loop is then entered which uses R6 and R7 to set up the read index register (X6) and perform three reads under cosequence control (the first simply bypasses the EOF record), during which R6 is incremented by 1 and R7 by 5 to refer to each input set in succession. During this process, the output buffer counter (S1) is set to zero, the subpass counter (S3) is set to its initial value, so that N, S3 will refer to the proper constant when it is used later to modify EXITA. The con-

tents of the output buffer table are stored in X0, and a special initializing switch is stored in MERGE to allow use of the TREE for the one initial read, but then control returns to the top of the tree for merging. X1 through X5 are set to STOPPER. R6, the working register, is again set to TABLEA, and a small loop is used to shift each of the input buffer tables 32 bits around to allow for the table switching that takes place in DUMSTR. An instruction in EOF, common to both ascending and descending passes, is restored to its initial value (it becomes modified at the time each subpass is completed). The banner switch is set to SETBOS and control goes to SWEOS to enter the dummy string adjustment area and begin the first string.

Ending the Merge Sort (ENDSORT)

When PASSES is equal to 2 at the end of the next-to-last pass, control goes to a section called LASTPASS to set up all the procedures unique to the final pass of the merge sort. In LASTPASS, preparation must be made for own-coding, if specified. The final output tape must be positioned back one record to eliminate the second BOF record, and the banner switch must be set to always write a normal (not BOS) banner word. Finally, the EOF section must be modified to go to ENDSORT, rather than ENDPASS, at the completion of the string. ENDSORT simply prints "MERGED" and exits back to the original macrocoding.

It will be recalled that MERGE+28, when immediately following the transfer of an item from the input buffer to the output buffer, is normally at PR. LASTPASS transfers the entrance to own-coding, if any, to this location. The output tape is read backward once into the stopper buffer. BEGPASS is modified to skip the initialization of the banner switch, and the banner switch is set to transfer a normal banner word. Finally, LASTPASS replaces the read forward (RF) instruction in EOF, which starts the positioning of the next output tape. A sequence change to ENDSORT is transferred into this location. LASTPASS then goes to the comparison which determines whether to go to APASS or DPASS to perform the basic set up of the final pass.

In the final pass, the first tape to be written (the final output) has been backed up one record. APASS and SWCTRS are gone through in the usual manner, eventually leading to BEGPASS. BEGPASS is the same except for the last instruction, which no longer sets the banner switch to SETBOS, but simply goes to SWEOS to start the final pass. Since all dummy string counters, by now, will be reduced to zero, this section is completed. Also, all inputs are unstoppered, and control goes to TREE. This exits to MERGE in the normal way, except that immediately after transferring an item to the output buffer, control is turned over to own-coding, if specified. Since the banner switch is set to normal, the banner word of the first data record to be written is the same as all the others to follow (instead of being a BOS banner word). One by one, the input tapes will reach end, but merging continues until the

items compared are stoppered. This leads to SWEOS, which was set to transfer SETEOF into the banner switch each time an input tape reached end. From here, control is turned over to WRITE to write the EOF and EORI records, and then to EOF section. This backs up the output tape one record, and leads (because of LASTPASS modification) to ENDSORT. In ENDSORT, "MERGED" is printed, and exit is made from the routine.

SPECIAL CASE: One Item per Record

In the generator portion of the merge sort, (discussed later) specific parameters of item size, key location, items per record, etc., are used to set up a specific routine. In most cases, the specific routine will be similar or identical to the general one described in the preceding paragraphs of this section.

The case of a single item per record requires some special handling, however. The most obvious problem is in the main loop of the program, where the time to read is determined based on filling the output buffer but for one item. With one item per record, this difference between "n-1" items and "n" items amounts to a full buffer, with corresponding problems in synchronizing the reads and writes.

Thus, in this one case, the generator modifies the reading comparison in MERGE+31 to compare on NIB instead of NMINUS1 (actually the contents of NMINUS1 are changed). Also, the exit of the dummy string adjustment area has been set up to start a read based on expected depletion at the beginning of every string (rather than at the beginning of every subpass). This is necessary because of "n-1" occurring one full record apart from "n". The result is that a read is skipped at the end of each string, since everything is stoppered at the time the last read should take place. The extra read at the beginning of the string, then, represents a read at "n-1" relative to the first item of the string to be transferred out.

Over-all Flow of the ARGUS Merge Sort

Thus far, the merge sort has been discussed in general terms, and various components of the merge sort have been explained in detail. These components are tied together in the following paragraphs to present a complete merge sort picture. A merge sort flow chart is shown in Figure 14.

The first section to be performed, BEGPASS, is entered before each pass of the sort, including the first. BEGPASS primes the buffers, and initializes variables and switches.

The EOS Switch (which might more appropriately be called the BOS Switch) is entered at

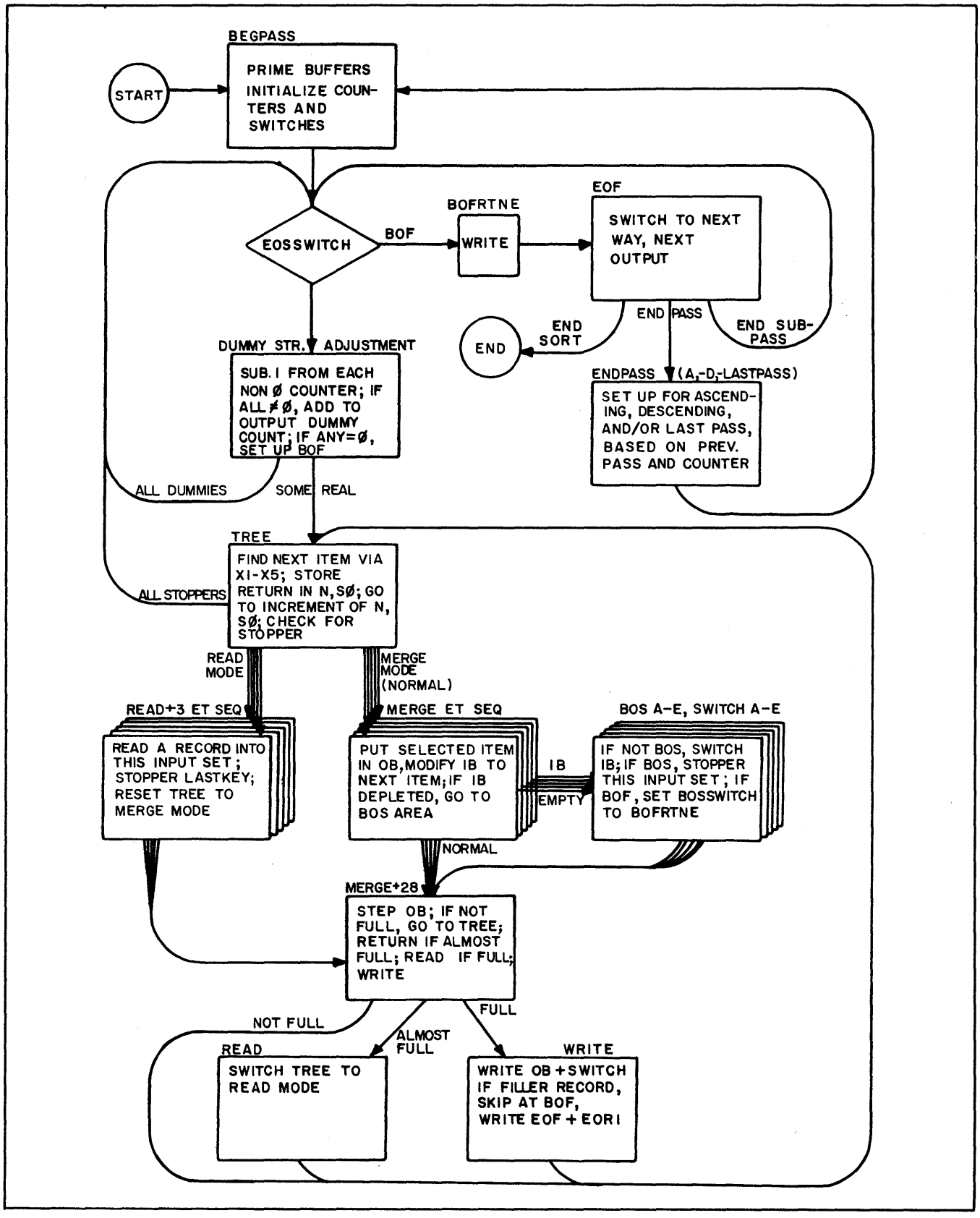


Figure 14. Over-all Flow Chart of the ARGUS Merge Sort

the beginning of each string. Normally, it leads to the appropriate section of the dummy string adjustment area, except that after any input tape is exhausted (and the string then in progress is completed), it is set to the Beginning-of-File Routine (BOFRTNE).

The dummy string adjustment area handles the merging of any dummy strings. If there are dummy strings on an input (as indicated by the dummy string counters), it subtracts 1 from each counter, adds 1 to the output counter, and returns to SWEOS for the next string. If any inputs do not have the dummy strings, the dummy string adjustment area exits to the TREE, after setting the banner switch in WRITE to BOS, and subtracting 1 from any non-zero dummy string counters.

TREE is associated with the quantities to be compared via the index registers, and with the instructions which follow it via special register S0. Through these special registers, TREE is either associated with the current input items and the MERGE, or with the last items of the current input buffers (LASTKEY) and the READ coding. The merge mode is the normal one. In this mode, the TREE will exit to one of several sections of coding to transfer the selected item to the output buffer, step the input buffer from which the selected item came to the next item and, if the buffer has been depleted, go to one of several sections of coding in BOS to switch that input set.

After the item has been transferred to the output buffer and the input buffer has been stepped and tested for depletion, a common section of coding, MERGE+28, is entered. This section steps the output buffer and tests to see if it is full but for one item, or if it is completely filled. If neither of these conditions are met, return is made to the TREE to select the next item.

If the buffer is almost full (one item to go), control goes to READ, which sets the TREE to the read mode, and proceeds to TREE. This compares the LASTKEY areas to find which input area needs refilling the most, and accordingly exits to the appropriate group of instructions starting at READ+3. Here a record is read into the vacant buffer of the selected input set, its LASTKEY word is stoppered, and TREE is reset to the normal merge mode. Return is then made to TREE to select the next item.

When the output buffer is full, control goes to WRITE to write the record on tape and switch the output buffers. If the output record contains redundant fillers only, WRITE is bypassed. If the banner switch is set to BOS, a BOS banner word is written with the record, and banner switch is set to normal. When set to normal, the banner switch writes a normal (middle-of-string) banner word. From WRITE, return is made to TREE to select the next item.

After an input buffer was stepped and found depleted, BOS, which corresponds to the input set of the depleted buffer, checks the banner word of the buffer. If it is a normal banner word, control goes to SWITCH, which corresponds to this input set, and the input buffers are switched, setting up a new last word in the process. If the banner word is a BOS banner, the index register corresponding to this input set is set to stopper, and the next record of this set is checked to see if it is a BOF (meaning the input tape has just been depleted). If it is, the SWEOS is set to BOFRTNE, and merging of the current string continues.

One by one, the different input strings reach end and become stoppered. Finally, TREE discovers that all items being compared are equal and, furthermore, that they are all stoppered. (When in the reading mode, this simply causes skipping a record.) If all items are stoppered, the TREE exits to SWEOS, signifying that this string is ended and another is to be started If the SWEOS is set to its normal setting, control goes to the dummy string adjustment area to begin a new string.

If an input tape (as opposed to a string) had been depleted earlier during the current string, SWEOS would have been set to BOFRTNE. In this case, when the string ends, control goes to the SWEOS, which now leads to WRITE with the banner switch set up to write EOF and EORI records on the output tape. In this case, control goes to EOF from WRITE, which modifies the routine for the next "way", and makes the input tape just depleted the new output tape. If this is not the last subpass, then control goes to SWEOS (now reset to normal) to start another string. Or, if this was the last subpass (meaning the final input tape has been depleted), control goes from here to ENDPASS.

In ENDPASS, a check is made to see if an ascending or descending pass has just been completed. ENDPASS then gets ready to do just the opposite type of pass. This is accomplished in either APASS or DPASS, from which, as at the beginning, control goes to BEGPASS. Upon ending the next-to-last pass, however, control goes to LASTPASS also, to make a few modifications to the program for the final pass. From LASTPASS, control goes to APASS, and then on to BEGPASS. Among other changes, LASTPASS modifies EOF so that, upon completing the first (and by definition, last) string of the final pass, EOF writes the EOF and EORI records and exits from the sort.

Merge Sort Generation

Two special registers are used to relay information from the presort to the merge sort. Index register X7 contains the address of the macrocoding as set up at the beginning of the presort and R1 contains the peripheral address of the tape drive from which the merge sort can

obtain the sort parameters. If output edit own-coding is used, the programmer must set up special register S2 with the address of the own-coding. The own-coding must be under the control of the cosequence counter and the contents of the special registers used by own-coding, must be stored and restored, as for the presort.

After reading the end-of-file identification record, from the tape specified by R1, into memory, the merge sort modifier checks the various options transmitted to it by the presort from the beginning FID parameters. If variable-size items are specified, modifications are made to the sort to use the item size indicated in the low-order 16 bits of the end-of-item word rather than a fixed item size constant. If the banner word option indicates that banner words are not used on data records on the input file, the last merge pass is modified to eliminate the banner words affixed by the presort. If masked keys are indicated in the parameter, the necessary masks are set up in memory as given in the end FID, from the presort, and the merge sort is modified to handle masked keys.

Tape addresses and the "way merge" indicators transmitted from the presort are used to generate the proper read and write instructions and to modify the merge sort as required. Input buffers are allocated for the given record size for each input tape, and output buffers are allocated for the given record size.

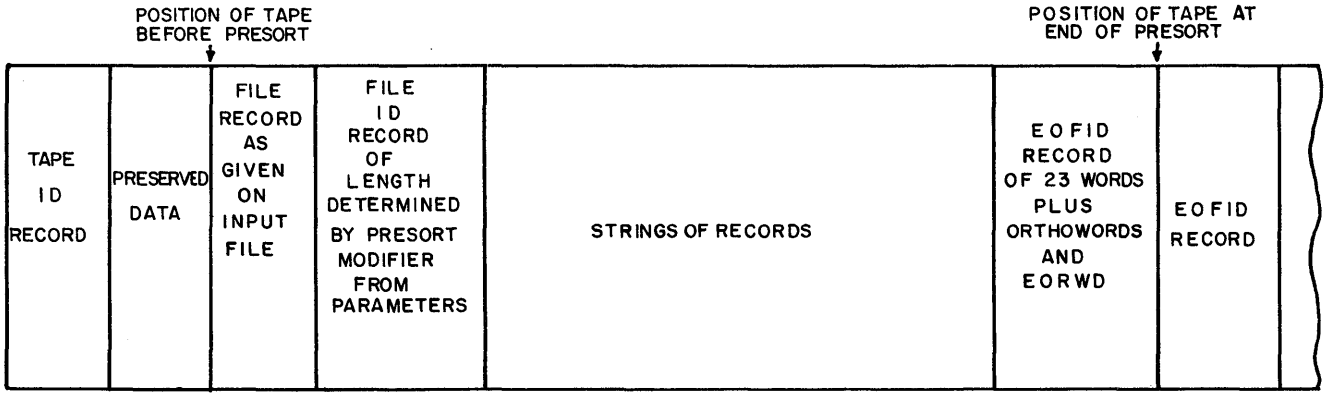


Figure 15. Appearance of Work Tape at End of Presort

Error Correction and Restarts

In the event of a read error, the location of the suspected record is determined, and control is turned over to the Executive Routine to try to repair it. If unsuccessful, the information is reread by the sort, and if still bad, control is again turned over to the Executive Routine. This process is repeated several times. Suitable printouts at the console indicate the nature and disposition of the trouble.

If physical end of tape is reached, the tape is rewound with interlock, and a printout tells the operator what has happened. He may then mount a longer tape, and the program will revert automatically to the most recent restart point.

Restarts in the merge sort, as in the presort, are part of the sort coding. This is especially necessary in the merge sort, since (due to the nature of data manipulation on tapes) restart points must be established at every subpass. (These built-in restarts are especially tailored for the sorts and are therefore more efficient than the general restarts provided by the Executive Routine.) As with the presort, a restart point is established just after the routine is loaded, this time on the last, or nth work tape, to allow restarting the routine from the beginning. Subsequent restart points are stored internally, in the form of counter settings, at the beginning of each subpass. All of the dummy string counters, as well as the record number counts (from the banner words in the "current" input buffers), are stored at this time, and the restart routine is modified to handle the type of subpass about to be performed. These restart points are established during the EOF routine when all inputs are stoppered, and no partially depleted input buffers remain.

There are several possible restart procedures. Use of the proper one is governed by how far the merge sort has progressed at the time it is necessary to restart, and upon what type of subpass is being performed. During the first subpass of any pass, when writing on the nth tape, restarting is accomplished by positioning all the input tapes forward to the EOF records, and positioning the output tape backward to the BOF record. Then, with tape positioned as at the beginning of the pass, PASSES is incremented and control goes to either APASS or DPASS. During intermediate subpasses, restarting is somewhat more complex, since the input tapes will not necessarily be at end of file at the beginning of the subpass. For this reason record counters are stored, corresponding to each input tape. Using these counters, which were stored during the most recent break between subpasses, all current input tapes are repositioned forward to the record counter that is stored. The output tape is positioned backward to the beginning FID. The deficiency counters are replaced by their values, which were stored at the most recent break between subpasses, and all tapes are stoppered. The current input tapes are primed (two buffers each, as in BEGPASS), and those instructions in BEGPASS, which are the same for any number of input tapes, are performed. The routine is then re-entered at SWEOS (just as in BEGPASS) and the current subpass is restarted.

SECTION V
THE COLLATE

The ARGUS collate combines from 2 to 99 ordered files into one long file. Each input file may be contained on a single reel or may occupy any number of reels. The programming logic of the collate routine resembles that of the merge sort, although the two routines differ in function and in outward characteristics. The heart of the collate is a merging operation, accomplished by means of trees, but the associated reading and writing controls are more complex than those of the merge sort.

Like the ARGUS sort, the collate is stored with the Library of Routines as a subroutine and consists of a skeleton routine and a modifier-generator. Whereas the presort and merge sort are generated and performed by executing a single pseudo instruction, collating is a separate operation which is called out by executing a collate pseudo instruction. Therefore, although the collate may be used to combine the outputs of several sorting operations, it is performed as a separate program, completely disassociated from the sorts which produced the files to be collated.

The "Way" of the Merge

In addition to the parameters supplied to the sort (e.g., item size, key location, etc.), the collate is supplied with information concerning the number of files to be merged as well as the "way" merge to be performed. A file here is assumed to be a single series of ordered items, which may, or may not, extend over several reels of tape. "Way" means the number of input files merged at any one time. This is indirectly limited by the number of tape drives available. Whereas a file contained on a single reel of tape may be read from a single tape drive, a multi-reel file is usually allotted two drives. In this mode, no time is lost when one tape (or segment) of a file is depleted because the machine has immediate access to the subsequent section on the alternate drive; therefore, the depleted tape can be replaced, when it is convenient, by the next installment.

Merging Function

In the simplest case, the collate is used to merge two, three, four, or five files to form one; an example would be combining the (sorted) outputs from several weekly runs at the end of a month. In such a case, a two-, three-, four-, or five-way merge (one single logical pass) would be performed depending upon the number of weekly outputs to be merged. (In

this case, five would be a maximum.) On the other hand, there are times when there are more files to be merged than the number of the "way" that the collate can handle. In such cases, unlike the monthly operation just mentioned, it is necessary to perform several passes which, for instance, combine files A and B into a file W, C and D into file X, E and F into file Y etc., and then combine files W, X, Y etc. into a final file. Therefore, in multi-pass merging, it is important to have a firm system, controlling the sequence of files to be merged. As the number of original input files becomes greater (for instance 20 rather than the six), the need for a firm controlling system becomes increasingly more imperative because, without such a system, it would be highly confusing to maintain a fixed control over the entire process.

Equipment and Memory Considerations

In the collate, buffering, reading, and writing are similar to the corresponding portions of the merge. As with the merge, an optimum balance between reading and writing operations is established to make the routine as fast as possible. The tree portion of the routine is simpler than that of the merge since a collate performs only an ascending pass over all data, requiring only a single tree. In addition, the input buffers are easier to visualize because reading is always in the forward direction and information is always taken from the top of the buffers rather than from the bottom.

In spite of the apparent simplification of the collate over the merge sort, there are several factors which could make a collate (particularly one involving small items with large keys) slower than a corresponding merge. For one thing, a greater equality of keys can be expected as strings become longer and longer, extending over one or more tapes. Each case of equality of keys necessitates extra levels of comparisons in the tree, as well as in the comparison section which may be attached. As a precaution, a sequence check has been built into the collate. This, in effect, checks each item coming in to be sure that it is equal to or greater than the item which preceded it on the same file. In many cases, such a safety device can often isolate incorrectly written tape before it has a chance to destroy the whole collating process. This means that two sets of comparisons, the tree and the sequence check, are performed for each item processed. If a break in sequence is found, a printout on the console typewriter informs the operator. The operator then has the chance to redo that tape and, through the restart procedure, start the collating process over at a point before that tape was first read.

The Collate Plan

When the collate pseudo instruction is executed and the routine generated, the generator devises a plan which represents the most efficient run for the conditions specified and prints

this plan on the console typewriter. The operator follows the collate plan in mounting tapes and uses it to track the progress of the collate.

If the number of files to be merged does not exceed the way of the collate, the plan is relatively simple. However, if the number of files is large enough to require more than one pass, a more complicated plan is devised which minimizes the total number of passes over the input files. This is accomplished by utilizing, as nearly as possible, the full way of the collate during each individual pass.

Figure 16 illustrates this principle in terms of two collate runs, A and B, each of which uses a three-way merge to combine 17 files. In example (a), five three-way merges and a two-way merge are first performed to reduce the original 17 files to six. Two three-way merges then reduce these to two files which are finally combined by a two-way merge. Note that each of the three layers of merges processes all 17 files for a total of 51 file times. Example (b) represents a more powerful collate of the same files. Note that by withholding certain files from the first-layer merging, a full three-way collate can be performed in each individual merge and, in this instance, the same files can be combined in a total of 46 file times. This is typical of the manner in which the advance-planning feature of the ARGUS collate results in the most efficient plan for any collating program.

Calculation of the Plan

The calculation of the plan occurs as a part of the modifying-generating process, before the collate is run. The theory of the calculation is relatively simple, and is based on the fact that if a less-than specified way pass has to be done, it is better done at the beginning of the run when only a few files are involved, rather than at the final pass when the entire volume of data must be passed. Therefore, the final pass is planned first and is specified as full-way. If there are more original files to be collated than the number that this would handle, then the next-to-last pass is specified. This is another full pass, whose output will be one of the inputs to the final pass. This yields a capacity of way-1 additional input files (-1 being the input of the last pass which is presently taken up with the output of the new pass). If this is not enough, another pass is specified whose output will be another input of the final pass. This process continues (going to the inputs of the next-to-last pass when those of the last are filled, etc.) until the total number of inputs is equal to or greater than the number of files to be collated. The last pass calculated (the first performed) may be less than a full-way merge, but this will be the only one. When completed, the plan is printed in a tabular manner on the line printer or console typewriter. Each line represents a pass, and specifies each input and output file by a unique number.

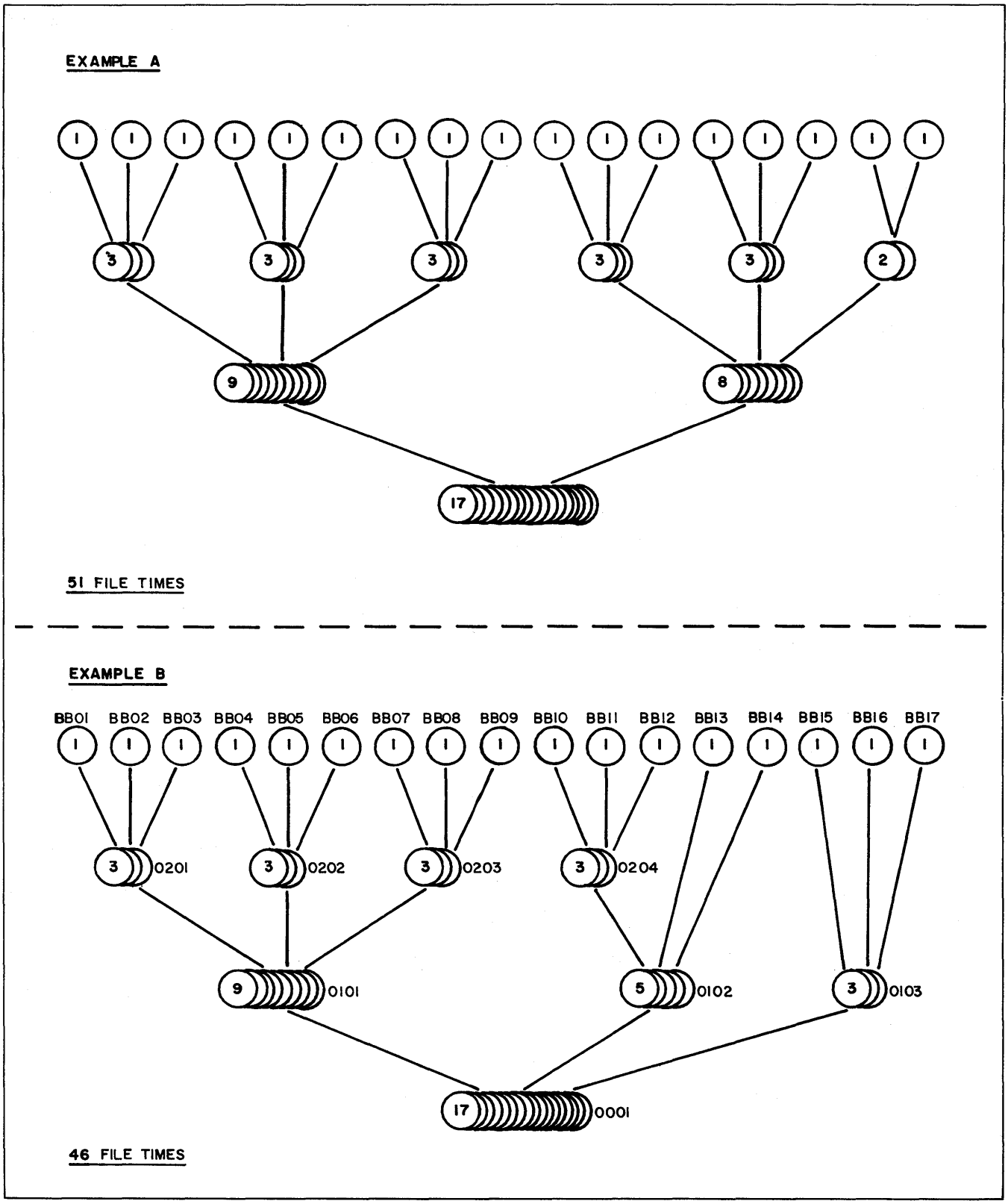


Figure 16. Collate Merging Sequence

The plan is also kept in memory in compacted form and will be used later by the routine to determine how many passes there will be, and what files are involved in each. The file numbers of the plan are used as a basis for writing a file name on tape, printing it on the console, and later for checking the same tape when it is to be read again.

Tape Control

In connection with the collate plan, there is a system of communication between the machine and the operator with regard to the identification of each file going into or coming out of the system. In fact, one of the significant advantages of this plan is the fact that it serves as a check to be sure that the proper files are mounted at all times. This becomes extremely critical during a large-volume, multi-pass collate when tapes are being constantly mounted, dismounted, and changed. To implement this communication between the operator and the machine, each file is assigned a unique number in the plan. This number becomes the file name of each file written by the collate, and as each tape is completed, the number is printed on the console typewriter so that the operator can label the tape in the same manner. Each file written by the collate, and later read by it, is checked at the time of reading, to insure that it is the file being called for at that time. As a further check, each segment (tape) of a file is assigned a sequential number, and this too is checked. The file name for the first segment of any original file, obviously, cannot be checked, but the name which is found there is retained so that the following segments of the file can be checked. The final output may have any file name specified.

File Identification

File numbers, or names, are assigned in the following manner. Each file will have a four-digit number. The first two digits specify the level number. The last two digits specify the file number within that level. Level refers to the number of times the data within the file must be passed before the collate is complete. Thus, the final output is level zero; those files which are merged to create it are of level one; the files merged to create any first-level files are of level two, etc. The file numbers are assigned sequentially, starting with 1. Therefore, the final output file will be 0001 (level zero, file 1) and, depending upon the way of the merge, it will be created by the merging of 0101, 0102, etc. (Figure 16 (b)).

Initially, the plan section sets up a complete table of passes, with file numbers assigned as above. It then determines which of the files are original inputs, and reassigns new numbers to them, with "BB" as level number and sequential file numbers. Thus, when the collate finds that an input file having a BB number is called for, it bypasses the check for file number and instead stores the file name which it finds on the tape. Also, at the beginning of each pass, the

output file number is checked against 0001; if they are found equal, the option to insert a user-specified file name is exercised.

The file name or number, described above, is identical for each tape of a file (by definition, a file may extend over any number of physical tapes), so in order to insure that tapes of a file are mounted in correct sequence, some further check is necessary. This check is provided through the segment name or number. Tapes written by the collate to make up a file are numbered sequentially in the rightmost two digits of the segment name (01, 02, etc.). When these tapes are read, this number is checked for unity at the beginning of a pass, and for the next higher number for each subsequent tape of the file. On any one tape, these numbers are the same in the beginning- and end-of-file identification records, except that the last tape of the file ends with a segment number of GG. As with the file name, the segment name of the final output may be specified by the user, except that the last two numeric positions will be sequential numbers placed there by the collate. Initial input files (those with the BB numbers) are not checked for segment number unity at the beginning of the pass. Instead, whatever segment number is found is stored, and the next tape is checked for one greater than the one found. Thus, the input numbering sequence is apt to begin with any number.

The plan, therefore, which is printed at the beginning of the collate will have as many BB file numbers as there are files to be collated. The user arbitrarily assigns one of these numbers to each file so that he can know what tapes to mount as the routine calls for them. The only requirement of each of these initial files is that their file name be constant throughout, the final two digits of the segment name increase sequentially throughout from tape to tape, and the final end segment name have a word ending in hex G's (GG). As each output tape is completed by the collate, the console typewriter will print out: "REMOVE (file number) (segment number) FROM (peripheral address of drive)". The operator should then remove the tape from the specified address and label it with the file and segment numbers specified. Subsequently, when the plan calls for that file to be processed, there will be no question about which tape is to be mounted.

Tape Changing

The collate is set up to handle each input and output on two tape drives each, as specified by the user. Internal indicators tell the collate which drives correspond to each set. If a single drive is specified, the two internal drive indicators corresponding to that set refer to the same drive. In either case, as each tape is completed, it is rewound with interlock and the internal indicators are switched to refer to the alternate drive (unless the tape just completed was the last of that set). If dual drives are used, the next tape will be processed immediately and the

operator can change tapes at his leisure. If a single drive is used, the routine will stall on the tape interlock until the new tape is mounted (as it would with dual drives if the tape were not changed in time). Although the routine tells what tapes are to be removed, the mounting of the correct tapes (based upon the plan) is left to the operator. This is because the routine (if dual drives are used) has no way of telling when any file will end until the end-of-file identification is reached. By this time, the next tape (if there is one) should already have been mounted on the alternate drive; if not, the routine will stall. The console typewriter prints an indication as each pass is begun, and this, together with the knowledge of what drives correspond to each input set and to the output, enable the operator to interpret the plan and mount the tapes accordingly. In the event an incorrect tape is mounted, and just when the collate is ready to use it, WRONG TAPE, together with the tape drive address and the number of the correct tape, is printed on the console typewriter.

NOTE: The following text refers to many special registers used in the ARGUS collate. Appendix D of this manual provides a list and functional description of these special registers.

Buffers

Except for the differences which arise from reading forward instead of backward, buffering in the collate is handled in exactly the same manner as in the merge sort. This is described in Section IV.

Each of the five input sets, A through E, corresponds to a pair of tape drives, only one of which is active at any one time. Logically, A through E are identical. If a less than five-way pass is specified, input sets are dropped starting with E. Thus, a two-way merge uses input sets A and B.

As in the merge, there are three buffers per input set. They are a "current" buffer, a "next" buffer, and an "open" buffer. A CAC- (Complete Address Constant) type three-way switch is used, and in this case refers to the beginning locations of the three buffers. Switching is accomplished by shifting the switch end around 16 bits.

Index registers X1 through X5 are used to refer to the current item, and R1 through R5 are used as buffer item counters, as in the merge. Since the buffers are stepped through in a forward direction, the index register starts by referring to the first item SWITCH+1 to bypass the banner word. After each item is transferred, AU1 is used to set the index register to the next higher item, and the R counter is incremented.

When the buffer is depleted and switched, the last item (last key) is set up by adding the constant LKEY to the new switch setting. In the case of variable-length items, the last item is found by going through the buffer word-by-word, and looking for and counting end-of-item words. Both the CAC switches and last key settings are kept in a section called TABLE.

Switching the input buffer is done, whenever a buffer is depleted, in the area called SWITCH. To switch a buffer, assuming fixed-sized items (under 64 words), the CAC-type switch is shifted with a mask of all hex G's 16 bits back into itself. The switch plus LKEY is then added through a Word Add (WA) instruction, with a mask of the low-order 16 bits, to find the last key setting. This is kept in the word immediately after the switch. Then the contents of the switch are transferred to the index register, and the address thus referred to (the banner word of the new buffer) is masked and compared with a constant to determine if this is an end-of-file record. If equal, control goes to the appropriate one of five housekeeping routines (A-E HSKEEP) to switch tapes. Otherwise, the index register is incremented once (to refer to the first item), the R register counter is set to unity, and control goes to JJ. JJ, the common routine which corresponds to MERGE+28 in the merge, is used to step the output buffer and determine when it is time to read or write.

The procedure is exactly the same for variable-size items (or those over 63 words) except that the WA is replaced by a sequence change to a subroutine in cosequence. These subroutines, 1V through 5V corresponding to the A through E sets, will set up the last key portion of TABLE by looking for and counting end-of-item words. Each of these subroutines is the same, and works as follows: the index register is incremented, and the address it specifies is mask compared with a constant to see if it is an end-of-item word. If not, it is incremented and compared again. If it is, 1 is added to a working location called POCKET and POCKET is compared with FILE (FILE is a constant set up to equal one less than the number of items per record). If not equal, return is made to increment the index register. If it is equal, a WA adds 1 to the index register into the last key portion of TABLE, and after clearing POCKET back to zero, a return is made to the sequence mode.

The CAC-type switches for the A through E sets are located in TABLE, TABLE+2, TABLE+4, TABLE+6, and TABLE+8. The last key settings are in TABLE+1, TABLE+3, TABLE+5, TABLE+7, and TABLE+9.

Whenever a buffer is written, the output buffer is switched in the JJ section. The two-part switch to accomplish this is in TABLE+10. X6 is the output index register and R6 is the output item counter. R7 is also used as a working register at the time of writing. More detail on

output buffer switching is contained in the subsequent discussion of the merge and read loop, which is part of the section describing the trees used in the collate.

The input buffers must be primed at the beginning of each pass, as well as when an input tape is switched. In the first instance, as in the merge, two buffers of each set are primed, and reading starts into one of the available buffers based on expected depletion. However, when an input tape is switched, only two buffers are primed. In this latter case, a detour is made from the routine simply to check the FID records, switch tapes, and get the new tape started. Return will be made to the main routine when the new routine has replaced, in the buffers, the end FID record of the old tape with the first record of data of the new tape.

Trees

Like the sort routines, collate routines are available for single, double, or triple precision. These routines differ only in the structure of their trees. In each case, the trees used by a collate are logically identical to those used on ascending passes by a merge sort of the same precision. The collate routine can also be modified by means of own-coding to accommodate any additional number of key fields. Section IV contains a description of the structure of these trees.

As in the merge, S0 is used at the exits of the tree for storing the return and going to the appropriate coding to transfer the selected item. As before, this allows the same tree to determine the next input read and the next item to be transferred. One slight difference between the collate and merge tree is that the former increments S0 in multiples of four instead of five, since only four instructions are required to be unique to each input set. S0 is set each time to TRANSFER, which functions exactly as the location MERGE in the merge sort. At TRANSFER, the return to the tree is stored.

When switching to the read mode, the contents of X1 through X5 are stored in HOLD through HOLD+4, and the last key settings (TABLE+1, +3, etc.) are transferred to X1 through X5. S0 is set to READ to refer to the proper set of read instructions. As in the merge, the return in READ will not be used.

When all items being compared are equal, S0 is incremented to refer to a sixth group of instructions, either in TRANSFER or in READ, which will check to see if all inputs are stoppered. If the items are not stoppered, a normal exit is made to one of the other five groups of instructions.

There are five groups of four instructions in the TRANSFER section. In the group

performed when the A item is smallest, for example, the first instruction is a Less-Than (LA) instruction between a working location ASKEY (there are corresponding locations in the other groups, B-ESKEY) and the key of the item addressed through the index register. ASKEY contains the key of the previous item from this input set (an explanation of how this is set up follows later). Assuming single precision, if ASKEY is less than or equal to the current key, then there is no break in sequence, and the item is ready to be transferred to the output buffer. To do this, control is transferred to Continued (CONT) to transfer the item out. The CONT area consists of groups of five instructions each, so the B set uses CONT+5, the C set uses CONT+10, etc. If ASKEY had been greater than the current key, control would have gone to the second of four A instructions in TRANSFER. This, together with the third and fourth instructions, prints SEQ, ERROR, FILE A, and then stops. Instead of going to CONT (or CONT+5, etc.), the double- and triple-precision sequence check comparisons go to a section called SUBTRANS to check the second and third keys. SUBTRANS will exit either back to the print instructions in TRANSFER, or will go to the appropriate section of CONT.

In CONT, the key of the current item is transferred (through the index register) to ASKEY for use the next time around. The item is transferred out by an n-word transfer (item transfer for variable-size items) using index registers 1 and 6. The contents of AU1 are transferred to X1 to set it to the next item. Finally, R1 is checked, incrementing it against FILE+1 (a constant set up to equal the number of items per record), and if it should be unequal, then control is transferred to a common routine, JJ, to complete the main merging loop. Otherwise, control goes to the appropriate area in SWITCH to switch the input buffer.

In double precision, SUBTRANS makes an equality check on the first keys of the last and current item and, if equal, it checks the second key for sequence. ASKEYA through ESKEYA are used to store the second last key. In triple precision, another set of comparisons are made where the third last key is stored in ASKEYB through ESKEYB. In double and triple precision, the CONT area is expanded accordingly to allow the storage of additional last keys. In the event that the user wishes to extend precision beyond triple, he does not have to modify the sequence check accordingly, as it should suffice to check sequence on only the first three keys. Most troubles that a sequence check would isolate revolve around major breaks in sequence. These would probably manifest themselves in the first key comparison.

When the tree is in the read mode, the five groups of four instructions in READ function as follows: the CAC-type switch is shifted 32 bits into working index register X7, a read is performed into the address specified by X7, STOPADD (a SPEC constant) is transferred to the

last key portion of the TABLE area which applies to this input set (this is the address which will reference the stopper word of all hex G's), control goes to the common area GG to restore the index registers X1 through X5 to their normal values, and finally a return is made to the main merging loop.

Multi-precision

Since the collate trees are identical to the merge trees, the discussion of multi-precision in Section IV is applicable here also. There is one additional problem to watch when precision is extended beyond triple through the use of own-coding, and that is when keys, other than the first three, are specified (refer to the final paragraph of Multi-precision, Section IV). Thus, if a fake third key is specified, then the third key comparisons of each of the five sequence checks (in SUBTRANS) will have to be modified to refer to the logical third key.

Main Loop

The main loop of the program, that which is performed for each item processed, has been essentially covered in the preceding sections. Only a few instructions are involved in completing it. In the tree (called LOOP in the collate), the smallest of the items addressed through index registers X1 through X5 is found, a return is stored in the location specified by S0, and S0 is incremented to go to one of the appropriate groups of instructions in TRANSFER. Here (as also in the multi-precision of SUBTRANS) the item is sequence checked and, through the section CONT, is transferred to the output buffer. Also, the input buffer is adjusted and tested to see if it is empty, and if it is, then a sequence change is made to JJ.

In JJ, the contents of AU2 are transferred to X6 to step the output buffer index register to the next item position. R6, the output buffer item counter, is then compared with FILE (items per record minus 1) to determine if it is time to read. If it should not be equal, control goes to JJ+13. In JJ+13, R6 is compared (incrementing it by 1) with FILE+1 (items per record). If it is not equal, control goes to TRANSFER. Here the return to the tree is stored. The collate returns are slightly different than the merge returns in that they make use of the A and B addresses of the return instruction to reset S0 for the next time. (This was done in a separate instruction in the merge.) After resetting S0, then the return leads to the tree to process another item.

Had it been time to read, as determined by an equality in JJ+1, control would have remained in sequence, the contents of X1 through X5 would have been stored in HOLD through HOLD+4, and TABLE+1, +3, +5, +7, and +9 would have been stored in X1 through X5, all

through a series of TX instructions. The index registers are thus set to the read mode, and control goes to FF. This sets S0 to READ and goes to LOOP (top of the tree) in order to initiate a read.

Previously, it was explained that the tree would select the appropriate set to be read and, to do this, it would have gone to a series of instructions in READ. From this point, control would have gone to a common section, GG, to restore the index registers, and then from there to JJ+13 to get back into the main loop.

When it is time to write, as determined by an equality in JJ+13, sequence control remains the same. In JJ+14, 1 is added to a counter called ACMLATE, which is the output record counter. ACMLATE is then compared with PAR+2, the portion of the parameter which specifies the number of records per tape to be written by the collate. If these are equal, the tape is full and control goes to EOPT to change output tapes. If not equal, X6 (now set to the word beyond the last item) is transferred to working register R7, and TABLE+10 (the output buffer switch) is transferred to X6. The contents of ACMLATE are mask transferred into the banner word of the output buffer, as specified by X6. Ortho is computed from the word specified by X6 to the word specified by R7, and writing starts from 6, 0. TABLE+10 is then shifted 24 bits end around into itself to switch output buffers, and this plus 1 (to get by the bannnr word) is stored in X6. R6 is set to unity, and control goes to the return in TRANSFER.

The main loop and the read in the collate, except for the resetting of S0, are very similar to the corresponding portions of the merge. The writing portion of the collate is considerably simpler because of the lack of beginning-of-string markers and the corresponding need for a banner switch.

Input Buffer Switching

After selecting the smallest item and transferring it out, as already described in this section under Trees, the input buffer is stepped to the next item. If, in this process, the input buffer is found empty, control goes to the appropriate section of SWITCH, depending upon which input set is being handled. All five of these sections are similar. SWITCH corresponds to the Beginning-of-String (BOS) check in the merge, except that here it is not necessary to check for beginning of string.

The details of SWITCH are covered in this section under Buffers. In general, the buffer switch (TABLE+1, +3, +5, +7, or +9) is set up. The procedure for this is somewhat involved if

the item size is variable. The index register is reset to the beginning of the new buffer, and the banner word is checked to see if this is an end-of-file record. If not, control goes to JJ, as would have been done if the buffer had not been depleted. Otherwise, if this is an end-of-file record, control goes to A through EHSKEEP to switch input tapes. This process corresponds to going to EOPT when an output tape is filled, although the latter is sensed by an internal counter or by hitting the physical end of tape, rather than by sensing the end FID record in the data. If the physical end of an input tape is reached, the resulting unprogrammed transfer is bypassed, since there must always be an end FID record at the end of each tape.

End of Output

This section of the program, called EOPT, may be reached in either of two ways. A parameter-supplied limit of the number of records to be written on each output tape allows switching outputs as soon as that many records have been written. This is detected in JJ when the record counter is incremented and compared against the constant PAR+2. If, on the other hand, the output tapes are to be filled to capacity, an infinitely large number of hex G's may be supplied through the parameters, in which case the equality of JJ will never be met. An end-of-tape unprogrammed transfer will be made, however, when the physical end of tape is reached. This unprogrammed transfer results from writing an output record. This write will be correctly initiated, and the unprogrammed transfer then leads to EOPT, just as if the record counter had been found equal to the parameter-supplied limit. Actually, the record counter leads to EOPT only during the last pass because a count of hex G's is arbitrarily used during intermediate passes to fill tapes to their maximum capacity.

In EOPT, X7 is set to the internally stored FID reserve area, and the end FID banner word is transferred to 7,0. Ortho is computed from 7,0 to 7,9 (the size of the special ID records), at which time control goes in cosequence to the write instruction in Write First Begin ID (W1STBID). This is a write instruction especially set up to write ID records which, like the write instruction in JJ, uses X7 rather than X6. The instruction following the write instruction in W1STBID drops control from the cosequence mode, and also transfers plus zero to MEMLOC. (This latter step has no effect at this time.) Back in EOPT, an end-of-information banner word is stored into the beginning of IDRES through X7. Ortho is then computed, and again control goes to the above write instruction in cosequence. Again back in EOPT, a rewind order (with interlock) is set up this time with the tape address of the current output tape (found in DRIVE+5). It is performed to initiate the rewinding of the tape just completed.

Following this, "END OPUT, REMOVE TAPE ON" is printed. The tape address is

substituted from DRIVE+5 into the working print location MEMLOC+3, and this is printed in octal. TAPE is then printed, as are contents of KEEP+10 and KEEP+11, which give the file and segment numbers assigned to the tape.

At this time, the just-completed tape is removed, and the next one is started. DRIVE+5 is shifted left, 24 bits end around, to switch the tape drive. The new tape address in the leftmost portion of DRIVE+5 is substituted into JJ+21 and into the write instruction in W1STBID. Decimally, 1 is added to the segment name in IDRES (through 7, 2) and a beginning FID is transferred to 7, 0. The new tape is read into stopper. Again, a compute orthocount is performed from 7, 0 to 7, 9, the new segment name is also transferred from 7, 2 to KEEP+11, and finally control goes to the write instruction in W1STBID in cosequence to write the beginning FID on the new output tape. Note that if this is on the same drive as the tape just written, or if a new tape has not yet been mounted since the last tape was written on this drive, stall is initiated trying to read because of the rewind with interlock. Once the new tape is mounted, however, the read, and then write, will proceed correctly.

Back again in EOPT, having bypassed the tape ID record and written the beginning-of-file ID record, control goes to an exit switch. This is normally set to clear ACMLATE to zero, and go to JJ+17, which assumes that control was directed here from the counter comparison in JJ+15. In this case, the output buffer is still full, and control returns to JJ at the point where it originally branched off just before writing the buffer. If, on the other hand, control had gone here from the unprogrammed transfer area, the buffer would already have been written. Thus, the unprogrammed transfer instruction sets ACMLATE to zero and leads to another TS instruction. This second TS instruction sets a special switch setting at the end of EOPT which will replace itself with the normal setting (described above) and go to JJ+22. This will lead back to JJ immediately after the write instruction, which effectively again is the point where control was originally transferred (through the unprogrammed transfer register) from JJ.

End of Input

In SWITCH, after the input buffers were switched, it will be recalled a check was made to see if the new record is an end FID. If it is, control goes to AHSKEEP through EHSKEEP, depending upon the input set which is presently being worked. Since these five areas are similar, it will suffice to describe just AHSKEEP. Actually much of the processing necessary at this time is common to all five of the input sets, and so AHSKEEP through EHSKEEP serve only to perform those instructions unique to each set, and to set up several common locations. From each of these, control goes to Master Housekeeping (MHSKEEP) in order to determine if an

end of segment or an end of file is reached, and also to check the new input tape (or, if an end of file, to stopper this input set).

AHSKEEP begins by setting up common locations. The A input drive switch (DRIVE) is transferred into WL1 (the other sets up DRIVE+1 through DRIVE+4). TABLE, the input buffer switch, is transferred to WL1+1 (TABLE+2, +4, +6, and +8 for the other inputs). The address of KEEP, which gives the file and segment name, is stored in Z, S1 (likewise KEEP+2, +4, +6, and +8). The address of the read instruction, READ+2 (+6, +10, +14, and +18) is stored in Z, R7. The input drive switch is then switched end around 24 bits into itself, thereby switching input drives (DRIVE, and DRIVE+1 through DRIVE+4). The input item counter, R1 (R2 through R5) is set to +1, and control goes to MHSKEEP.

In MHSKEEP, the leftmost tape address from WL1 (the address of the tape just depleted) is substituted into WL1+2 (zeros). Using this, a rewind instruction is set up and performed with interlock, thus rewinding the input tape just depleted. Then the input drive switch in WL1 is shifted end around 24 bits, so that it also is switched, and its leftmost (new tape) address is substituted into the read instruction through N, R7. REMOVE is printed and, through N, S1, 1, the file name is also. The two low-order digits of the segment name, obtained by substituting incremented N, S1 into WL+3, are printed, as is TAPE ON and the drive address (from WL+2). Now the address of the end FID, still in the input buffer, is stored in Z, X7 (from the buffer switch in WL+1) so that it can be interrogated. Next, the low-order two digits of 7, 2, the segment name word, are compared against a constant of hex G's.

Assume for purposes of explanation that the low-order two digits were not hex G's, meaning there is more of this file on another reel; then END SEG would be printed out, and 1 would be added to N, S1 (segment name in KEEP). A read instruction is then set up and performed to bypass the tape ID of the new tape (into stopper). As was true of the output tape, this will cause a stall if both drives of this set are the same, or if the new tape has not yet been mounted. If the new tape is mounted, the tape ID record is bypassed, and control goes in cosequence to N, R7. This was previously set up as the new read instruction, which will bring the beginning FID record of the new tape into the input buffer just occupied by the end FID of the old tape. The instruction after each read in READ is a void TX instruction, which will return to MHSKEEP. Here a dummy read (void A address) is set up and performed to insure that the record just read is in memory. A check is also made of 7, 0 to see if this is a proper beginning FID banner word. If not, control goes to the error routine covered in the next paragraph.

If this was the proper banner word, Z, S1 would have been reset one word backward to reference the file name word once again, and 7, 1 would have been checked against it (incrementing S1 by 1 again). This matches the new file name against that in KEEP which corresponds to this input set. If unequal, control goes to the error routine covered below. If the file name is correct, the low-order two digits of 7, 2, the segment name, are checked against N, S1, and again, if these do not agree, control goes to the error routine. This common error routine prints X TAPEON (for wrong tape on), and substituting the drive address from WL1, prints that in octal. S1 is again backed up one word, and used to print the file name and the low-order two digits of the segment name. CORRECT TAPE is then printed, and a rewind is set up and performed with interlock. Having done this, a sequence change is made back to the point in MHSKEEP where the tape ID record is read and at this point a stall is initiated until another tape (presumably the correct one) is mounted.

If all tests for a new tape had been passed, control would go in cosequence again to the normal read instruction, thus bringing in the first data record over the FID record just checked. WL+1 is shifted one position into X7 and the read is again performed in cosequence, thus starting the read of the next record of data into the next input buffer to insure the first record of data is in. At this point, the end FID record originally discovered has been replaced by the first record of data from the next tape, and control returns to JJ as if no ID record had ever been discovered.

Thus far, it has been assumed that the end FID record discovered in SWITCH was not the final end of the file, but that there was more information on another tape to follow. Consider now the case where the end FID record was a final end, with hex G's as the low-order two digits of the segment number. This fact is discovered in the first comparison of MHSKEEP. In this case, it is not necessary to examine the new tape, but only to stopper this input set.

First, END FILE is printed (this will follow the printouts calling for the removal of the final tape from the specified drive) and control will go to N, SH. At this point, there has not been a sequence change since MHSKEEP was reached from AHSKEEP (or B through EHSKEEP), so this device will return to a special set of instructions at the end of each of these sections. In the case of AHSKEEP, these will transfer STOPADD (the stopper base address) to TABLE+1 (+3, +5, +7, +9) to stopper the last key section of this input set. STOPADD will also be transferred to Z, X1 (X2, 3, 4, 5) to stopper the current item of the set, at which time control returns to JJ.

The following summary applies to what happens in the A through EHSKEEP areas, as well as the MHSKEEP area. These areas are reached upon discovering an end FID when the input buffers are switched in SWITCH. Finding such a record causes the removal information to be printed on the typewriter, the tape to be rewound, and the alternate input drive to be switched on. If this should be an intermediate end FID record (no hex G's in segment number), then the new tape on the alternate drive would be checked, and if it should yield a correct indication, the buffers would then be primed and the operation would proceed. If the new tape is not the correct one, information must be printed on the typewriter telling which one should be mounted, and preparation should be made to check the ID record again. However, if the old end FID was the final one of this input set (hex G's in segment number), this input set would be stoppered and the operation would proceed.

All Items Equal

The end of a collate pass is determined by the discovery that all current input items are stoppered. This will happen after each of the inputs (in turn) has reached a final end FID record, as described in the preceding paragraphs. As in the merge, when all keys being compared by the tree are equal, S0 is incremented to a special sixth group of instructions which exist in both the read and merge modes.

In the read mode, this special group of instructions compares each index register, X5 through X1 in turn, with STOPADD. The first comparison yielding a not equal result will lead to the normal instructions in READ for processing the corresponding input set. If all index registers are equal to STOPADD, control goes directly to GG (where control would have gone in any case after the instructions in READ had been performed), thus skipping any actual reading at this time.

In the merge or transfer mode, much the same procedures are followed. That is, X5 through X1 are compared with STOPADD, and the one not equal will lead to the corresponding instructions in TRANSFER. If all items are stoppered, control goes to ENDPASS to finish the output tape and initiate the setup for the next pass.

In ENDPASS, X7 is set to IDRES, the common FID record in memory, also an end FID banner word is set up in 7, 0, and the low-order two digits are set up as hex G's in 7, 2 (segment name). Ortho is computed from 7, 0 to 7, 9 the drive address is set up in the write instruction in W1STBID, and control goes there in cosequence, thus writing the end file ID on the final output tape. Likewise, an end-of-information record is set up and written, and a rewind of the final output tape is then set up and performed with interlock. Tape removal information relative

to this tape is printed. END PASS is printed, the Pass Count (PSCOUNT) is incremented and printed. Finally, BEGPASS is printed and also the contents of PSCOUNT plus 1, which signify the beginning of the next pass. (At the time the final pass of the collate is set up, this final set of print instructions is replaced by instructions which end the collate.) From here, control goes to MODIFY, where the collate is regenerated to set up for the next pass, based upon the next entry in the plan.

Regeneration of the Collate

Each pass of the collate is a complete logical entity. It is quite possible that the very first pass will be somewhat different from the others (the "way" being less) because of the particular plan calculated. Each pass involves a different set of file and segment names to be stored, based upon the plan, and also the routine must be completely initialized for each pass. Because of the considerations, the collate is partially regenerated between each pass. A general description of the collate is therefore described in the following paragraphs in so much as the generating process may occur a number of times during the course of the program. Also, a portion of the generator, which is repeated, includes the priming of buffers and the writing of the beginning FID record on the first output tape.

Generation of the Collate

The initial portion of the generator, STARTUP through CALCULATE, calculates, stores, and prints the plan. These sections will not be repeated. Following this, MODIFY and SETPATH are entered and repeated for each pass. They determine the "way" of the pass, and set up the core routine accordingly. At this time a switch is reached which leads to GENERATE the very first time through, but which bypasses GENERATE all times after that in order to go to AA. GENERATE sets up the tape address in the DRIVE area, based upon the parameters in the macrocoding, as well as setting up some peripheral instructions in the core (skeleton) program. After GENERATE, BEGIN interprets the parameter from the FID record on tape. A1, A2, and A3 calculate buffer size, set up variable-size items, modify the sequence check, modify the TN instructions to output, as well as modifying tree comparisons, stopper base address, etc. This represents the bulk of the process normally considered generation. After the first pass is set up, all of the coding mentioned above (except MODIFY, SETPATH, and the plan) is clobbered by the buffers, since it will never be performed again. From here (as well as from MODIFY after the first pass), control goes to AA.

In AA, BB, BBB, and CCC (which, together with the sections of coding following, are performed before every pass), the buffer areas are set up, and the buffer switches in TABLE are

built up accordingly. These sections are performed each time because of the possible discrepancy between first and subsequent passes (as the "way" increases, the buffer area must be expanded). From here control goes to LOAD and W1STBID. These sections check all of the input FID's, store the file and segment names in KEEP, write the beginning FID record on the output tape, and check to determine if this is the last pass. If it is, special last pass modifications are made to the routine.

Following this, CC performs two priming reads for each input tape. EE sets up the last key areas, and sets the tree to the read mode to do an extra read based on expected depletion. Input and output counters, as well as the initial sequence check key storage areas, are initialized. Finally, control goes to the tree (LOOP) to begin the actual pass.

Over-all Flow of the ARGUS Collate

Thus far, the over-all collate has been discussed in general terms, and various components of the collate have been explained in detail. These components are tied together in the following paragraphs to present a complete collate picture. A collate flow chart is shown in Figure 17.

The very first part of the program to be performed, STARTUP through CALCULATE, creates the plan and stores it adjacent to the program in memory. At this point, the option exists to print the plan on the console typewriter or to proceed directly with the program. If the plan is printed, there is a stop point afterwards to allow termination or continuation of the collate. These options are intended to allow the plan to be printed at one time (by stopping immediately after) and later to perform the actual collate (by starting over again, but bypassing the printing), or to allow printing the plan and immediately performing the collate.

In any event, from CALCULATE, control goes to MODIFY and SETPATH to determine and set up the "way". These portions of the generator will be performed at the beginning of every pass. Upon completing SETPATH (if this is the start of the first pass), control goes to GENERATE to perform the bulk of the generation of the collate. After the first pass, control goes from SETPATH directly to AA. The sections, GENERATE through A3, perform all once-only generation, and they also exit to section AA.

Sections AA through EE once again are performed at the beginning of each pass. Here, the buffers are set up, the input tapes are checked, and the file and segment names found on these tapes are stored. Also, the beginning FID record on the initial output tape of this pass is

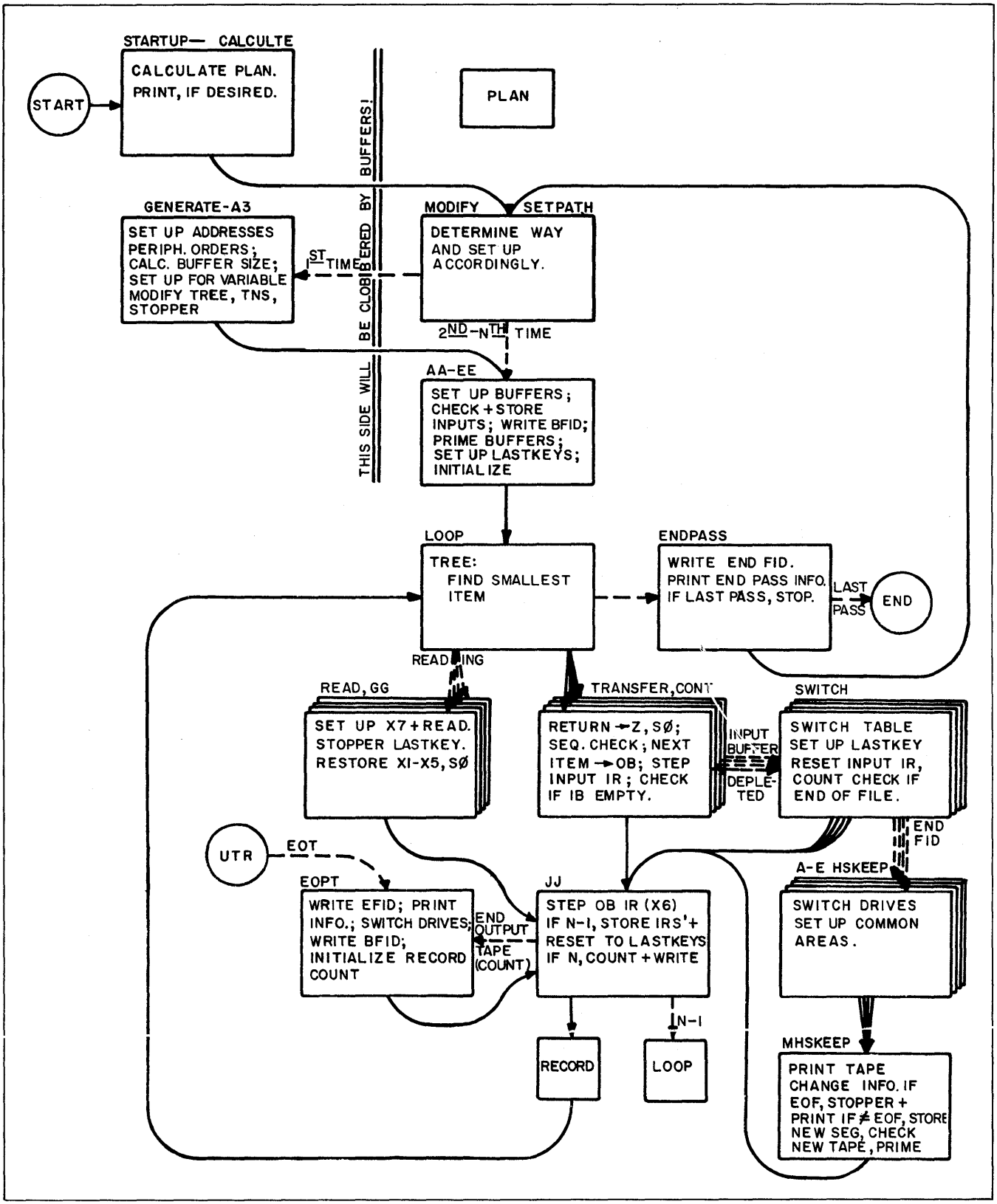


Figure 17. Over-all Flow Chart of the ARGUS Collate

written, the routine is set up specially if this is to be the final pass of the collate, and finally the buffers are primed and the routine is initialized and prepared to do one extra read based upon expected depletion.

At this point, as indicated in the flow chart, Figure 17, the sections STARTUP through CALCULATE and GENERATE through A3 have been clobbered by buffers. From this point on, only the sections MODIFY, SETPATH, and AA through EE, as well as the previously stored plan, will be used. In the tree (LOOP), one extra read is initiated, and then the tree is restored to normal operation, and since the output buffer is not yet filled, control goes to the return. The return is initially set to the top of the tree.

Now the main loop of the program begins. In LOOP, a comparison is made for the next item, and accordingly control goes to the appropriate section of TRANSFER and CONT. Here the return to the tree is stored. The item is sequence checked and then transferred to the output buffer. The input buffer thus affected is stepped to the next item position. Then, in JJ, the output buffer is similarly stepped and tested to see if it is full but for one item or to see if it is completely full. If neither of these conditions exist, then control goes to the stored return, which leads back to LOOP to process another item.

When the output buffer is full but for one item, it is time to read. Again in JJ, the contents of the index registers are stored and the index registers are set to the last key values. S0 is set to READ and control goes to LOOP to initiate a read into the input set which would otherwise run out first. LOOP determines which set this is and goes to the corresponding section of READ. Here, a working read index register, X7, is set up and the read instruction is performed. The last key location of this input set is stoppered, and the index registers are restored to their normal values. This latter action occurs in section GG, which then leads back to JJ, and eventually to the return location to process another item.

When the output buffer is full, as discovered in JJ, the record count is stepped and then written, and the output buffer is switched and then reset. During the last pass, if the number of records specified had been written on the output tape, control would go to EOPT to switch output tapes. If not, then control would go to the return location to process another item.

Eventually, one or another of the input buffers will become depleted, as will be discovered in CONT when the buffer is stepped to the next item. In such a case, a detour is made to SWITCH, where the input buffers are switched around, the last key location is set up to its new

value, and the index registers, as well as the R register associated with this input, are reset. A check is also made to see if the new input record is an end FID. If it is not, control goes on to JJ, as would normally be done from CONT; if it is however, control goes to the appropriate HSKEEP area for this input set.

Eventually, an output tape will be filled, and the corresponding unprogrammed transfer register will lead to EOPT (unless the counter in JJ did so instead during the last pass). Here, the end FID is written on the filled output tape, at which time the tape is rewound; here also, the removal information pertaining to it is printed, drives are switched, and a beginning FID record is written on the next output tape. From here, control returns to JJ at the point which was vacated to go to EOPT.

When an input tape is depleted, as discovered in SWITCH, control goes to one of the HSKEEP programs (A through E). Here, input drives are switched, a few common locations are set up, and control goes to MHSKEEP. The depleted input tape is rewound, and removal information is printed. If this was the last tape of this file, the input set would be stoppered, and return would be made to the normal flow in JJ. If there is more to follow in this file, however, the new input tape is checked and the buffers primed with the first records of data from it. These will replace the end FID record originally discovered, so return can be made to JJ to continue in the normal flow.

When all input have reached final end and have been stoppered, LOOP will sense that all keys are stoppered and lead to ENDPASS. Here, the final end FID record is written on the current output tape, and ENDPASS is printed. If this should be the final pass, the collate ends here. If, however, there are more passes, the fact that a new pass is beginning is printed out, and control goes back to MODIFY to set up for it.

In the process of interpreting the plan, MODIFY and other associated routines step the plan down one entry each time, so that this next pass will be set up according to the next entry in the plan. Finally, when it is discovered that tape number 0001 is being written, this will be assumed to be the final pass and the routine will be set up accordingly.

Error Correction and Restarts

The collate, like the presort and merge, makes use of the orthotronic correction provisions of the Executive Routine to detect and repair any read errors and to document such actions at the console. In the event of a read error, the location of the suspected record is

determined, and control is turned over to the Executive Routine to try and repair it. If unsuccessful, the information is reread by the collate, and if still determined incorrect, control is again turned over to the Executive Routine. This process is repeated several times. Suitable printouts at the console indicate the nature and disposition of the trouble.

If the physical end of tape is reached while reading, the unprogrammed transfer is ignored. If reached while writing, the output tape involved is completed and tapes are switched. Since all reading and writing is in the forward direction, there should never be an unprogrammed transfer caused by reaching the physical beginning of tape.

Restarting is implemented in much the same manner as it is in the merge, although in the collate restarting is much more versatile. The general theory of restarting used by the collate is explained here. Restart dumps, or anchor points, are placed on the tape label record of each output tape. In order to re-create any tape, that tape should be mounted so that the dump may be read into memory. At the time it is written, the memory dump is checked for correct parity by reading it back into memory. In the dumping process, all of the special registers are transferred to memory locations contiguous with the program, as are the banner words from the current input buffers. Then, one long record, including the plan, program, special registers, and banner words (but not the buffers) is written onto tape. In the event of a restart, the dump is read back into the same area, and the special registers are eventually restored. To position, the input tapes are read into the buffer area, where their banner words are compared with the stored ones from the restart dump.

Therefore, the restart, or anchor points, are established just prior to beginning a new output tape, so that return can be made to that point at a later time to re-create that tape. There are two areas in the collate from which a detour can be made to the restart dump routine (the routine which creates these anchor points). These two areas are found: first, in AA through EE when the first tape of a pass is started; second, in EOPT when any additional tapes of a pass are started.

The restart dump routine writes the current status of memory on a tape specified by the user, in such a form that it can be associated with the tape about to be written. To do this, all of the special registers, and the banner word from each of the current input buffers, are stored in a block of memory adjacent to the program. This contiguous portion of a program (including a plan), special registers, and banner words make up one long record and this record is written on the dump tape. Following the dump portion, the program is continued at the point originally

vacated. If restarting during the collate is not necessary, then this will be the only portion of the restart program performed. Suppose, however, that while writing the fifth tape of pass number seven, it is discovered that one of the inputs cannot be read. Assume this was the third tape produced during the second pass. Obviously, this tape must be re-created before continuing. Thus, a restart is initiated (by starting at R0). This is a small program logically independent from the collate routine itself. Before doing anything, the restart program asks that the tape which is to be re-created, be specified. The file name and segment number of the tape are typed in by the operator, and the restart routine begins searching the dump tape for a dump with that identification. When the dump is found, it is read into memory over the exact locations from which it was written, that is, the program (including the plan), the special registers, and the banner words.

Inspecting the KEEP area of the program just brought in, the restart program then indicates the tape number of each input tape mounted when the dump was taken. After indicating these on the typewriter, it comes to a halt. The operator then mounts these backup tapes onto the same drive on which they were originally mounted (also indicated by the restart program, based on the settings in DRIVE). Again the restart program is started, and it checks each of the beginning FID records of the tapes just mounted for the correct file name and segment number. If these are good, each tape in turn is positioned, comparing the banner word from each record with the corresponding one brought in from the dump.

This paragraph considers the positioning of tapes in more detail. The restart program will first check each of the five index registers, X1 through X5, and if any of these are set to stopper, then the corresponding tape is inactive and not needed. (This is due to a limited "way" pass, or when reaching a final end of file.) Then each active tape is positioned, reading directly into the "current" buffer until the correct record is brought in. (In order to insure each record is in, the active and dummy read instructions are alternated. In searching, this will not be a slowing factor.) Another read is started into the "next" buffer, and then the last key location of this set is checked. If this last key position is stoppered, an additional read has been started and a read is initiated into the "open" buffer also. Otherwise, if the last key location is not stoppered, only two buffers are primed.

When the input tapes are all positioned (and the input buffers filled), it is about the time to re-enter the collate. The special registers are now set to the values stored for them in the dump. Using one of the history registers, return can be made to the point in the routine where a return was originally made just after making the dump. The tapes and memory should now be exactly as they were then, so it is necessary to re-create the output tape exactly as had been done before.

If desired, the collate could be continued from this point forward. However, in this example mentioned above, it was required that only the third tape written during the second pass be reproduced. Once this is completed, it is possible to go back to the fifth tape of the seventh pass where trouble was encountered during the writing process. To do so, a restart is performed in exactly the same manner as explained above, this time specifying that the fifth tape of pass seven be re-created. The tape just re-created will be called for as one of the inputs. Using this in place of the illegible one, the operation should be able to proceed without any further trouble.

SECTION VI
OWN-CODING

The ARGUS sort routines are designed to handle a wide variety of requirements. Nevertheless the need may occasionally arise to sort items whose specifications exceed the parameter limitations, or to combine the first or last pass of the sort with some simple item processing (editing). It is impractical to provide for every such special case in a sort generator, or to write a complete sort routine for each of these variations. ARGUS provides for all such special cases by allowing the programmer to write the additional coding necessary to accomplish them. This coding may either be associated with a standard generated sort, or if necessary, it may actually be used to modify the generated sort itself. The ARGUS sorts have been built to incorporate optional detours at specified points to facilitate the tie-in of just such coding. With this technique, appropriately referred to as own-coding, there is virtually no limitation to the modifications which can be made.

Own coding may be divided into two general categories:

1. Data modification; and
2. Routine modification.

Data modification (the simpler type) involves any changes to the data which is being sorted. This includes rearrangement or translation of keys, batch totalling, addition or deletion of items, and modification of item size. Also included in this category are any changes to the beginning-of-file identification record, such as modification of parameters specifying item size, key location, etc. Data modification own-coding is covered in this section in detail, with appropriate examples.

Routine modification own-coding involves changes to the sort routine itself, and requires a more complete knowledge of the ARGUS sort generators plus a complete and accurate listing of the routines. Such own-coding might be used to modify a sort to read directly from a card reader, to provide output directly to a line printer, or to handle extended precision. The general methods by which this type of own-coding may be implemented are described in this section.

Own-Coding (Edit) Options in the Sorts

The presort, merge sort, and collate each have several own-coding options, which may be specified in the pseudo instruction the programmer writes to call a sort. In each case, 00

specifies no derails, that is, the sort is to run by itself with no own-coding modifications.

There is provision for four presort options (01, 02, 03, or 04) which are specified by writing one of these numbers in the sort pseudo instruction.

Option 01: specifies a single detour immediately after the beginning-of-file ID is read and checked by the presort generator. This allows modification or complete replacement of the file ID record.

Option 02: specifies a detour immediately after the presort has been generated, but before any data has been handled by the sort. This allows any type of modification to the generated presort itself.

Option 03: is used for data modification and specifies a detour immediately before each item is transferred from the input buffer. It allows changes to be made to each item before it is used by the sort.

Option 04: specifies that each of the options 01, 02, and 03 will be observed; hence, all three detours will be taken.

The merge sort has provisions for options 02, 03, and 04, specified by writing one of these numbers in the sort pseudo instruction. Option 01 is not available since the parameters specified in the beginning-of-file ID record are transferred directly from the presort to the merge sort.

Option 02: allows modification of the merge sort itself immediately after it is generated but before any sorting takes place.

Option 03: specifies a detour only during the last pass of the sort, each time an item has been placed in the output buffer. This allows data modification on the same scale as performed in option 03 of the presort.

Option 04: specifies the use of both detours for options 02 and 03.

It should be noted that any merge sort own-coding must be written as a unique segment separate from the sort segment itself. The name of this own-coding segment is specified in the sort pseudo instruction.

The pseudo instruction, which is used to execute the collate, allows options 01, 02, 03, and 04.

Option 01: is identical to the presort option 01. The beginning-of-file ID record, used by the collate as the source of data parameters, is taken from the first input tape of the first pass.

Option 02: is identical to the 02 option of both the presort and merge sort. In the collate, this option is performed immediately after all of the "once-only" generation has taken place.

Option 03: is like the merge sort option 03 in that it specifies a derail for each item during the final pass only. This derail is performed immediately after the item is placed in the output buffer.

Option 04: specifies that all three of the options 01, 02, and 03 are performed.

General Technique

Before beginning a detailed discussion of the methods of own-coding, it would be well to review some of the general problems involved. Since the sort itself is a subroutine, it exists on the symbolic program tape, not in ARGUS tag notation, but in binary relocatable form. Own-coding is normally written in ARGUS language. However, the own-coding may not make use of the symbolic tags originally used in coding the sorts because it is assembled independently from the sorts. Furthermore, since the sorts occupy a full bank of memory by themselves, the own-coding must of necessity be located in another bank. Communication between the sorts and own-coding, therefore, cannot be through the use of tags, nor any form of direct addressing. Quite obviously, special registers must be used.

S2 is reserved by the sorts as the commumicator between the sorts and own-coding. If own-coding is used, S2 must be directly loaded by the programmer with the address of the entrance to the own-coding. Thus, any detour from the sort will be in the form of a transfer to N, S2. This transfer will always specify the cosequence mode, implying that if own-coding is written entirely in the cosequence mode, a return to the sort is made simply by reverting to the sequence counter. It should be noted, however, that the bank indicator of the cosequence counter must be restored to the sort's bank. This may be done by transferring the contents of the sequence counter to the cosequence counter before returning. Alternatively, the contents of the sequence counter may be stored by own-coding, allowing it to use the sequence counter also. In this case, a return to the sort is effected by restoring the sequence counter and reverting to the sequence mode.

Communication between simple forms of own-coding and the sort is further aided by use of other special registers. For example, the location of the beginning FID record or of an item in a buffer is always specified by a certain index register. At times, the sorts use every available special register (save S4 through S7), requiring that own-coding store and restore any registers that are to be disturbed for its own use. Alternatively, the own-coding could use another special register group although communication with the sort must be through the sort's group.

If own-coding should be the routine modification type, then communication is more difficult than with data modification. At the time the option 02 detour is made from the sort, the sequence counter is set to a known location at the beginning of the particular sort routine. Any location in the sort may be addressed by using the contents of the sequence counter as a base, and adding fixed quantities (which may be in the form of FXBIN constants) to this value to increment to the desired address. S2 may be used to store the new address, since it has already

served its purpose in getting from the sort to own-coding. Thus, if it should be required to change a location in the sort which is 613 words beyond the current setting of the sequence counter, and if there should be a constant SXTHRTN FXBIN 613, it would be necessary to perform a word add (WA) instruction (WA Z,SC SXTHRTN Z,S2) to get the address of the location of interest into S2. Then, anything could be done with the word addressed as N,S2.

Since S2 is used as the detour communication point for all the options, own-coding must set it up, not only for the first detour point, but also for each successive one. Thus, if option 04 is specified, S2 must initially be set to the entrance of the option 01 own-coding (except in the case of the merge sort). After this part of own-coding is finished, but before returning to the sort, S2 must be set to the entrance of option 02 own-coding. Likewise, this portion of own-coding must set S2 to the entrance of the option 03 own-coding before returning control to the sort (where it should remain set for the duration of the sort). If any detour is not needed under option 04, then the corresponding portion of own-coding would merely consist of a single word that resets S2 and returns control to the sort. It should be noted that a detour is performed only once for each option except 03, but also noted, however, that option 03 detours for each item in the file.

Thus, it has been established in this section that detours exit from the sort at specified points throughout the routine to perform additional instructions, or own-coding. It should be noted, however, that it is also possible to perform additional coding either before or after the sort, accomplishing such functions as tape positioning, splitting output, etc. Although this additional coding is not exactly own-coding as it has been defined for using detours from within the sort, these techniques can be used to accomplish similar objectives to the own-coding options, and are thus included in this discussion.

Relocation and Bank Assignments

Since they may expand "down-memory" only, the sorts should be loaded into the highest possible bank in a given system. Although they are normally contained within that bank, the user may specify that additional memory is available beyond one bank in any amount he wishes. The sort will correspondingly expand the storage or buffer area over and above the previous bank, or banks, if it can use the space.

The MMMM field of the sort pseudo instruction indicates to the generators how much memory the sort may use beyond the one bank it occupies. To properly relocate the sort and to reserve the appropriate amount of memory, the programmer should specify at least one

SETLOC preceding the sort pseudo instruction. If no own-coding is given for the sort, one SETLOC will determine where the sort routine, along with the sort macrocoding, will be located. However, when own-coding is used, a SETLOC, if desired for the sort routine itself, must be preceded within that segment by another SETLOC which will locate, relative to the sort, the macrocoding and any further coding the programmer writes. If only one SETLOC is given with the programmer coding, the sort will be located in the succeeding bank. The MMMM memory specification in the pseudo instruction, and the A field of the SETLOC instruction, are related. Starting with location 0000 of a bank, MMMM specifies the number of locations over which the sort routine will (or may) expand backward. Depending upon the number of words of programmer coding, and the location specified by the SETLOC, the highest location used by the programmer, as well as the amount of memory available to the sort routine for storage beyond its own bank, can be readily determined. The MMMM field and the SETLOC should be so established that the sort and programmer coding do not conflict. An exception would be the own-coding of option-type 01 or 02; then own-coding and the MMMM area can overlap, since own-coding would be completed by the time the MMMM area is used to store data. In determining the number of locations used by his coding, the programmer should bear in mind that pseudo instructions will require various amounts of macrocoding. Specifically, L,SORTp requires 15 words, L,READSEG three, and L,EXIT a single word.

The sort is contained between locations 0020 and 2043, within one bank, leaving 20 unused words at the bottom and four at the top of the bank. The latter conforms to the stopper requirements of the Executive Routine, allowing the sort to be relocated to the highest bank of a particular machine. The initial 20 words are enough to allow the inclusion of the L,SORTp macrocoding plus the L,READSEG or L,EXIT macrocoding. Thus, to facilitate parallel processing, a sort program may be contained entirely within one bank, when additional memory (MMMM) equals 0000 and own-coding and pre- or post-coding are not requested. In this case, the programmer would specify a SETLOC of 0000 in any bank he desires above bank 0. If no SETLOC is provided by the programmer, ARGUS Assembly will assume bank 0, location 0512, group 1 for the programmer coding, and the sort would be placed in the next higher bank (normally bank 1), group 1.

The programmer of own-coding will want to use masks many times. Because the sort uses the Mask Index Register (MXR), this can present a problem. There are several possibilities, however. First, the own-coding may store and restore the contents of MXR each time own-coding is entered. Or secondly, certain masked operations can be performed using a few extra instructions, by means of the substitute (SS) and extract (EX) instructions, which do not

require the use of the MXR. Thirdly, with a list of the sort coding, the programmer could use the sort routine's masks. Fourth, and finally, own-coding could operate in a separate group, using its own MXR. The first two possibilities are the most practical, since the third one requires the programmer to write masked instructions in the form of decimal constants to specify the MXR augment because the sort's tags are unavailable. The fourth method involves the use of the Program Control Register (PCR) to turn groups off and on, and this is accompanied by the associated problems of control in parallel processing. However, all methods are feasible.

ARGUS Techniques for Own-Coding

It is appropriate first to review the method of writing the simplest type of sort call, in which no own-coding is to be performed.

ARGUS CODING FORM

PROBLEM _____ PROGRAMMER _____ DATE _____ PAGE _____ OF _____

	LOCATION (1–10)	COMMAND CODE (11–22)	S/C (24)	A ADDRESS (24–37)	B ADDRESS (38–51)	C ADDRESS (52–65)	REMARKS (66–80)
1		PROGRAM		SORTFILE	ONLYSEG		
2		SETLOC		ØØØØ	B1	G1	
3	Z,SC	SPEC		-	-	ENTER	
4	ENTER	L,SORT1,S		ØØ/ØØ/	ØØØØ/AB/AB/AC	AD/GG/GG/GG/AB	
5		L,EXIT					
6		END		SORTFILE	ONLYSEG		

In this example, program SORTFILE is nothing more than a sort. The SETLOC specifies that the sort macrocoding will be included with the sort in bank 1, and thus occupy a minimum amount of space. Whatever special register group is specified in the SETLOC will be used by the sort as well as by the additional coding. No extra memory is available to the sort (thus the 0000 in the first field of the B address of the pseudo instruction).

The sort may also exist without own-coding as one segment of a program as such:

ARGUS CODING FORM

PROBLEM _____ PROGRAMMER _____ DATE _____ PAGE _____ OF _____

	LOCATION (1–10)	COMMAND CODE (11–22)	S/C (24)	A ADDRESS (24–37)	B ADDRESS (38–51)	C ADDRESS (52–65)	REMARKS (66–80)
1				(PRECEDING SEGMENT(S))			
2		L,READSEG		SORTFILE	ENTER		
3		SEGMENT		DAILYRUN	SORTFILE		
4		SETLOC		ØØØØ	B1	G1	
5	L,ENTER	L,SORT1,S		ØØ/ØØ/	ØØØØ/AB/AB/AC	AD/BG/GG/GG/AB	
6		L,READSEG		EDITOUT	STARTED		
7				(FOLLOWING SEGMENT(S))			

In this case, the sort segment is preceded and followed by one or more segments. If the sort were the first segment, then the lines preceding the pseudo instruction would be written as in the first example. If the sort were the last segment, then the L, READSEG following the pseudo instruction would be replaced by the L, EXIT instruction. Note that in this example the tag ENTER is a link tag, specifying the entrance to the sort segment in the L, READSEG preceding the SEGMENT card. Had the sort been the first segment, the tag ENTER would be stored directly in the sequence counter, as in the previous case. For the sake of uniformity, it should be assumed, in the following examples, that the sort, together with any own-coding that is included, is a unique program.

If it should be necessary to perform some operation before the sort, the coding to accomplish this could take place in a separate preceding segment, or it could be part of a single-segment sort program such as this:

ARGUS CODING FORM

PROBLEM _____ PROGRAMMER _____ DATE _____ PAGE _____ OF _____

	1 LOCATION 10	11 COMMAND CODE 22	S/C	24 A ADDRESS 37	38 B ADDRESS 51	52 C ADDRESS 65	REMARKS 66 LINE NUMBER 73	74 80
1		PROGRAM		SORTFILE	ONLYSEG			
2		SETLOC		2ØØØ	BØ	G1		
3	Z,SC	SPEC		-	-	START		
4	START			FIRST ORDER OF EXTRA CODING				
5				LAST ORDER OF EXTRA CODING				
6		L,SORT1,S		ØØ/ØØ/	ØØØØ/AB/AB/Ac	AD/GG/GG/GG/AB		
7		L,EXIT						
8		END		SORTFILE	ONLYSEG			
9								

In this case, the first instruction of the program to be performed is START, which is part of the extra coding. At its completion, the sort pseudo instruction follows in sequence, and the sort is performed. The relationship between the extra coding and the pseudo instruction is unimportant; it would have been just as well to have sequence changed to the pseudo instruction. Note, however, that the SETLOC reflects the assignment of the programmer coding to the top of the bank which will precede the sort. Of the 48 words between the address 2000 and the top of the bank, 15 are used by the sort macrocoding, and one by the exit macrocoding; thus 32 words are allotted for the extra coding. (It should be noted that because the MMMM field in the pseudo instruction equals 0000, the sort will not extend storage over the bank boundary.)

Now, if there had been presort own-coding, it might be written as in the following example:

ARGUS CODING FORM

PROBLEM _____ PROGRAMMER _____ DATE _____ PAGE _____ OF _____

	1 LOCATION 10	11 COMMAND CODE 22	S/C	24 A ADDRESS 37	38 B ADDRESS 51	52 C ADDRESS 65	66 REMARKS LINE NUMBER 73	74 80
1		PROGRAM		SORTFILE	ONLYSEG			
2		SETLOC		2000	B0	G1		
3	Z,SC	SPEC		-	-	ENTER		
4	Z,S2	SPEC		-	-	OWNCODE		
5	ENTER	L,SORT 1,S		01/00/	0000/AB/AB/AC	AD/GG/GG/GG/AB		
6		L,EXIT						
7	OWNCODE			FIRST INSTRUCTION OF OWNCODING				
8				LAST INSTRUCTION OF OWN-CODING				
9		END		SORTFILE	ONLYSEG			

This example is very similar to the preceding one, except that the starting location in this case (ENTER) is the sort pseudo instruction. Again, the relationship of the lines of own-coding to the line containing the pseudo instruction is unimportant, since the SPEC constant loaded into Z,S2 indicates the starting address. Note, however, that the exit pseudo instruction is again placed immediately after the sort pseudo instruction since, after the sort, return will be made to this line of coding. Once again, it is assumed that the own-coding requires 32 words or less, and that the program should occupy a minimum amount of space, thus having a SETLOC of 2000. The 01 in the first field of the A address of the sort pseudo instruction indicates that the detour to own-coding is to be made just after the beginning FID record is read from tape AB by the sort modifier-generator.

Now if there should be merge sort own-coding, it might be written as in the following example:

ARGUS CODING FORM

PROBLEM _____ PROGRAMMER _____ DATE _____ PAGE _____ OF _____

	1 LOCATION 10	11 COMMAND CODE 22	S/C	24 A ADDRESS 37	38 B ADDRESS 51	52 C ADDRESS 65	66 REMARKS LINE NUMBER 73	74 80
1		PROGRAM		SORTFILE	SORTSEG			
2		SETLOC		0000	B1	G1		
3	Z,SC	SPEC		-		ENTER		
4	ENTER	L,SORT,S		00/02/OWNCODE	0000/AB/AB/AC	AD/GG/GG/GG/AB		
5		L,EXIT						
6		SEGMENT		SORTFILE	OWNCODE			
7		SETLOC		2000	B0	G1		
8	Z,S2	SPEC		-	-	START		
9	START			FIRST INSTRUCTION OF OWN-CODING				
10				LAST INSTRUCTION OF OWN-CODING				
11		END		SORTFILE	SORTSEG			

This time, the coding is a bit more involved. The first SETLOC specifies the bank for the sort, since it is used only to locate the sort and exit macrocoding. Notice that the segment name OWNCODE is specified in the sort pseudo instruction, where it performs the function of a L,READSEG pseudo instruction. The own-coding, including the setting up of Z,S2 and any SETLOC, must be in a separate segment with this name. The exit pseudo instruction is still located in the same segment as the sort pseudo instruction, and immediately after it. The second SETLOC now specifies that the own-coding will be contained in 48 words. Care must be taken that the merge sort own-coding segment will not overlay the original sort macrocoding.

Now consider the coding to be performed after the sort is finished. As with coding before the sort, this may take place in a separate segment, or it may be part of the segment including the sort:

ARGUS CODING FORM

PROBLEM _____ PROGRAMMER _____ DATE _____ PAGE _____ OF _____

	1 LOCATION 10	11 COMMAND CODE 22	S/C	24 A ADDRESS 37	38 B ADDRESS 51	52 C ADDRESS 65	66 REMARKS LINE NUMBER 73	74 80
1		PROGRAM		SORTFILE	ONLYSEG			
2		SETLOC		2ØØØ	BØ	G1		
3	Z,SC	SPEC		-	-	ENTER		
4	ENTER	L,SORT,S		ØØ/ØØ/	ØØØØ/AB/AB/AC	AD/GG/GG/GG/AB		
5				FIRST INSTRUCTION OF ADDITIONAL CODING				
6				LAST INSTRUCTION OF ADDITIONAL CODING				
7		L,EXIT						
8		END		SORTFILE	ONLYSEG			
9								

The SETLOC should again be specified to allow room for the programmer coding in one bank. Thirty-two words or less of extra coding are assumed, since the two pieces of macrocoding will require 16 words. When the sort is finished, control reverts to the line following the sort pseudo instruction; thus, the extra coding is performed.

One final example is given to show how all of these features may be combined. Of course, any combination of own-coding, or before- and after-coding, may be used, and the arrangement shown is only a suggested approach. To point out some different techniques, assume, for purpose of example, the following: a four-bank machine (implying the existence of the Executive Routine in locations 0000 to 0511 of the first bank), presort own-coding, merge sort own-coding, before- and after-extra coding, each of 100 words, and no other program in parallel. For optimum efficiency, it is required that the sort use as much extra memory as possible. Now

observe the following example:

ARGUS CODING FORM

PROBLEM _____ PROGRAMMER _____ DATE _____ PAGE _____ OF _____

	1 LOCATION 10	11 COMMAND CODE 22	S/C	24 A ADDRESS 37	38 B ADDRESS 51	52 C ADDRESS 65	66 REMARKS LINE NUMBER 73 74 80
1		PROGRAM		SCRAMBLE	MAINSEG		
2		SETLOC		ϕ512	Bϕ	G3	
3	Z,SC	SPEC		-	-	COMMENCE	
4	Z,S2	SPEC		-	-	PSOWNCOD	
5	COMMENCE	FIRST		INSTRUCTION OF BEFORE-SORT EXTRA CODING			
6		LAST		INSTRUCTION OF BEFORE-SORT EXTRA CODING			
7		L,SORT1,ϕ		ϕ3/ϕ3/MERCODE	5316/CA/CA/CB	CC/CD/CE/GG/AB	
8		FIRST		INSTRUCTION OF AFTER-SORT EXTRA CODING			
9		LAST		INSTRUCTION OF AFTER-SORT EXTRA CODING			
10		L,EXIT					
11	PSOWNCOD	FIRST		INSTRUCTION OF PRESORT OWN-CODING			
12		LAST		INSTRUCTION OF PRESORT OWN-CODING			
13		SETLOC		ϕϕϕϕ	B3	G3	
14		SEGMENT		SCRAMBLE	MERCODE		
15		SETLOC		ϕ728	Bϕ	G3	
16	Z,S2	SPEC		-	-	MSOWNCODE	
17	MSOWNCOD	FIRST		INSTRUCTION OF MERGESORT OWN-CODING			
18		LAST		INSTRUCTION OF MERGESORT OWN-CODING			
19		END		SCRAMBLE	MAINSEG		
20							

Several things should be pointed out here. The initial SETLOC of 512 indicates that the before-sort coding will occupy memory locations 512 through 611, the sort macrocoding 612 through 626, the after-sort coding 627 through 726, and the exit macrocoding location 727. The presort own-coding would then occupy 728 through 827. Notice, however, that the second SETLOC places the merge sort own-coding directly over this area. This is entirely feasible since segment MERCODE is loaded only at the time the sort is reloaded, that is, at the beginning of the merge sort. By the same reasoning, Z,S2 is overlaid by the value MSOWNCOD at this time, so that the merge sort may communicate with MSOWNCOD rather than PSOWNCOD. In general, the merge sort own-coding may always overlay the presort own-coding in order to save space. However, care must be taken that the macrocoding and any before- or after-sort instructions are not destroyed. Because of overlaying presort and merge sort own-coding, the highest location reached by programmer coding is 827. 3(2048) minus 828 gives 5316, the amount of memory which the sort my use for item storage in addition to its own bank.

Specific Own-Coding Options

In the preceding section, it was explained how to relate own-coding and before- or after-sort coding to the sort in terms of ARGUS Assembly and the Executive Routine. Now it is appropriate to examine each of the own-coding options in detail, considering what can and cannot be done with each option, as well as reviewing all the significant special registers.

Most of the operations performed, before the operation of the sort commences, would be simple tape positioning routines, since anything more complicated should normally warrant a separate segment or program. Before the sort, it might be necessary to search the input tape for a certain segment or file, or rewind all tapes used by the sort, or position all tapes. The last-mentioned item might be useful at an installation where information is kept in several initial records of each tape. With a small amount of coding attached to the sort to position the required tapes, tapes could be mounted and control could go directly to the sort segment, without loading and performing a complete tape positioning routine. Since no special registers are loaded directly by the sort, the programmer has complete freedom to use any he wishes. In general, in order to facilitate relocation, it would be well to perform all but the simplest operations as a separate segment or program.

Although presort option 02 is the method normally used to modify the sort routine after it is generated, there are occasions when it is necessary to make some modification before generation. It might be necessary to eliminate the reading of the first record from the input tape, in the event that a non-standard FID record is used, or it might even be necessary to eliminate the sort's read routine altogether. Any such changes would require a detailed listing and knowledge of the sort, as well as some means of locating a point in the sort. The former is beyond the scope of this manual. The only tie-point at this time is a SPEC constant in the thirteenth word of the sort macrocoding, which contains the address of the first location of the sort routines. (See Section II, Calling for, Assembling and Executing the Sort, for a list of the sort macrocoding.) Using this location as a base, it is possible to step up to the area of the sort which reads the first record from tape. This can be done by using address arithmetic, and then modifying or negating the coding found there.

Presort option 01 has been provided to allow a standard set of beginning FID sort parameters to be created for files which do not carry such parameters (but which nevertheless must have a standard beginning FID banner word), as well as to allow the revision of parameters which may be there. It allows complete specification of the sort parameters through coding, independent of the data. The transfer to N,S2 is made after the presort modifier has interpreted

the sort pseudo instruction parameters, but before the generator has interpreted the beginning FID record from the input tape. X6 is set by the presort to the first word of this record, which has just been read into memory. Therefore, it may be used to address (through index addressing) any words to be replaced with constants from own-coding. The following registers may not be changed during option 01 coding (unless stored and restored): SC, MXR, UTR, X6, X7, R0, S0, and also the bank bits of CSC.

To illustrate the normal use of option 01, assume that it is necessary to sort a master payroll file. Because this file is already ordered, and therefore not normally sorted, its beginning FID record contains no parameters. It is ordered by employee number in word 1 of the item. However, it is necessary in this case to sort by employee name in words 2 and 3. The items are variable in size, maximum of 30 words each, and packed five to a record. The master file itself will, of course, be saved. The output of the sort will be used as input to a report generator, which will be a separate program following the sort. Here is how such a sort program could be written.

ARGUS CODING FORM

PROBLEM _____ PROGRAMMER _____ DATE _____ PAGE _____ OF _____

	1 LOCATION 10	11 COMMAND CODE 22	S/C	24 A ADDRESS 37	38 B ADDRESS 51	52 C ADDRESS 65	66 REMARKS LINE NUMBER 73	74 80
1		PROGRAM		SORTMF	NAMEFLD			
2		SETLOC		2027	B0	G1		
3	Z,SC	SPEC		-	-	STRTSORT		
4	Z,52	SPEC		-	-	OWNCODN0		
5	STRTSORT	L,SORT2,5		01/00/	0000/BA/BA/BB	BC/GG/GG/GG/BA		
6		L,EXIT						
7	OWNCODNG	TN	C	C,+1	2	6,4		
8		TX	S	Z,SC	-	Z,CSC		
9		DEC		00503001				
10		DEC		00200003				
11		END		SORTMF	NAMEFLD			
12								

This own-coding consists of two instructions and two constants, representing the parameters normally found in the beginning FID record. The TX instruction restores the cosequence counter bits, and the sequence mode is specified, so that control will be returned to the sort after the TN instruction is performed.

Presort option 02 has been provided to allow changes to be made to the presort itself, after it is modified and generated. These changes may be as extensive as the programmer

wishes, and may be made by overlaying or modifying any existing instructions in the sort program. This, of course, requires familiarity with an Assembly listing of the sort routine. The technique used to address the sort, as explained earlier, is to start with some known address in the sort as a base, and to use indirect addressing in the own-coding to step to each word of the sort to be modified. The most convenient tie-point at this time is the sequence counter, which is set to the first instruction of the presort program. The following registers may not be used (unless stored and restored): SC, MXR, UTR, R0, S0, and also bank bits of CSC.

Some typical modifications which could be performed at this option are: extend precision beyond triple; add a detour at the end of presort (to perform summarizing, totalling, or checking functions); or modify or replace the READ area (for instance to read directly from cards). Detailed procedures for these modifications are beyond the scope of this manual, but the general approach for each is given. Extended precision is gained, in the triple-precision sort, by modifying the WL area (common multi-precision routine associated with the tree), as explained under Precision in Section III. Branches at the end of the presort might be installed at any of a number of places, depending on what was to be done at that point, or what was to be changed. A technique for branching off from the presort to own-coding is to replace two of the sort instructions with a TS instruction and a SPEC constant. The SPEC constant can contain the address of the first instruction of own-coding to be performed when the branch is reached. The TS instruction can transfer this SPEC to some unused special register, and go to own-coding indirectly through the special register. Of course, the two instructions replaced in the sort routine should be ones that are no longer useful, or they should be performed in the own-coding area when the branch is made. In changing the READ portion of the presort, a sort should be generated which comes closest to looking like the final version desired by the user. This would suggest generating a sort to handle the item size and record packing to be used throughout presort and merge sort, (specified by the ID record or through presort option 01), and then, at option 02, modifying the read and/or input routine to conform to the input specifications. If, however, input record blocking size is larger than that to be used by the sort, the buffers will have to be set up in own-coding, and the modified items must be supplied to the sort one at a time. The area in the presort involved in modifying reading is the READ area, and also a portion of BEGIN (the initial read).

Presort option 03 has been provided to allow changes to be made to each item before it is processed by the sort. This option differs from the others in that the detour to own-coding is performed any number of times, depending on the number of items, rather than just at one time for the entire sort. The simplest uses of this option are those involving changes within the item: key translating based on a table; key rearranging or compacting into some unused

location within the item; batch totalling; or simple item processing. With a small amount of extra coding, it is possible to duplicate items (expanding a compacted file), or delete items (selective sorting). Item size may also be expanded or decreased to insert new keys, or discard unnecessary information.

When the transfer is made to N, S2 for option 03, the sort is ready to transfer an item from its input buffer to item storage. The item location in the buffer is addressed by X1, and the vacant location in storage is addressed by X7. If control is returned directly to the sort (without changing the sequence counter), the transfer will be performed by the sort. Alternatively, own-coding may perform the transfer and increment the sequence counter by 1 to bypass the transfer in the sort. An item transfer (IT) or TN instruction should be used, since the sort depends on the contents of AU1 and AU2 to modify X1 and to place a word count in the end-of-item symbol of variable-size items. At option 03, the following special registers may not be changed (unless stored and restored): AU1 and AU2 (after item transfer), SC (except as noted), MXR, UTR, X0, X1, X2, X3, X7, R0, R2, R6, R7, S0, S2 (used continuously as a link to own-coding), and also bank bits of CSC. Since the presort uses all registers except S1 and S4 through S7, nothing can be stored by own-coding in any registers except S1 and S4 through S7.

Adding or Deleting Items (Presort)

In adding or deleting items, it is possible to retain the sort's reading and buffering processes. To do this, it is necessary to understand the relationship of own-coding to the sort, as shown in Figure 18. When the option 03 detour occurs, X1 contains the address of the current item in the sort input buffer. Immediately after the detour, the sort will transfer this item to its storage area. (placing a word count in the low-order portion of the end-of-item word if variable-size items are being handled). The sort will then step the input buffer to the next item (reading and switching buffers, if necessary). These functions (transferring, word counting, and buffer stepping) are performed in an area called ITEMTRAN.

The final instruction in the ITEMTRAN area may be addressed at option 03 by incrementing the contents of the sequence counter by 5. This instruction is a TS instruction with all three addresses active. Its C address is pertinent to this discussion. By effectively replacing the C address with a branch to own-coding, a return may be made to own-coding immediately after stepping the buffer and without processing the item. Conversely, own-coding may exit to the location specified by the C address of the sort TS instruction, and hence return to the sort to process the current item without stepping the buffer. When performing ITEMTRAN, but bypassing the complete sorting process of an item, it first is necessary to

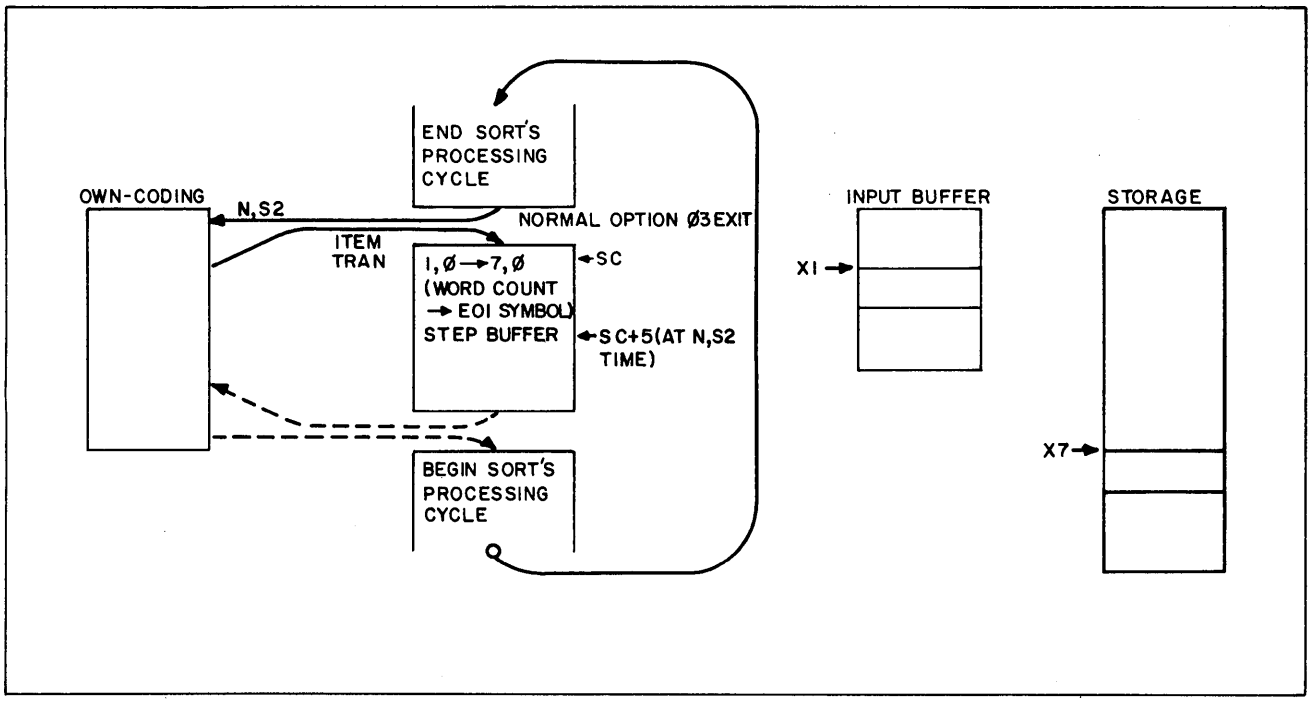


Figure 18. Presort Own-Coding

maintain the transfer from A to B in the last order of ITEMTRAN; this is always Z, AU1 into Z, X1 (stepping the input buffer). This TS instruction may be replaced in the ITEMTRAN area with a simple sequence change to own-coding, where the transfer of Z, AU1 to Z, X1 should be done. The instruction originally at the end of ITEMTRAN should be stored at some time by own-coding, so that it may be replaced when normal operation is desired. Any of the special registers, except those used in ITEMTRAN and the reading routine (X1, X2, X7, AU1, AU2, R7), can be used to transfer control to a particular portion of own-coding from the end of ITEMTRAN, as long as all the necessary ones are restored before returning to the normal sort process. When bypassing ITEMTRAN, the low-order 11 bits of the C address of the final instruction should be substituted into some working special register (the bank designators in the working special register may be obtained from the sequence counter), whereupon this register can be used to return to the sort. Once the substitution has been made, the address thus obtained may be stored by own-coding as a SPEC constant for subsequent use.

Therefore, two connection points between the sort and own-coding must be considered in the addition and deletion of items. One is the normal option 03 detour from sort to own-coding. Own-coding may return to the sort at this point by returning control to the sequence counter (if it has been destroyed, then by restoring the sequence counter and going there). The second connection point is the end of the ITEMTRAN area, which must be modified by own-coding

if a detour is to be made there. Thus a detour from sort to own-coding is made by placing a TS sequence change in the last location of ITEMTRAN, addressed through the sequence counter (at the previous detour) plus 5; own-coding must then perform the final instruction. At this point, a return from own-coding is made by going to the address stored by own-coding from the C address of the original instruction at the end of ITEMTRAN.

Depending on the type of own-coding desired, these two connecting points may be used in a variety of ways. For instance, to add items, it would be necessary to bypass the sort's input buffer stepping (ITEMTRAN). A generated item may be sent to the location specified by X7, and control returned to the sort at the exit of the ITEMTRAN area. In the case of variable-size items, the following instructions will supply the word count as the ITEMTRAN area would have: WD Z, AU2 Z, X7 WORKING, WD Z, AU2 ONE Z, AU2, SS WORKING 16BITS N, AU2, where WORKING is a working location, ONE is a constant of 1 in the right-most position, and 16BITS is a mask of the low-order 16 bits. When an item is to be processed from the sort's input (either as it came or as modified by own-coding), then the ITEMTRAN may be performed in the usual manner or, alternatively, own-coding can transfer the item to 7, 0 and skip the first instruction of ITEMTRAN. In this case, Z, AU1 should be set as it would after transfer of the item from the input buffer, and Z, AU2 should be set as it would be after the transfer of the item to storage. To step through the input buffer without processing items (deleting), ITEMTRAN is performed (transferring the item and stepping the buffer), and then a return to own-coding is made to consider the next item. With this method, one or more unwanted items may be transferred to the same storage location, but they will be overlayed by the next item to be processed.

In expanding or contracting items, the largest item size involved should be specified when stating the sort parameters. Also, the sort should be specified for variable item size, whether the input is variable or not. Own-coding can transfer the item from N, X1 (the item in the input buffer) to its own working area, operate on it (expanding, contracting, adding end-of-item symbol), and then transfer it to N, X7 (the item storage area in the sort). After the first of these transfers, the contents of AU1 must be stored. It must then be restored after the second transfer, before control is returned to the sort, otherwise X1 will be set to the own-coding working area rather than to the sort input buffer. The sequence counter should be incremented by 1 before returning to the sort in order to bypass the sort's transfer instruction.

The following examples illustrate some of the techniques which can be used with own-coding option 03. Corresponding examples in the section on merge sort option 03 will relate to these. EXAMPLEA illustrates a simple key translation, assuming a signed numeric key

(sign and 11 digits) with some positive and some negative quantities. Also, it is necessary to sort so that the output is in strict ascending numeric order: -99...9 through ±00...0 through +99...9. This will occur if all negative numbers are complemented, retaining the zero negative sign, and if careful consideration is given to insure that all positive signs are Gs. Assume for purposes of explanation, a 10-word, fixed-size item, packed 10 to the record, with the numeric key in word 1. The correct parameters are in the file ID record.

ARGUS CODING FORM

PROBLEM _____ PROGRAMMER _____ DATE _____ PAGE _____ OF _____

	LOCATION 1–10	COMMAND CODE 11–22	S/C 23	A ADDRESS 24–37	B ADDRESS 38–51	C ADDRESS 52–65	REMARKS 66–80
1		PROGRAM		EXAMPLEA	ONLYSEG		
2		SETLOC		2ϕ21	Bϕ	G2	
3	Z,SC	SPEC		-	-	START	
4	Z,S2	SPEC		-	-	OWNCODE	
5	START	L,SORT1,ϕ		ϕ3/ϕϕ/	ϕϕϕϕ/CA/CA/CB	CC/CD/GG/GG/CE	
6		L,EXIT					
7	OWNCODE	EX	C	1,ϕ	SIGNMASK	WORKING	
8		NA	C	WORKING	ZERO	C,+3	
9		HA	C	1,ϕ	NUMASK	1,ϕ	
10		TX	S	Z,SC	-	Z,CSC	
11		SS	C	SIGNMASK	SIGNMASK	1,ϕ	
12		TX	S	Z,SC	-	Z,CSC	
13	WORKING	RESERVE		1			
14	SIGNMASK	DEC		G			
15	ZERO	DEC		ϕ			
16	NUMMASK	DEC		ϕGGGGGGGGGGG			
17		END		EXAMPLEA	ONLYSEG		

EXAMPLEB illustrates a useful technique for keeping running batch totals from program to program. Because it would be necessary to use own-coding option 02 in both the presort and merge sort in order to modify the end FID records, it is convenient to have a final filler item, or items, at the end of the file, whose keys are all higher than any possible in the file, so as to contain the final total for the entire file. In EXAMPLEB, assume the same file structure as in EXAMPLEA. Here it is desired to keep a running total of the amount in word 5 (assume a full signed decimal word) and also an item count. These are to be recorded in words 5 and 6 of the dummy item, which consists of words of GG...GF (to distinguish it from other fillers of all hex G's).

ARGUS CODING FORM

PROBLEM _____ PROGRAMMER _____ DATE _____ PAGE _____ OF _____

	LOCATION 1–10	COMMAND CODE 11–22	S/C 23	A ADDRESS 24–37	B ADDRESS 38–51	C ADDRESS 52–65	REMARKS 66–80
1		PROGRAM		EXAMPLEB	ONLYSEG		
2		SETLOC		2ϕ22	Bϕ	G2	

3	Z,SC	SPEC		-	-	START			
4	Z,S2	SPEC		-	-	OWNCODE			
5	START	L,SORT1,Ø		Ø3/ØØ/	ØØØØ/CA/CA/CB	CC/CD/GG/GG/CE			
6		L,EXIT							
7	OWNCODE	NA	C	1,Ø	GSWITHF	C,+3			
8		TN	C	TOTALS	2	1,4			
9		TX	S	Z,SC	-	Z,SC			
10		DA	C	TOTALS	1,4	TOTALS			
11		DA	C	TOTALS+1	ONE	TOTALS+1			
12		TX	S	Z,SC	-	Z,CSC			
13	GSWITHF	DEC		GGGGGGGGGGF					
14	ONE	DEC		+1					
15	TOTALS	DEC		+Ø					
16		DEC		+Ø					
17		END		EXAMPLEB	ONLYSEG				

EXAMPLEC illustrates a useful technique to sort a "compacted" file, which might be an insurance policy file with one item per policy. Within each item, starting with word 11, are the names of family members (if any). The number of additional names is specified in the twelfth digit of word 2. Word 3 contains the family's last name, and word 4, the policyholder's first name. The items are variable size with a maximum of 30 words. It is also desirable, at this point, to set up a cross-reference file, sorted on a double-precision key of last name and then first name, and which represents everyone covered by insurance. The policyholder (original) items are to remain the same, and the cross-reference items are to have a special identification (stating that this is a cross-reference item) in word 1. Also, the policyholder's first name is to be in word 2, the last name in word 3, and the family member's name in word 4. It shall be assumed that the beginning FID record contains the proper parameters to allow sorting on words 3 and 4.

ARGUS CODING FORM

PROBLEM _____ PROGRAMMER _____ DATE _____ PAGE _____ OF _____

	LOCATION (1–10)	COMMAND CODE (11–22)	S/C (23)	A ADDRESS (24–37)	B ADDRESS (38–51)	C ADDRESS (52–65)	REMARKS (66–80)
1		PROGRAM		EXAMPLEC	ONLYSEG		
2		SETLOC		1964	BØ	GZ	
3	Z,SC	SPEC		-	-	START	
4	Z,S2	SPEC		-	-	INITIAL	
5	START	L,SORT2,Ø		Ø3/ØØ/	ØØØØ/CA/CA/CB	CC/CD/GG/GG/CE	
6		L,EXIT					
7	INITIAL	TX	C	Z,SC	-	Z,X4	
8		SS	C	4,5	SUBADDRS	Z,X4	
9		TX	C	Z,X4	-	SKIPITTR	

```
10                TX    C  SPECOWNC      -              Z,S2
11  OWNCODE       EX    C  1,1           G              WORKING
12                NN    C  WORKING       ZERO           C,+2
13                TX    S  Z,SC          -              Z,CSC
14                IT    C  N,X1          DUMMITEM+4     CURRITEM
15                TX    C  SPECBYPS      -              Z,S2
16                TX    C  SPECNAME      -              STORSPEC
17                TX    C  CURRITEM+2    -              DUMMITEM+2
18                TX    C  CURRITEM+3    -              DUMMITEM+1
19                TX    S  Z,SC          -              Z,CSC
20  BYPASS        TX    C  STORSPEC      -              Z,R1
 1                TX    C  N,R1,1        -              DUMMITEM+3
 2                WD    C  WORKING       ONE            WORKING
 3                IT    C  DUMMITEM      DUMMITEM+4     N,X7
 4                TX    C  Z,R1          -              STORSPEC
 5                TX    C  SKIPITTR      -              Z,R1
 6                NA    C  WORKING       ZERO           C,+3
 7                TS    C  SPECOWNC      Z,S2           N,R1
 8                TX    S  Z,SC          -              Z,CSC
 9                TS    S  Z,SC          Z,CSC          N,R1
10  SUBADDRS      DEC      -7GG
11  G             DEC      -G
12  ZERO          DEC      -Ø
13  ONE           DEC      -1
14  SPECOWNC      SPEC     -              -              OWNCODE
15  SPECBYPS      SPEC     -              -              BYPASS
16  SPECNAME      SPEC     -              -              CURRITEM+1Ø
17  DUMMITEM      ALF      CROSSREF
18                RESERVE  4
19  CURRITEM      RESERVE  3Ø
20  WORKING       RESERVE  1
 1  SKIPITTR      SPEC     -              -              -
 2  STORSPEC      SPEC     -              -              -
 3                END      -              -              -
```

The section called INITIAL is performed only when the first option 03 branch to own-coding is made. This section changes the contents of Z, S2 to the normal setting of OWNCODE. This picks up the C address of the last instruction of the ITEMTRAN routine and stores it as a SPEC constant in SKIPITTR. Whenever an item is found containing family names, several working areas are initialized. Z, S2 is temporarily reset to BYPASS, and the item is handled

normally. Each time the sort branches to BYPASS, through the normal option 03 exit, a dummy item is created and sent to N, X7, and the sort is entered through the previously stored exit from the end of ITEMTRAN, thus bypassing the ITEMTRAN routine. When the last dummy item of a particular policy is sent into item storage, Z, S2 is restored to OWNCODE and the normal process resumes in order to consider a new item from the input buffer.

EXAMPLED illustrates the opposite approach of deleting items. Suppose it is necessary to write a "one-shot" program to extract all males over 25 years of age from an employee file and sort them by age, and (within age) by years of service. Again, it can be assumed that the proper information is set up in the beginning FID record. Also, assume the sex code is the first digit of word 2 (0 for female, 1 for male), age is the second and third digits, and years of service the fourth and fifth digits. (The sort can, therefore, be a single-precision sort with mask of 0GGGG0000000.) The own-coding to perform the selection follows:

ARGUS CODING FORM

PROBLEM _____ PROGRAMMER _____ DATE _____ PAGE _____ OF _____

	LOCATION (1–10)	COMMAND CODE (11–22)	S/C	A ADDRESS (24–37)	B ADDRESS (38–51)	C ADDRESS (52–65)	REMARKS (66–80)
1		PROGRAM		EXAMPLED	ONLYSEG		
2		SETLOC		2ϕ11	Bϕ	G2	
3	Z,SC	SPEC		-	-	START	
4	Z,S2	SPEC		-	—	OWNCODE	
5	START	L,SORT1,ϕ		ϕ3/ϕϕ/	ϕϕϕϕ/CA/CA/CB	CC/CD/GG/GG/CE	
6		L,EXIT					
7	OWNCODE	EX	C	1,1	SEXMASK	WORKING	
8		NA	C	WORKING	ONE	DELETE	
9		EX	C	1,1	AGEMASK	WORKING	
10		LA	C	WORKING	TWNTYFIV	DELETE	
11		TX	S	Z,SC	-	Z,CSC	
12	DELETE	TX	C	Z,SC	—	Z,X4	
13		TX	C	4,5	—	STORE	
14		TX	C	RETURN	—	4,5	
15		TX	C	SPECRSTR	—	Z,R1	
16		TX	S	Z,SC	—	Z,CSC	
17	RESTORE	TX	C	Z,AU1	—	Z,X1	
18		TX	C	STORE	—	4,5	
19		TS	C	Z,X4	Z,SC	OWNCODE	
20	SEXMASK	DEC		G			
1	AGEMASK	DEC		ϕGG			
2	ONE	DEC		1			
3	TWNTYFIV	DEC		ϕ25			
4	SPECRSTR	SPEC		-	-	RESTORE	

5	RETURN	TS	C	-	-	N, R1		
6	WORKING	DEC		Ø				
7	STORE	DEC		Ø				
8		END		EXAMPLED	ONLYSEG			

(Note that the second instruction in the DELETE section could have been performed once only, the first time through own-coding, as performed in the preceding example.) The processing of a normal item to be sorted will require only the OWNCODE section. When an item is to be deleted, the DELETE section sets up the exit of the ITEMTRAN portion of the sort to return directly to the RESTORE section of own-coding. Thus ITEMTRAN steps to the next item in the input buffer, reading if necessary, and then transfers control to RESTORE. Here, the transfer of Z, AU1 to Z, X1, normally done at the end of ITEMTRAN, is performed and the exit of ITEMTRAN is restored to its normal setting. OWNCODE is then entered to process the next item.

In the next example, EXAMPLEE, it is desirable that one word be added to each item, which will be an item count to be used as an additional key. This technique is useful if an order-preserving sort is desired, that is, one in which all equal keys will be ordered in the output the same way they were in the input. In this example, there are again 10-word items, fixed length, with keys in word 1. Assume the beginning FID record has been modified to prescribe variable-length items (maximum length of 12 words) with the first key in word 1 and the second key in word 11. Own-coding will be used to expand each item to include the item count in word 11, and an end-of-item symbol in word 12.

ARGUS CODING FORM

PROBLEM _____ PROGRAMMER _____ DATE _____ PAGE _____ OF _____

	1 LOCATION 10	11 COMMAND CODE 22	S/C	24 A ADDRESS 37	38 B ADDRESS 51	52 C ADDRESS 65	REMARKS 66 LINE NUMBER 73	74 80
1		PROGRAM		EXAMPLEE	ONLYSEG			
2		SETLOC		2Ø12	BØ	G2		
3	Z, SC	SPEC		-	-	START		
4	Z, S2	SPEC		-	-	OWNCODE		
5	START	L, SORT2, Ø		Ø3/ØØ/	ØØØØ/CA/CA/CB	CC/CD/GG/GG/CE		
6		L, EXIT						
7	OWNCODE	TN	C	1, Ø	1Ø	WORKING		
8		TX	C	Z, AU1	-	STOREAU1		
9		DA	C	ITEMCNT	ONE	ITEMCNT		
10		IT	C	WORKING	WORKING + 11	7, Ø		
11		TX	C	STOREAU1	-	Z, AU1		
12		TX	S	Z, SC, 1	-	Z, CSC		
13	WORKING	RESERVE		1Ø				

14	ITEMCNT	DEC	+Ø				
15		DEC	Ø				
16	STOREAUI	DEC	Ø				
17	ONE	DEC	+1				
18		END	EXAMPLEE	ONLYSEG			

Of course, if the original file had contained variable-size items, the TN instruction in OWN-CODE would be an IT instruction. In such a case, the first IT would use the B address to dispose of the end-of-item symbol in some unused location. Care would also have to be taken to obliterate the original end-of-item symbol in the WORKING area.

The final example, EXAMPLEF, represents a situation where there is a file of variable-length items (maximum of 30 words), but only the first four words of each item, regardless of its size, are of interest. In this case, it is still necessary to specify a sort which will have storage and buffer capacity for 30-word items. However, the amount of information on tape can be reduced by making all items five words (including end-of-item symbol) thus reducing the overall time of the sort. Once again, a proper beginning FID record, calling for a variable item-size sort (maximum of 30 words) can be assumed.

ARGUS CODING FORM

PROBLEM _____ PROGRAMMER _____ DATE _____ PAGE _____ OF _____

	1 LOCATION 10	11 COMMAND CODE 22	S/C	24 A ADDRESS 37	38 B ADDRESS 51	52 C ADDRESS 65	66 REMARKS LINE NUMBER 73 74 80
1		PROGRAM		EXAMPLEF	ONLYSEG		
2		SETLOC		1996	BØ	G2	
3	Z, SC	SPEC		-	—	START	
4	Z, S2	SPEC		-	-	OWNCODE	
5	START	L, SORTI, Ø		Ø3/ØØ/	ØØØØ/cA/cA/cB	cc/cD/GG/GG/cE	
6		L, EXIT					
7	OWNCODE.	IT	C	N, X1	WORKING	WORKING	
8		TX	C	Z, AUI	—	STOREAUI	
9		IT	C	WORKING	WORKING+4	N, X7	
10		TX	C	STOREAUI	—	Z, AUI	
11		TX	S	Z, SC, 1	—	Z, CSC	
12	WORKING	RESERVE		3Ø			
13	STOREAUI	DEC		Ø			
14		END		EXAMPLEF	ONLYSEG		

It might be considered that a single item transfer instruction, under control of own-coding, would accomplish the same thing. However, it is necessary to perform both transfers in order

that AU1 will be properly set up to find the next item in the input buffer. Thus, one transfer handles the full, original, variable item, while the other handles the new, compacted item. The former is used to pre-set AU1, the latter to set AU2.

Presort option 04 specifies that all options, 01, 02, and 03, are to be observed by the sort. It is therefore necessary that there be own-coding corresponding to each option which, if nothing else, performs the task of setting Z, S2 to the entrance of the next set of own-coding. For example, suppose it should be necessary to perform both the example illustrating option 01 and option 03, (EXAMPLEA).

ARGUS CODING FORM

PROBLEM _____ PROGRAMMER _____ DATE _____ PAGE _____ OF _____

	1 LOCATION 10	11 COMMAND CODE 22	S/C	24 A ADDRESS 37	38 B ADDRESS 51	52 C ADDRESS 65	REMARKS 66 LINE NUMBER 73 74 80
1		PROGRAM		OPTIONØ4	ONLYSEG		
2		SETLOC		2Ø13	BØ	G2	
3	Z,SC	SPEC		-	-	STRTSORT	
4	Z,S2	SPEC		-	-	OWNCODE1	
5	STRTSORT	L,SORT1,Ø		Ø4/ØØ/	ØØØØ/CA/CA/CB	CC/GG/GG/GG/CA	
6		L,EXIT					
7	OWNCODE1	TN	C	C,+2	2	6,4	
8		TX	C	SPECOC2	-	Z,S2	
9		TX	S	Z,SC	-	Z,CSC	
10		DEC		Ø1ØØ1Ø			
11		DEC		ØØ1			
12	SPECOC2	SPEC		-	-	OWNCODE2	
13	OWNCODE2	TX	C	SPECOC3	-	Z,S2	
14		TX	S	Z,SC	-	Z,CSC	
15	SPECOC3	SPEC		-	-	OWNCODE3	
16	OWNCODE3	EX	C	1,Ø	SIGNMASK	WORKING	
17		NA	C	WORKING	ZERO	C,+3	
18		HA	C	1,Ø	NUMMASK	1,Ø	
19		TX	S	Z,SC	-	Z,CSC	
20		SS	C	SIGNMASK	SIGNMASK	1,Ø	
1		TX	S	Z,SC	-	Z,CSC	
2	WORKING	DEC		Ø			
3	SIGNMASK	DEC		G			
4	ZERO	DEC		Ø			
5	NUMMASK	DEC		ØGGGGGGGGGG			
6		END		OPTIONØ4	ONLYSEG		

Merge sort option 01 - It should be emphasized again that this option does not exist. If it should be specified during the generating process, "NO EDIT1" will be printed. Following this there will be an unconditional stop.

Merge sort option 02 has been provided to allow changes to be made to the merge sort itself, after it has been modified and generated. These changes may be as extensive as the programmer wishes, and may be made by overlaying or modifying any existing instructions in the sort program. This, of course, requires an extensive knowledge of an Assembly listing of the sort coding. The technique used to address the sort, as explained earlier, starts with some known address in the sort as a base and, using indirect addressing and address arithmetic in the own-coding, steps to each word of the sort to be modified. The most convenient communicator at this time is the sequence counter, which is set to the first instruction of the merge sort program. The following registers may not be used (unless stored and restored): SC, MXR, UTR, R0, and also bank bits of CSC.

Some typical modifications which could be performed at this option are: extension of precision beyond triple; addition of a detour at the end of merge sorting but before writing the end FID record (for instance to perform summarizing, totalling, or checking functions); or modification or replacement of the write area (for instance to write directly to the printer). Detailed procedures for these modifications are beyond the scope of this manual, but the general approach for each is given. Extended precision is gained, in the triple-precision sort, by modifying the COMMON area (common multi-precision routine associated with the trees), as explained under Precision in Section IV. As pointed out in this section, it will sometimes be necessary also to change the constant K2, which addresses the sort's stopper area. The sort coding called LASTPASS already has the function of making certain modifications to the sort routine just before the last merge pass begins. This area can be modified, or expanded, with own-coding to make even more changes. Thus, the instruction in LASTPASS which sets up a branch to ENDSORT, after the end FID record is written, could be replaced by an instruction to go to a special end section of own-coding rather than to the ENDSORT section. The technique used to branch off from the merge sort to own-coding simply replaces two of the sort instructions with a TS instruction and a SPEC constant. The SPEC can be the address of the first instruction of own-coding to be performed when the branch is hit, and the TS instruction can transfer this SPEC to some unused special register, and go to own-coding through the special register. Of course, the two sort instructions replaced should be ones that are no longer essential to the sort operation, or they should be performed in the own-coding area when the branch is made. Alternatively, the sort could be allowed to finish in a normal manner, and the end FID record could be modified with after-sort coding. In changing the write portion of the

merge sort (presumably only during the last pass), a sort should be generated which comes closest to looking like the final version to fulfill the specified requirements. This suggests generating a sort to handle the item size and record packing to be used throughout the merge sort, as well as between presort and merge sort. It also suggests augmenting the LASTPASS area to modify the output and write routines to conform to the output specifications. If output record blocking size is larger than that which will be used by the sort, buffers must be provided in the own-coding area, as well as providing communication to them during the final pass. Alternatively, by using variable-size items, the entire sort can be generated to the larger specifications. The WRITE area in the merge sort must be modified by LASTPASS in order to change the writing of records involved in the sort.

Merge sort option 03 has been provided to allow changes to be made to each item after it is processed by the sort for the last time. This option differs from 02 in that the detours to own-coding are made any number of times, depending on the number of items, rather than having just one detour at a time. The simplest uses of this option are those involving changes within the item: key unscrambling (based on a table); item restoration (if temporary changes were made by the presort); batch totaling; or simple item processing. It is possible to duplicate items (expand a compacted file based upon the new ordering), or delete items (eliminate duplicates produced by sorting). Item size may also be expanded or decreased to conform to the format of the following routine, or to eliminate temporary keys.

When the sort's transfer is made to N, S2, the sort has just transferred an item to its output buffer from one of its input buffers. The item location in the output buffer is addressed by X0. When control is returned directly to the sort by own-coding (without changing the sequence counter), the output buffer will be stepped. This is accomplished by transferring Z, AU2 to Z, X0 (if variable items); or word differencing the constant 1 from Z, AU2 into Z, X0 (if fixed items). Also, Z, S1, 1 will be checked to determine if it is time to write. If it is not time to write, the sort returns to process another item. At option 03, the following special registers may not be changed (unless stored and restored): AU2 (except as noted), SC, MXR, UTR, X0 (except as noted), X1, X2, X3, X4, X5, R0, R1, R2, R3, R4, R5, S0 (except as noted), S1 (except as noted), S2 (used continuously for link to own-coding), S3, and also the bank bits of CSC. Since the sort uses all but S4 through S7, nothing can be stored by own-coding in any other registers but these.

Adding or Deleting Items (Merge Sort)

In adding or deleting items, it is possible to retain the sort's buffering and writing processes. To do this, it is necessary to understand the relationship of own-coding to the sort,

as shown in Figure 19. When the option 03 detour occurs, X0 contains the address of the item just transferred by the sort to the sort's output buffer. Immediately after the detour, the sort will step the output buffer, writing if necessary, and return to the location addressed by N, S0 to process another item.

The word addressed by N, S0 is the sort's return to its tree, and it is changed during each item processing cycle. By storing and replacing this instruction with a branch to own-coding, return may be made to own-coding immediately after stepping the buffer and without producing another item. Conversely, own-coding may restore the original instruction to N, S0 and go there, and hence return to the sort to produce another item without stepping the buffer. Any of the special registers except those used in the output buffer and writing routine (AU2, X0, X7, S1) can be used to transfer control to a particular portion of own-coding from N, S0. However, it is imperative that all the necessary ones are restored before returning to the normal sort process.

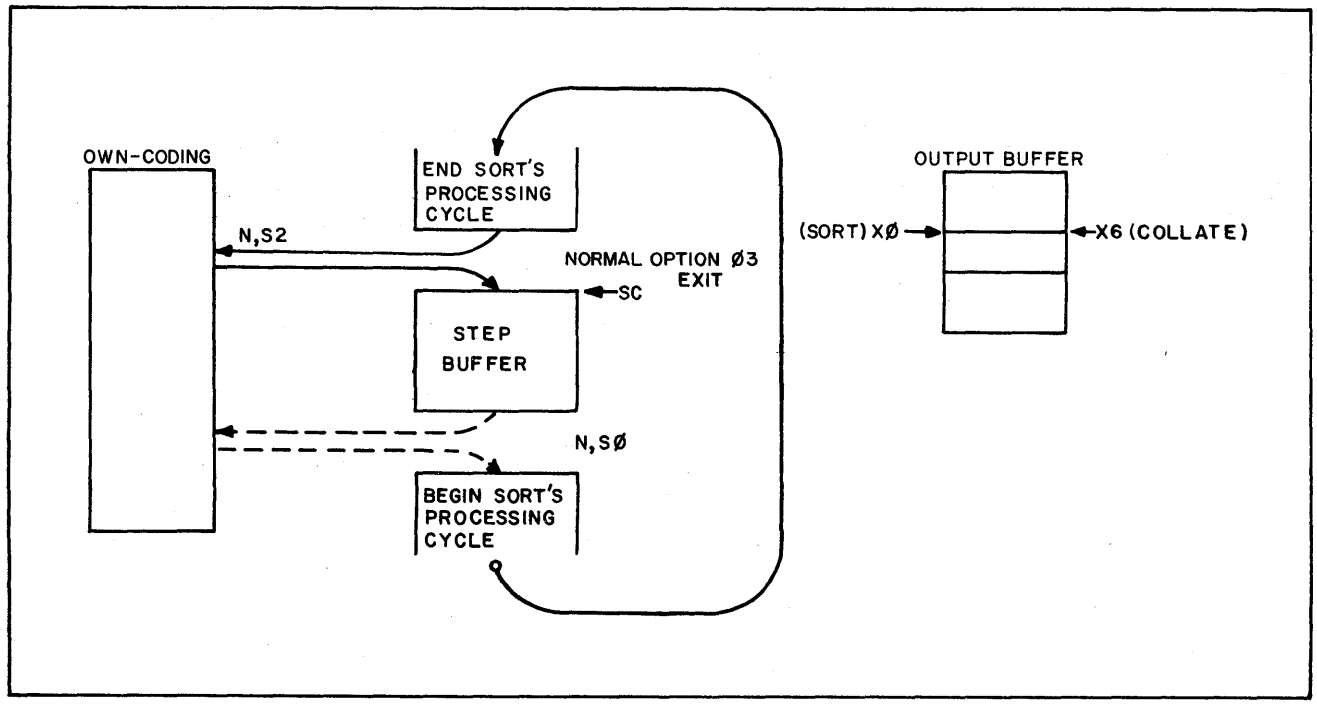


Figure 19. Merge Sort and Collate Own-Coding

Therefore, in adding and deleting items, two connection points must be considered between the sort and own-coding. One is the normal option 03 detour from sort to own-coding. Own-coding may return to the sort at this point by returning control to the sequence counter (if it should have been destroyed, then it can return by restoring the sequence counter and going there). The second connection point is the return to the sort's trees, addressed N, S0, which must be modified by own-coding if a detour is to be made there. Thus, a detour from sort to own-coding is made by placing a TS sequence change into N, S0, having first stored the instruction found

there. A return from own-coding, at this point, is made by restoring the original instruction into N, S0, and going to N, S0.

Depending on the type of own-coding desired, these two connecting points may be used in a variety of ways. For instance in the deletion of items, it is necessary to bypass the sort's output buffer stepping. At each option 03 branch, a return may be made to the sort through N, S0, and one or more items will be overlayed in the output buffer. When an item is transferred to the desired output buffer, either by the sort or by own-coding, a return is made to the sort at the location specified by the sequence counter at the option 03 detour. If own-coding provided the item, care must be taken that AU2 is properly set; this will be the case if an item transfer instruction is always used (rather than an n-word transfer). If successive items are to be produced by own-coding (adding items), N, S0 can be used to return to own-coding after the buffer stepping.

It should be pointed out that either during the addition or deletion of items, when an end of string is reached during the last pass (and hence the end of the sort), the sort then assumes that the last output record has just been written. Consequently, if the input and output counters get out of phase, as they will when adding or deleting, there may be a partial record of output still in the buffer when the sort finishes. To get around this, there should be a full record's worth of filler items in the file, which have keys larger than any legitimate items, and which will therefore be sorted to the end of the file. Alternatively, own-coding can sense for the last valid item. When it is found, it then fills up the output buffer with fillers (Z, S1 will always contain the number of items in the output buffer).

In the expansion or contraction of item sizes, an item size should be specified to the sort which will cover the maximum size involved. The sort should be specified to handle variable item size, whether the final output is variable or not. Own-coding can operate on the item in the output buffer (expanding, contracting, deleting the end-item symbol), and accordingly adjust the output buffer index register (X0) for the next item. The sequence counter should be incremented by 1 before returning to the sort to bypass the sort's output buffer modification.

The following examples should serve to illustrate some of the techniques which can be used with own-coding option 03. These examples correspond to the examples in the section on presort option 03. EXAMPLEA illustrates a simple key translation. Assume for the purpose of explanation, the availability of a signed numeric key (sign and 11 digits), and the fact that the negative quantities have been complemented for sorting purposes. It is then necessary to normalize them and this may be done by re-complementing, retaining the zero sign. Assume

again for purposes of explanation, a 10-word fixed-size item, packed 10 to the record, with the numeric key in word 1. The correct parameters are in the FID record. Although there would probably be corresponding presort own-coding, this is not shown in EXAMPLEA.

ARGUS CODING FORM

PROBLEM _____ PROGRAMMER _____ DATE _____ PAGE _____ OF _____

	LOCATION 1–10	COMMAND CODE 11–22	S/C	A ADDRESS 24–37	B ADDRESS 38–51	C ADDRESS 52–65	REMARKS 66–80 (LINE NUMBER 73–74)
1		PROGRAM		EXAMPLEA	SORTSEG		
2		SETLOC		ØØØØ	B1	G3	
3	Z,SC	SPEC		-	-	START	
4	START	L,SORT1,Ø		ØØ/Ø3/MERGSEG	ØØØØ/CA/CA/CB	CC/CD/GG/GG/CE	
5		L,EXIT					
6		SEGMENT		EXAMPLEA	MERGSEG		
7		SETLOC		2Ø4Ø	BØ	G3	
8	Z,S2	SPEC		-	-	OWNCODE	
9	OWNCODE	EX	C	Ø,Ø	SIGNMASK	WORKING	
10		NA	C	WORKING	ZERO	C,+2	
11		HA	C	Ø,Ø	NUMMASK	Ø,Ø	
12		TX	S	Z,SC	-	Z,CSC	
13	WORKING	DEC		Ø			
14	SIGNMASK	DEC		G			
15	ZERO	DEC		Ø			
16	NUMMASK	DEC		ØGGGGGGGGGGG			
17		END		EXAMPLEA	SORTSEG		

EXAMPLEB illustrates a useful technique for keeping running batch totals from program to program. Because it is necessary to use own-coding option 02 in both the presort and merge sort to modify the end FID records, it is convenient to have a final filler item, or items, at the end of the file, whose keys are all higher than any others possible. This guarantees that it, or they, will remain at the end of the file to contain the final total for the entire file. In this example, assume the same file structure as in EXAMPLEA. It is necessary to keep a running total of the amount in word 5 (assume a full signed decimal word) and also an item count. These are to be compared with words 5 and 6 of the dummy item, which were calculated previously in the same manner. The key of the dummy item consists of a word of GG...GF (to distinguish it from other fillers of all hex G's, which the sort would eliminate).

ARGUS CODING FORM

PROBLEM _____ PROGRAMMER _____ DATE _____ PAGE _____ OF _____

	LOCATION (1–10)	COMMAND CODE (11–22)	S/C	A ADDRESS (24–37)	B ADDRESS (38–51)	C ADDRESS (52–65)	REMARKS (66–80)
1		PROGRAM		EXAMPLEB	SORTSEG		
2		SETLOC		ØØØØ	B1	G3	
3	Z,SC	SPEC		-	-	START	
4	START	L,SORT1,Ø		ØØ/Ø3/MERGSEG	ØØØØ/CA/CA/CB	CC/CD/GG/GG/CE	
5		L,EXIT					
6		SEGMENT		EXAMPLEB	MERGSEG		
7		SETLOC		2Ø33	BØ	G3	
8	Z,S2	SPEC		-	-	OWNCODE	
9	OWNCODE	NA	C	Ø,Ø	GSWITHF	C,+4	
10		NA	C	TOTALS	Ø,4	ERROR	
11		NA	C	TOTALS+1	Ø,5	ERROR	
12		PRA	C	ALFOK	-	C,+3	
13		DA	C	TOTALS	1,4	TOTALS	
14		DA	C	TOTALS+1	ONE	TOTALS+1	
15		TX	S	Z,SC	-	Z,CSC	
16	GSWITHF	DEC		GGGGGGGGGGGF			
17	ONE	DEC		+1			
18	ERROR	PRA	C	ALFERROR			
19		STOP	S	-	-	-	
20	ALFOK	ALF		TOTALSOK			
1	ALFERROR	ALF		TOTALSNG			
2	TOTALS	DEC		+Ø			
3		DEC		+Ø			
4		END		EXAMPLEB	SORTSEG		

Now, in EXAMPLEC, suppose that the file was compacted for the sake of sorting speed, but that for subsequent use it is necessary to expand it. This might be an insurance policy file, with one item per policy carried through the sort. Within each item, and starting with word 11, are the names of family members (if any). The number of additional names is specified in the twelfth digit of word 2. Word 3 contains the family's last name, and word 4 the policyholder's first name. The items are variable size, maximum of 30 words. It is necessary to create trailer items, one for each family member, to follow the header items. The policyholder, or header, items are to remain the same, and the trailer items are to have a special

identification (stating that this is a trailer item) in word 1, the policyholder's first name in word 2, the last name in word 3, and the family member's name in word 4. It shall be assumed that the beginning FID record contains the proper parameters to allow sorting on words 3 and 4 of the original items. Note that this will result in a file sorted by policyholder, with trailers following the header items in the order that the names appear in the policyholder items. By contrast, EXAMPLEC of presort option 03 results in a file completely ordered on name, regardless of the original grouping by family.

ARGUS CODING FORM

PROBLEM _____ PROGRAMMER _____ DATE _____ PAGE _____ OF _____

	1 LOCATION 10	11 COMMAND CODE 22	S/C	24 A ADDRESS 37	38 B ADDRESS 51	52 C ADDRESS 65	REMARKS 66 LINE NUMBER 73 74 80
1		PROGRAM		EXAMPLEC	SORTSEG		
2		SETLOC		ØØØØ	BI	G3	
3	Z,SC	SPEC		-	-	START	
4	START	L,SORT2,Ø		ØØ/Ø3/MERGSEG	ØØØØ/CA/CA/CB	CC/CD/GG/GG/CE	
5		L,EXIT					
6		SEGMENT		EXAMPLEC	MERGSEG		
7		SETLOC		1983	BØ	G3	
8	Z,S2	SPEC		-	-	OWNCODE	
9	OWNCODE	EX	C	Ø,1	G	WORKING	
10		NN	C	WORKING	ZERO	C,+2	
11		TX	S	Z,SC	-	Z,CSC	
12		IT	C	N,XØ	DUMMITEM÷4	CURRITEM	
13		TX	C	N,SØ	-	STORETRN	
14		TX		Z,SC	-	STORESC	
15		TX	C	TSBYPASS	-	N,SØ	
16		TX		SPECBYPS	-	Z,R7	
17		TX	C	SPECNAME	-	Z,R6	
18		TX	C	CURRITEM+2	-	DUMMITEM	
19		TS	C	CURRITEM+3	DUMMITEM+1	OWNCODE+2	
20	BYPASS	NA	C	WORKING	ZERO	C,+2	
1		TX	C	STORETRN	-	N,SØ	
2		TS	S	Z,SC	Z,CSC	N,SØ	
3		TX	C	N,R6,1	-	DUMMITEM÷3	
4		WD	C	WORKING	ONE	WORKING	
5		IT	C	DUMMITEM	DUMMITEM÷4	N,XØ	
6		TX	C	STORE	-	Z,SC	
7		TX	S	Z,SC	-	Z,CSC	

```
8   G          DEC      -G
9   ZERO       DEC      -Ø
10  ONE        DEC      -1
11  TSBYPASS   TS       -          -          N,R7
12  SPECBYPS   SPEC     -          -          BYPASS
13  SPECNAME   SPEC     -          -          CURRITEM+1Ø
14  DUMMITEM   ALF      TRAILER
15             RESERVE  4
16  CURRITEM   RESERVE  3Ø
17  WORKING    DEC      Ø
18  STORETRN   DEC      Ø
19  STORSPEC   SPEC     -
20  STORSC     SPEC     -          -          -
1              END      EXAMPLEC   SORTSEG
```

Whenever an item is found containing family names, several working areas are initialized, N, S0 (merge) is temporarily reset to transfer control to BYPASS, and control is returned to the sort to step and interrogate the buffer counters. Each time the sort branches to BYPASS, a check is made to see if any more dummy items are to be created. If not, N, S0 is restored and the sort is re-entered to continue in a normal manner. Otherwise, a dummy item is created and sent to the output buffer through N, X0. The sort is then re-entered at the normal point where the output buffer is incremented and tested. Whether or not it is time to write, the sort will eventually go to N, S0 (merge) to process the next item, and then back to the BYPASS portion of own-coding. Note that the general approach here is slightly different from EXAMPLEC of presort option 03. There, the dummy item was first produced and then checked to see if there should be more to follow, whereas here a test is first made to see if there are more dummy items, and then produce one if necessary. This difference relates to the fact that, in one case, items are being brought in to the sort while in the other case they are being put out of the sort. Presort own-coding is performed when the sort wants an item, whereas merge sort own-coding is performed when an item has been produced by the sort.

Now, in EXAMPLED, assume the opposite case from EXAMPLEC, namely, the deletion of items. Suppose that after sorting, a file will contain many items with duplicate keys, in which case it may be necessary to delete the duplicates. Again it shall be assumed that the proper information is set up in the beginning FID record. This then is a single-precision sort, with the key in word 1.

ARGUS CODING FORM

PROBLEM _____ PROGRAMMER _____ DATE _____ PAGE _____ OF _____

	LOCATION (1–10)	COMMAND CODE (11–22)	S/C (23)	A ADDRESS (24–37)	B ADDRESS (38–51)	C ADDRESS (52–65)	REMARKS (66–80)
1		PROGRAM		EXAMPLED	SORTSEG		
2		SETLOC		ØØØØ	B1	G3	
3	Z,SC	SPEC		-	-	START	
4	START	L,SORT1,Ø		ØØ/Ø3/MERGSEG	ØØØØ/CA/CA/CB	CC/CD/GG/GG/CE	
5		L,EXIT					
6		SEGMENT		EXAMPLED	MERGSEG		
7		SETLOC		2Ø42	BØ	G3	
8	Z,S2	SPEC		-	-	OWNCODE	
9	OWNCODE	NA	C	LASTKEY	Ø,Ø	C,+3	
10		TS	S	Z,SC	Z,CSC	N,SØ	
11		TX	C	Ø,Ø	-	LASTKEY	
12		TX	S	Z,SC	-	Z,CSC	
13	LASTKEY	DEC		Ø			
14	ONE	DEC		-1			
15		END		EXAMPLED	SORTSEG		

Here, in order to delete an item, the processing of the output buffer is simply bypassed, so that the next item produced by the sort will overlay the one to be deleted.

For the next example, suppose it is necessary to add one word to each item so that it can be used as a serial number by a later run. A variable-item-size sort must have been specified by using own-coding in the presort, if necessary, so as to convert fixed-size items to variable (not shown). Assume that the items are, indeed, truly variable size, with a maximum size of 30 words. This means that the sort will have to be specified with a maximum item size of 31 words so as to make room in the output buffer for the expanded items. Own-coding will be used to expand each item to include the serial number in the position before the first word of the item, thus moving the rest of the item "up" one place.

ARGUS CODING FORM

PROBLEM _____ PROGRAMMER _____ DATE _____ PAGE _____ OF _____

	LOCATION (1–10)	COMMAND CODE (11–22)	S/C (23)	A ADDRESS (24–37)	B ADDRESS (38–51)	C ADDRESS (52–65)	REMARKS (66–80)
1		PROGRAM		EXAMPLEE	SORTSEG		
2		SETLOC		ØØØØ	B1	G3	
3	Z,SC	SPEC		-	-	START	

4	START	L, SORT1, ϕ		ϕϕ/ϕ3/MERGSEG	ϕϕϕϕ/CA/CA/CB	CC/CD/GG/GG/CE			
5		L, EXIT							
6		SEGMENT		EXAMPLEE	MERGSEG				
7		SETLOC		2ϕ11	Bϕ	G3			
8	Z, S2	SPEC		-	-	OWNCODE			
9	OWNCODE	IT	C	ϕ, ϕ	ϕ, 29	STORAGE			
10		DA	C	COUNTER	ONE	COUNTER			
11		TX	C	COUNTER	-	ϕ, ϕ			
12		IT	C	STORAGE	STORAGE + 29	ϕ, 1			
13		TX	S	Z, SC	-	Z, CSC			
14	STORAGE	RESERVE		3ϕ					
15	COUNTER	DEC		+ϕ					
16	ONE	DEC		+1					
17		END		EXAMPLEE	SORTSEG				

Notice that, in this case, X0 was not modified in own-coding, but instead returned directly to the sort to perform this. This was possible because the second item transfer instruction set up AU2 correctly, so that the sort's buffer modifying instruction would function correctly. Had the item size been changed directly in the buffer, by moving the end-of-item symbol with a TX, for instance, then it would have been necessary to modify X0 in own-coding, step the sequence counter by 1, and then return to the sort.

For the final example, again assume a file of variable-length items, maximum of 12 words. Assume that presort option 03, EXAMPLEE, had created these items from 10-word fixed-size items, by adding a key in word 11 and an end-of-item symbol in word 12. It is necessary to eliminate words 11 and 12 in order to make the output consist of fixed-length records of 10 words each. It shall be assumed that all items coming from the sort are exactly 12 words in length.

ARGUS CODING FORM

PROBLEM _____ PROGRAMMER _____ DATE _____ PAGE _____ OF _____

	LOCATION (1–10)	COMMAND CODE (11–22)	S/C (24)	A ADDRESS (24–37)	B ADDRESS (38–51)	C ADDRESS (52–65)	REMARKS (66–80)
1		PROGRAM		EXAMPLEF	SORTSEG		
2		SETLOC		ϕϕϕϕ	B1	G3	
3	Z, SC	SPEC		-	-	START	
4	START	L, SORT2, ϕ		ϕϕ/ϕ3/MERGSEG	ϕϕϕϕ/CA/CA/CB	CC/CD/GG/GG/CE	

```
5                L,EXIT
6                SEGMENT          EXAMPLEF    MERGSEG
7                SETLOC                       80          G3
8   Z,S2         SPEC             -           -           OWNCODE
9   OWNCODE      TX           C   Z,X0,10     -           -
10               TX           S   Z,SC,1      -           Z,CSC
11               END              EXAMPLEF    SORTSEG
```

Merge sort option 04 specifies that the exits for 02 and 03 are both to be observed by the sort. It is therefore necessary that option 02 coding set up Z, S2 for the option 03 coding, before returning to the sort.

After sort coding - Since, at this time, the sort has completed its operation, there are no restrictions on the use of special registers, as there are before the sort. Of course, any special registers loaded with the routine will have been destroyed by the sort, so these should be loaded by the coding instead. Since nothing done at this time affects the sort, no examples of this type of coding are shown.

Collate option 01 has been provided to allow a standard set of beginning FID parameters to be created for the collate in cases where they do not exist on tape, or to allow for revision of parameters which may be there. All original files read by the collate must still have standard beginning-of-file banner words. However, since input to the collate will presumably have come from the sort routines, the correct parameters will normally be already present on tape. This option allows complete specification of the collate parameters through coding, independent of the data. The transfer to N, S2 is made after the collate generator has interpreted the collate pseudo instruction parameters, but before it has interpreted the beginning FID record from the initial A input tape. Index register X7 is set to the first word of this record, in memory, and X7 may be used to address any of the parameters which will be replaced with constants from own-coding. The following registers may not be changed by the own-coding (unless stored and restored): SC, MXR, UTR, X0, X7, and also bank bits of CSC.

For an illustration of the normal use of option 01, refer to the above section on presort option 01 which, except for specific special register addresses, is similar.

Collate option 02 has been provided to allow changes to be made to the collate routine itself, after it is generated. Here the distinction must be made between the initial "once-only" generation of the collate, and that portion of the generator which is performed before each pass.

Option 02 causes a detour immediately after the former. The changes made at this time may be as extensive as the programmer wishes, and may be made by overlaying or modifying any existing instructions in the sort program. This, of course, requires a thorough knowledge of a listing of the collate routine. The technique used to address the collate, like the sort, is to start with a known address in the collate as a base and, using address modification in the own-coding, to step to each word of the collate to be altered. The most convenient communicator to the collate by the own-coding is the sequence counter, which addresses a location at the end of the A3 section, the last portion of the once-only generator. Because there are several breaks in the sequence of instructions performed during generation, the exact location of the branch to own-coding should be noted on the coding sheets; the sequence counter will be set to the location following this one. The following registers may not be used (unless stored and restored): SC, MXR, UTR, R0, X0, and also bank bits of CSC.

Some typical modifications which could be performed at this option are: extend precision beyond triple; add a detour at the end of each pass (or at the end of the final pass); perform specialized operations or modify the WRITE area. Because of the connections established between passes of the collate, the last function (mentioned above) should be performed during the last pass only. Detailed procedures for these modifications are beyond the scope of this manual, but the general approach for each is given. Since the collate uses the same tree as the merge sort, extended precision is handled as described under merge sort option 02 in this section. The collate coding, called FNAME, already has the function of making certain modifications to the collate routine just before the last pass begins. This area can be modified, or expanded, with own-coding to make more extensive changes. Thus, the FNAME instruction, which sets up a branch in ENDPASS to end the routine, may be replaced so as to set up a branch to a special end section of own-coding. One technique for branching off from the collate to own-coding is to replace two of the collate instructions with a TS and a SPEC constant. When the branch is taken, the SPEC can be the address of the first instruction of own-coding to be performed, and the TS can transfer this SPEC to some unused special register, and go to own-coding under cosequence control through the special register. Of course, the two collate instructions replaced should be either instructions that are no longer useful, or else they should be performed in the own-coding area when the branch is made. Alternatively, the collate could finish in a normal manner, and perform any special end function (such as totaling or checking) after the collate. In changing the WRITE portion of the collate (presumably during the last pass only), a collate should be generated which will handle the structure of the input and intermediate files. Then, at option 02, the FNAME area could, in turn, be augmented to modify the output and/or write routines to conform to the output specifications. If output record blocking size is larger than that to be used by the collate, separate buffer areas must be provided in the

own-coding, and in the connections to own-coding, for use during the final pass. Alternatively, by using variable-size items, the entire collate can be generated to the largest specifications. The area in the collate which must be modified by FNAME to change item transfer or writing is the JJ area.

Collate option 03 has been provided to allow changes to be made to each item after it is processed by the collate for the last time, i.e., during the final pass. This option differs from the other two in that the detour to own-coding is performed more than once, depending on the number of items. The simplest uses of this option are those involving changes within the item: item rearrangement; batch totaling; or simple item processing. It is possible to duplicate items (expand the new file), or delete items (eliminate duplicates). Item size may be increased or decreased to conform to the format of the succeeding routine or to conform to the system.

When the transfer is made to own-coding through N, S2, the collate has just transferred an item to its output buffer from one of its input buffers. The item location in the output buffer is addressed by X6. When control is returned directly to the collate (without changing the sequence counter), the output buffer will be stepped (by transferring Z, AU2 to Z, X6), and Z, R6, 1 will be checked to determine whether it is time to write. If writing is not indicated, a return is made to process another item. At option 03, the following special registers may not be changed (unless stored and restored): AU2 (except as noted), SC, MXR, UTR, X0, X1, X2, X3, X4, X5, X6 (except as noted), R0, R1, R2, R3, R4, R5, R6 (except as noted), S0 (except as noted), S2 (used as continuous link to own-coding), S3, and also bank bits of CSC. Since the collate uses all but S4 through S7, nothing can be stored by own-coding in any registers but these.

Adding or Deleting Items (Collate)

In the addition or deletion of items, it is possible to retain the collate's buffering and writing processes. To do this, it is necessary to understand the relationship of own-coding to the collate, as shown in Figure 19. When the option 03 detour occurs, X6 contains the address of the item just transferred by the collate to its output buffer. Immediately after the detour, the collate will step the output buffer, writing if necessary, and return to the location addressed by N, S0 to process another item.

The word addressed by N, S0 is the collate's return to its tree, and is changed during each item processing cycle. By storing and replacing this instruction with a branch to own-coding, a return may be made to own-coding immediately after stepping the buffer, and without producing another item. Conversely, own-coding may restore the original instruction to N, S0 and go there, and then return to the collate to produce another item without stepping the buffer. Any

of the special registers, except those used in the output buffer and writing routine (AU2, X6, R6, R7), can be used to transfer control to a particular portion of own-coding from N, S0, as long as all the necessary buffers are restored before returning to the normal collate process.

Therefore, in the addition and deletion of items, two connection points between the collate and own-coding must be considered. One is the normal option 03 detour from collate to own-coding; own-coding may return to the collate at this point by returning control to the sequence counter (if it has been destroyed, then by restoring the sequence counter and going there). The second connection point is the return to the collate's trees, addressed N, S0, which must be modified by own-coding if a detour is to be made there. Thus a detour from collate to own-coding is made by placing a TS sequence change into N, S0, having first stored the instruction found there. A return from own-coding at this point is made by restoring the original instruction into N, S0, and going to N, S0.

Depending on the type of own-coding desired, these two connecting points may be used in a variety of ways. For instance, to delete items, it is necessary to bypass the collate's output buffer stepping. At each option 03 branch, a return may be made to the collate via N, S0, and one or more items will be overlayed in the output buffer. When an item is transferred to the desired output buffer, either by the collate or by own-coding, a return is made to the collate at the location specified by the sequence counter at the option 03 detour. If own-coding provided the item, care must be taken that AU2 is properly set; unlike the merge, a TN instruction should be used for fixed-size items, with less than 63 words per item, and an IT instruction should be used for variable-size items or fixed-size items greater than 63 words per item. If a succession of items is to be produced by own-coding (adding items), N, S0 can be used to return to own-coding after the buffer stepping.

It should be pointed out that either during the addition or deletion of items, when end of pass is reached during the last pass (and hence the end of the collate), the collate then assumes that the last output record has just been written. Consequently, if the input and output counters get out of phase, as they will when adding or deleting, there may be a partial record of output still in the buffer when the collate finishes. To get around this, a full record of filler items at the end of the file should be produced (coming from any combination of inputs). Alternatively, own-coding can sense for the last valid item, and when found, fill up the output buffer (if necessary) with fillers. (Z, R6 will always contain the number of items in the output buffer.)

In the expansion or contraction of item sizes, an item size should be specified for the collate which will cover the largest size involved. The inputs must therefore be variable-size

items and it should be specified to the collate whether or not the final output is available. Own-coding can operate on the item in the output buffer (expanding, contracting, deleting end-of-item symbol), and can adjust the output buffer index register (X6) for the next item accordingly. The sequence counter should be incremented by 1 before returning to the sort, so as to bypass the collate's output buffer modification.

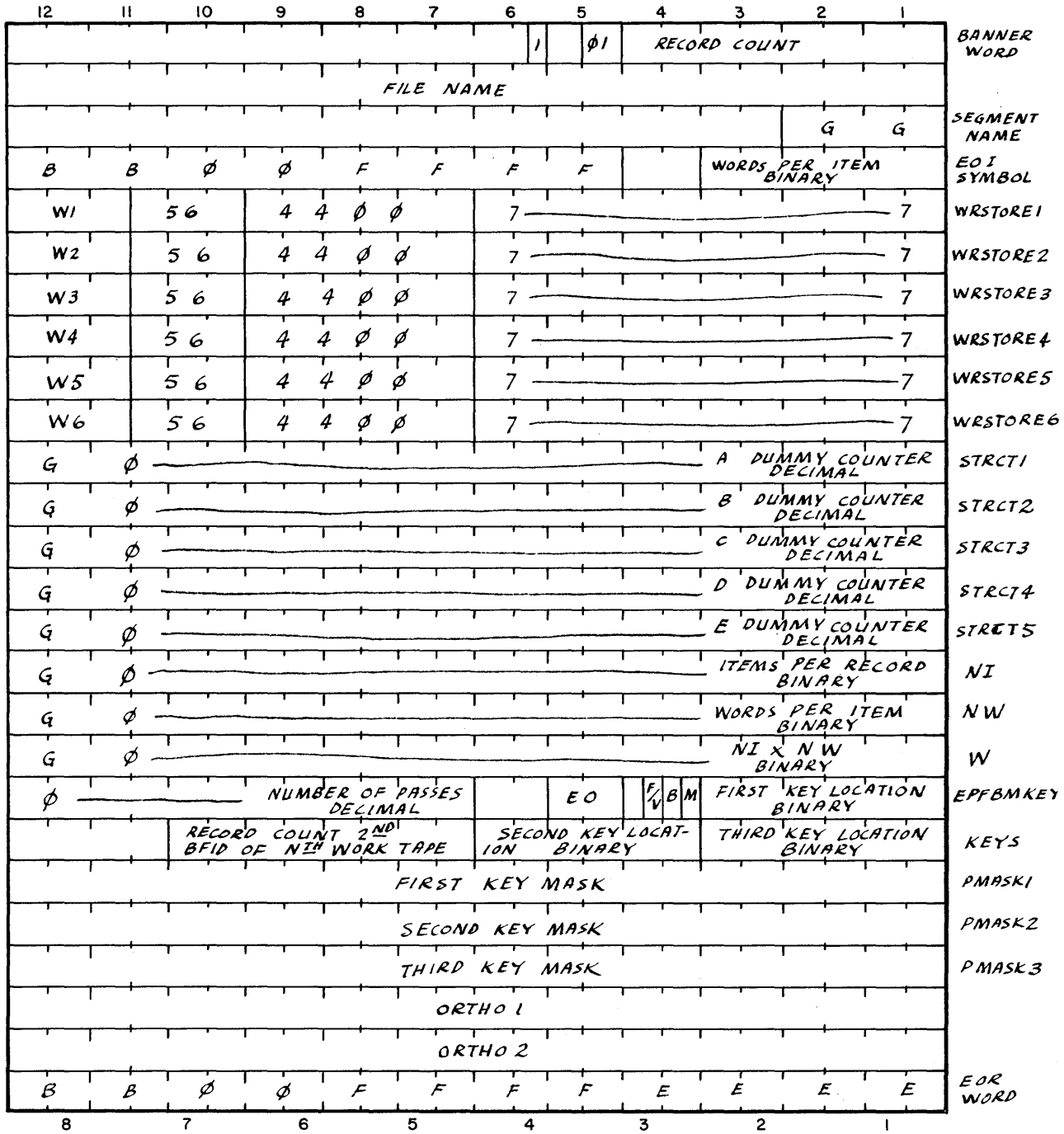
Since collate option 03 own-coding is similar to merge sort option 03 own-coding, the reader is referred to that section for appropriate examples. The only differences between these examples and the ones which would be prepared for a collate are the addresses of the special registers.

Collate option 04 specifies that options 01, 02, and 03 are all to be observed by the collate. It is therefore necessary that there be own-coding corresponding to each option which, if nothing else, resets Z, S2 to refer to the next option. The reader is referred to presort option 04, contained in this section, for an example of this.

APPENDIX A

END FILE IDENTIFICATION RECORD (ITEM DESIGN) PRESORT TO MERGE SORT

Included in this appendix is an item design of each word in the End File Identification Record (FID).



APPENDIX B

PRESORT SPECIAL REGISTERS

X Bin setting.

X1 Input buffer for item transfer and EDOFILE check.

X2 Input buffer for reading.

X3 Output buffer for item transfer.

X4 Output buffer for writing;

Stopper address for positioning and searching tape.

X5 Working index register:
1. In BEGFID for file ID reference;
2. In WRITE for CC.

X7 Working index register:
1. Macrocoding;
2. In FILSTR, MASTER, ITEMTRAN and EDCON for smallest item in storage.

R For restarts.

R1 Working special register (as counter):
1. In BEGIN1 for items in storage;
2. In FILBIN for keys in storage;
3. In SWITCH and RESET to reference 14th location of bin.

R2 Working special register (as counter):
1. In BEGIN1 to count words per item;
2. In FILSTR to count items in storage;
3. In FILBIN for filling bin with keys;
4. In EDCON to count words per item.

R3 Working special register (as counter):
1. In BEGIN1 to count items in storage;
2. In FILBIN to count items in storage;
3. In DSCALC to count two levels.

R4 Working special register (as counter):
1. In FILBIN to count items per bin;
2. In ENDSTR to count words per item;
3. In CHKIT to count items in storage.

R5 Working special register to address keys of item in storage in CHKIT.

R6 Counter for items per output buffer.

R7 Counter for items per input buffer.

S2 Own-coding.

AU1 Item in input buffer.

AU2 Item in output buffer.

APPENDIX C

MERGE SORT SPECIAL REGISTERS

X0	Output buffer.
X1-X5	Items in input buffers.
X6	Input buffer for reading.
X7	Working index register: 1. In WRITE for CC; 2. In EOF and LASTPASS to address STOPPER.
R0	For restarts.
R1-R5	Counters for items in input buffers.
R6	Working special register: 1. In VARSWCH as counter; 2. In BEGPASS for TABLE address.
R7	Working special register: 1. In VARSWCH to set item index register and LASTKEY; 2. In BEGPASS and ENDPASS for address in READ.
S0	Tree special register.
S1	Item counter for output buffer (used to determine when to read and write).
S2	Own-coding.
S3	EOF counter for "way merge" (used to reset EXITA).

APPENDIX D

COLLATE SPECIAL REGISTERS

X0	Current entry of plan.
X1-X5	Items in input buffers.
X6	Items in output buffer.
X7	Working index register: 1. In READ for reading; 2. In MHSKEEP for reading.
R0	For restarts.
R1-R5	Counters for items in input buffers.
R6	Counter for items in output buffer.
R7	Working special register: 1. In JJ for writing; 2. In MHSKEEP to reference read orders; 3. In BEGIN portion of Generator.
S0	Tree return special register.
S1	Working special register: 1. In MHSKEEP to reference KEEP location; 2. In Generator.
S2	Working special register: 1. In Generator.

APPENDIX E

TIMING OF HONEYWELL 800 SORT ROUTINES

Although there are tables with which sort times may be readily determined, these have certain limitations, particularly in the area of record packing. The following formulas may be used to determine times for any specific case.

Given the following factors:

Wb = words per bin:
  single and multi precision = 3.0
  double precision = 4.2

Wi = words per item (including EOI symbol if used).

Wm = words of memory available (2,024 + MMMM).

Wp = words in presort program:
  single precision = 500
  double and multi = 550

Wr = words per record (including banner, two ortho, and EOR word).

If = items per file (volume to be sorted).

Ir = items per record.

Is = items per strings = $2\frac{(Wm-Wp-4Wr)}{(Wi+Wb)}$

Rs = records per string = Is/Ir

D = number of tape drives used by merge sort.

S = number of strings produced by presort = If/Is

P = number of passes, from following table of maximum values of S:

P	D = 3	D = 4	D = 5	D = 6
2	3	6	10	15
4	8	31	85	190
6	21	157	707	2353
8	55	793	5864	29056
10	144	4004	48620	
12	377	20216		
14	987			
16	2584			
18	6765			
20	17711			

Trt = time to pass one record on tape, including gap.

Tf = time to pass entire file = $Trt\frac{(If)}{(Ir)}$

Td = time for dummy items produced by presort = $Trt\frac{(S)}{(2)}$

Tip = time per item, presort (given in memory cycles):

Is	Single Prc.	Double Prc.	Multi Prc.
1-12	4Wi + 125	4Wi + 156	4Wi + 156
13-72	4Wi + 171	4Wi + 218	4Wi + 229
73-432	4Wi + 219	4Wi + 280	4Wi + 302

Trp = time per record, presort (given in memory cycles) = 3Wr + 113

Tsp = time per string, presort (given in memory cycles):
single and multi precision = 12.1(Is) + 69
double precision = 17.0(Is) + 79

Tim = time per item, merge sort (given in memory cycles):

D	Single Prc.	Double Prc.	Multi Prc.
3	2Wi + 59	2Wi + 62	2Wi + 64
4	2Wi + 62	2Wi + 65	2Wi + 69
5	2Wi + 64	2Wi + 69	2Wi + 74
6	2Wi + 67	2Wi + 73	2Wi + 79

Trm = time per record, merge sort (given in memory cycles):

D	Single Prc.	Double Prc.	Multi Prc.
3	3Wr + 188	3Wr + 191	3Wr + 193
4	3Wr + 193	3Wr + 198	3Wr + 203
5	3Wr + 198	3Wr + 206	3Wr + 213
6	3Wr + 203	3Wr + 213	3Wr + 223

Fp = factor, presort = $\frac{(Ir)(Tip) + Trp + Tsp/Rs}{Trt}$

Note: For timing purposes, if Fp is less than 1, the value 1 must be used. In such cases, Fp represents the CP capacity used by the presort. For Wi less than 5, Fp is always greater than 1. For Wi greater than 15, Fp is always less than 1.

Fm = factor, merge sort = $\frac{(Ir)(Tim) + Trm}{Trt}$

Note: For timing purposes, if Fm is less than 1, the value 1 must be used. In such cases, Fm represents the CP capacity used by the merge sort. Fm is usually less than 1.

Tp = time for presort = (Tf)(Fp) + Td

Tm = time for merge sort = P(Tf)(Fm) + 2Td

Tt = total time for sort = Tp + Tm

A single pass collate routine may be readily timed in a manner similar to timing one pass of the merge sort. In this case, the factor "If", representing items per file, is the number of items in the output file, or the sum of the items on each input. Timing a multi-pass collate is more difficult, since different portions, and therefore different quantities of data, are handled during each pass. For these cases, it is necessary to determine the amount of

data being processed during each pass, and to time each pass accordingly. The total time is then the sum of the times for all passes.

Given the following factors:

Wi = words per item (including EOI symbol if used).

Wr = words per record (including banner, two ortho, and EOR word).

If = items per file (volume to be collated in one pass).

Ir = items per record.

W = way.

Trt = time to pass one record on tape, including gap.

Tf = time to pass entire (output) file = $Trt \frac{(If)}{(Ir)}$

Tic = time per item, collate (given in memory cycles):

W	Single Prc.	Double Prc.	Multi Prc.
2	2Wi + 51	2Wi + 57	2Wi + 60
3	2Wi + 53	2Wi + 60	2Wi + 64
4	2Wi + 56	2Wi + 65	2Wi + 69
5	2Wi + 58	2Wi + 68	2Wi + 72

Trc = time per record, collate (given in memory cycles):

Single Prc.	Double Prc.	Multi Prc.
3Wr + 179	3Wr + 189	3Wr + 194

Fc = factor, collate = $\frac{(Ir)(Tic) + Trc}{Trt}$

Note: For timing purposes, if Fc is less than 1, the value 1 must be used. In such cases, Fc represents the CP capicity used by the collate. Fc is usually less than 1.

Tc = time for collate = (Tf)(Fc)

A GLOSSARY OF SORTING TERMS

Ascending	Relating to ascending order; that is, a progression from the smallest alphanumeric key (all zeros) to the largest (all hex G's).
Bin	A storage area in memory, used in the presort, which contains a number of tags and related coding. The bin is arranged to coincide with the tree, so that the smallest tag contained in the bin may be found and transferred out.
Buffer	A working area in memory, to or from which data is read or written on tape. Buffers are usually used in pairs or sets, so that data may be read or written in one buffer, while other data is processed in the other buffer.
Collate	A routine which has as its input any number (up to 99) of ordered (sorted) files, and as its output a single ordered file containing all of the input data.
Data	Numeric and alphabetic information supplied to and processed by a computer. Data differs from a program in so much as the program supplies the computer with the logical step-by-step instructions to process this data. In the sorts, data is that which is sorted through a presort, merge, and collate program.
Descending	Relating to descending order; that is, a progression from the largest alphanumeric key (all hex G's) to the smallest (all zeros).

Error Routine	A section of programming, initiated automatically by a read-write error UTR, which attempts correction of the error by an orthotronic correction routine and/or by rereading.
File	A set of data used as input to a sort or collate. The final result of a sort or collate sequence.
Generator	A general routine, usually used with a modifier, which accepts specifications of a specific sort (parameters) and produces a routine meeting requirements. The generator sets up (generates) the variable portion of a routine (bins, buffers, etc.).
Item	The unit of data which is manipulated by the sorts.
Key	A set of characters, usually forming a field, which is the portion of an item used as a criterion for the alphanumeric arrangement of that item with other items.
Machine Limited	The condition which exists when the time used in processing a given amount of data exceeds the time used in moving this data between internal and external storage. Thus, in such a sort, a tape must periodically wait for the machine to finish processing data.
Merge	The process used in the ARGUS merge sort and collate which reads several ordered strings or files of input and, through a series of comparisons, selects the smallest (or largest), item by item, and from these comparisons produces one ordered string or file for output.
Merge Sort	The second and last portion of a sort routine which performs a series of merging operations until all data is combined into a single string of ordered information.

Modifier — A general routine, usually used with a generator, which accepts specifications of a specific sort (parameters) and produces a routine meeting requirements. The modifier sets up (modifies) the fixed portion of a routine (trees, switches, transfer instructions, etc.), using a skeleton routine as a basis.

N-1 — A sorting technique, more properly called "Cascade" sorting, developed by Honeywell and used by the ARGUS merge sorts.

Ordered — That which has been sorted; a term usually used to describe a sequence of items whose keys have been arranged in alphanumeric order.

Own-coding — A portion of program or additional coding, which is added to a sort and to which the sort detours at prescribed intervals. It is most often used on a item-by-item basis to modify data upon first reading by the sort, or upon final writing.

Parameter — A statement of prescribed format which specifies, to a modifier-generator, a specific sort to be generated. Such factors as item size, key position, tape allocation, etc., are usually specified in a parameter. In the ARGUS sort system, parameters are supplied in two portions: through the macrocoding routine which calls the sort, and through the beginning FID record of the data tape.

Presort — The first portion of a sort routine which reads a single tape of random data as input, and writes as output two or more tapes of ordered strings.

Precision — Size of key used for sorting, usually in terms of computer words. A single-word key is single

Precision (cont)	precision, a double-word key is double precision, a triple-word key is triple precision.
Random	Having no specified order. In sorting, random is the opposite of "ordered".
Restart	A section of programming, initiated by the console operator in case of trouble, which "backs up" the program to a specified point and "starts again".
Segment (as related to a sorting operation)	A portion of a file, usually a single tape.
Set	Used in the merge sort or collate to include everything pertaining to one of the inputs; usually designated A through E.
Skeleton	A basic, generalized, sort routine which provides the framework on which modification and generation can build a sort program. Such a routine cannot be run in itself, since important areas are missing.
Sort	A routine (consisting of a presort and merge sort) which reads random data as its input, and which writes the same data as its output, in alphanumeric order, based on a prescribed part of the data (key).
Stopper	A programming device, also called "freezing", which is used to indicate that an item in storage is temporarily not to be considered for sorting. (It should not be confused with the hardware definition of stopper.)
String	A portion of data on tape, and not necessarily on the entire tape, which has been ordered through a presort. A string is indicated by a special banner word in its first record.

Tag	A condensation of an item, consisting of a key (used for sorting) and an identification word (used to identify and relocate the rest of the item in memory, after the key of that item has been determined to represent the smallest item). The ID word is appended to the key to form a tag. The result is that which is actually shuffled around in memory during the sort.
Tape (data)	Data which exists on a single reel of magnetic tape, random or ordered. Such data may represent part of a file (string or segment), or a full file.
Tape Limited	The condition which exists when the time used in moving data from tapes to internal storage, or from internal storage to tapes, exceeds the time used in processing that data. Thus, in such a sort, the machine must periodically wait for the continuously running tape.
Tree	A logical array of machine instructions having one entrance, and one of several exits, the latter depending on which of several keys related to the tree is smallest (or largest).
Way	Presort: number of tags in a bin which the tree compares (maximum of six). Merge Sort and Collate: number of input sets which the tree compares (maximum of five).
Work Tape	A tape used by the sorts during the sorting process, the contents of which are immaterial before and after the sort routine.

NOTES

Honeywell

Electronic Data Processing